ArgQL to SPARQL Translation and Proof

Dimitra Zografistou[1]

Institute of Computer Science, ICS-FORTH, Heraklion, Crete,
dzograf@ics.forth.gr

In this report, we formally and extensively present the proccess of translation from ArgQL query language, to domain-specific SPARQL queries, executed against AIF structured data, stored in RDF. First we give the formal specification for the RDF and the SPARQL's syntax and semantics. Then, we formally describe our argumentative data model and the ArgQL query language (syntax and semantics). In the end, we show the process of translation, followed by a proof that verifies its soundness.

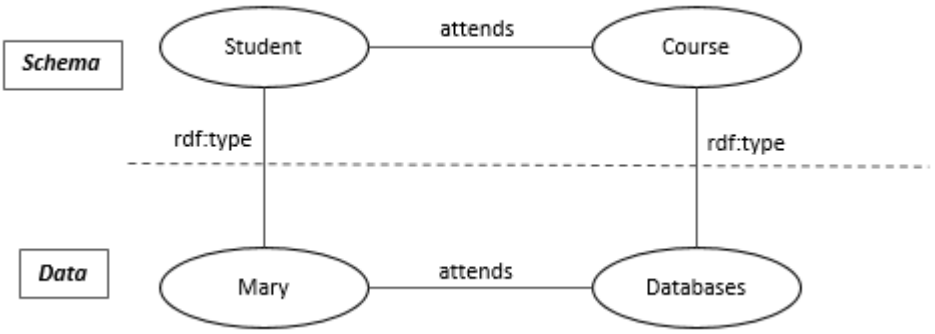
1 RDF and RDFS

RDF[5] 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 1 (RDF triple and RDF graph) *An RDF triple t is a tuple (s,p,o)* $\in \boldsymbol{U} \times \boldsymbol{U} \times (\boldsymbol{U} \cup \boldsymbol{L})$*, 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 [3] 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].

The following example shows a graph model and an excerpt of its RDF/XML representation.



```
<rdf:Description rdf:about=ex:Databases>
   <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Course/>
</rdf:Description>
<rdf:Description rdf:about=ex:Mary>
   <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Student"/>
   <ex:attends rdf:resource=ex:Databases"/>
</rdf:Description>
```

---

2 SPARQL

SPARQL[6] 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, V_S and UVL, UV, UL the sets $U \cup V_S \cup L$, $U \cup V_S$ 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 2 (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 3 (Graph pattern) *A graph pattern is defined by the following abstract grammar:*

$$gp \rightarrow tp \mid gp\ AND\ gp \mid gp\ OPT\ gp \mid gp\ UNION\ gp \mid gp\ FILTER\ expr$$

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 [6]. 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 4 (SPARQL query) *A SPARQL query sparql is defined as:*

$$sparql \rightarrow Select\ varlist\ Where\ gp$$

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

According to [7], the semantics of SPARQL are defined in terms of the following mapping function:

Definition 5 (Variable mapping) *Let a mapping* $\mu : V_S \rightarrow UL$ *be a partial function that assigns RDF terms of an RDF graph, to variables of a SPARQL query.*

Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, *dom*(μ), is the subset of V_S over which μ is defined. Two mappings μ_1 and μ_2 are *compatible*, (written as $\mu_1 \sim \mu_2$), if $\mu_1(x) = \mu_2(x)$ for all variables $x \in dom(\mu_1) \cap dom(\mu_2)$. 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:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\},$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$
$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2,\ \mu \text{ and } \mu' \text{ are not compatible}\},$$
$$\Omega_1 :\bowtie \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}.$$

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

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

Informally, the evaluation of a graph pattern over an RDF graph G is the set of triples $\mu(tp) \in G$ obtained by applying the evaluated mappings μ in each type of a graph pattern.

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

expr is bound(?X) and ?X $\in$ dom(μ);
expr is ?X op l, ?X $\in$ dom(μ) and μ(?X) op l, where op $\rightarrow < \mid \leq \mid \geq \mid > \mid =$;
expr is ?X op ?Y, ?X,?Y $\in$ dom(μ) and μ(?X) op μ(?Y), where op $\rightarrow < \mid \leq \mid \geq \mid > \mid =$;
expr is $(\neg expr_1)$ and it is not the case that $\mu \vDash_S expr_1$;
expr is $(expr_1 \vee expr_2)$ and $\mu \vDash expr_1$ or $\mu \vDash_S expr_2$;
expr is $(expr_1 \wedge expr_2)$, $\mu \vDash expr_1$ and $\mu \vDash_S expr_2$;

Again here, we use the notation $\vDash_S$ to distinguish the expression satisfaction of SPARQL, from the filter satisfiability of ArgQL in which we use the same symbol.

Property Paths were introduced as part of SPARQL 1.1 in 2013. Property Paths make it possible to define queries that match with an arbitrary amount of edges.

Definition 6 (Property Paths) *sPo is a property path, where $s \in V \cup U$, $o \in U \cup L \cup V$ and P is a property path expression. Furthermore. sPo is a basic graph pattern.*

There are several different property path expressions, which denote different paths. These different expressions and their syntax are presented in the following definition.

Definition 7 (Property Path expression) *A path expression may have one of the following forms:*
$p \in U$ is a property path expression.
ˆP with property path expression P, is the inverse property path expression.
P_1/P_2, with property path expressions P_1 and P_2, is the sequence property path expression.
$P_1|P_2$, with property path expressions P_1 and P_2, is the alternative path expression.
P∗, with property path expression P, is the transitive reflexive closure property path expression.
P+, with property path expression P, is the transitive closure property path expression.

Below we show an example of a SPARQL query containing a property path expression.

```
PREFIX ex: http://example.com/
PREFIX foaf: http://xmlns.com/foaf/0.1/
SELECT ?friend WHERE { <ex:Mary> <foaf:knows>* ?friend. }
```

The evaluation of a property path is presented in the following definition

Definition 8 (Evaluation of property paths) *For constants $s \in U$, $o \in U \cup L$ and variables $v, v_1, v_2 \in V_S$, the evaluation of property paths is defined as:*
$\mathcal{E}(sPo,G) = \{\mu \mid if\ (s,p) \in \mathcal{E}(P,G)\}$, $\varnothing$ *otherwise*
$\mathcal{E}(vPo,G) = \{\mu \mid (\mu(v),o) \in \mathcal{E}(P,G) \wedge dom(\mu) = \{v\}\}$
$\mathcal{E}(sPv,G) = \{\mu \mid (s,\mu(v)) \in \mathcal{E}(P,G) \wedge dom(\mu) = \{v\}\}$
$\mathcal{E}(v_1Pv_2,G) = \{\mu \mid (\mu(v_1),\mu(v_2)) \in \mathcal{E}(P,G) \wedge dom(\mu) = \{v_1,v_2\}\}$

The semantics of the evaluation of property path expressions are given bellow:

Definition 9 (Evaluation of property path expressions) *The evaluation $\mathcal{E}(P,G)$ of a property path expression P over an RDF graph G, is a set of pairs of RDF terms from $U \cup V$ defined as follows:*

$\mathcal{E}(p,G) = \{(s,o) \mid (s,p,o) \in G\}$,
$\mathcal{E}(\hat{}P,G) = \{(s,o) \mid (o,s) \in \mathcal{E}(P,G)\}$,
$\mathcal{E}(P_1/P_2,G) = \mathcal{E}(P_1,G) \circ \mathcal{E}(P_1,G)$,
$\mathcal{E}(P_1|P_2,G) = \mathcal{E}(P_1,G) \cup \mathcal{E}(P_1,G)$,
$\mathcal{E}(P+,G) = \bigcup_{i\geq 1} \mathcal{E}(P^i,G)$,
$\mathcal{E}(P*,G) = \mathcal{E}(P+,G) \cup \{(s,o) \mid (s,p,o) \in G\}$,

where $\circ$ is the usual composition of binary relations, and p^i is the concatenation $p/\ldots/p$ of i copies of p.

3 Data Model in Structured Argumentation

In this section we describe the theoretical principles that rule the proposed data model. In order to keep our language as general as possible and keep it independent of particular logics (like propositional, first-order etc.), we adopt Tarski's abstract model [**?**]. In particular, our argumentation framework is defined by the language $\mathcal{L} =< \mathcal{P}, \vdash >$, where $\mathcal{P}$is an infinite set of well-formed formulas, called propositions and $\vdash$denotes the *logic inference* between subsets of $\mathcal{P}$. Given two sets X, Y $\subseteq \mathcal{P}$, we write that X $\vdash$Y to denote that Y is implied by X. The inference operator is loaded with the following properties:

For all X, Y $\subseteq \mathcal{P}$, if X $\subseteq$ Y, then Y $\vdash$X *(reflexivity)*
For all X, Y, Z $\subseteq \mathcal{P}$, if X $\vdash$Y and X $\cup$ Y $\vdash$Z, then X $\vdash$Z *(transitivity)*
For all X,Y,Z $\subseteq \mathcal{P}$, if X $\vdash$Y, then X $\cup$ Z $\vdash$Y *(weakening)*

In the rest of this work, we assume an arbitrary, but fixed, logic $<\mathcal{P},\vdash>$. The inference process allows to define the concepts of equivalence and contrariness as follows:

Definition 1 (Equivalence). *We say that two sets X, Y $\subseteq \mathcal{P}$are equivalent (denoted by X $\equiv$Y) if and only if X $\vdash$Y and Y $\vdash$X.*

Definition 2 (Contrariness). *We say that two sets $X,Y \subseteq \mathcal{P}$ are contrary (denoted by X $\nvdash$Y) if and only if X $\cup$ Y $\vdash \mathcal{P}$.*

We abuse notation for the operators $\vdash$, $\nvdash$and $\equiv$and allow them to exists also between single propositions, without using the brackets $\{.\}$ For example, given two propositions $x, y \in \mathcal{P}$ we write $x \nvdash y$, instead of $\{x\} \nvdash \{y\}$

Definition 3 (Consistency). *A set $X \subseteq \mathcal{P}$ is inconsistent wrt a logic $(\mathcal{P},\vdash)$ iff X $\vdash \mathcal{P}$. Otherwise, it is consistent.*

An argument is a special case of inference, in which the inferred set is a singleton. That proposition is assumed to be the central claim, justified by a set of zero or more propositions. Formally:

Definition 4 (Argument). *An argument of the logic $\mathcal{L} =< \mathcal{P}, \vdash >$ is a pair $\langle pr, c \rangle$, where $pr \subseteq \mathcal{P}, c \in \mathcal{P}$, pr is a consistent and minimal set, and it holds that pr $\vdash$c. pr is called premise and $c \in \mathcal{P}$ is called conclusion. $\mathcal{A}$ is the infinite set of arguments.*

Given an argument *a*, *prem(a)* is called its premise set and *concl(a)* its conclusion.

We discern two general types of relations between arguments, *attack* and *support*, which are further specialized in two subrelations each, as follows:

Definition 5 (Argument relations). *We define the following relations among arguments:*

- *Rebut:* $R^{rebut} = \{(a_1,a_2) \mid a_1,a_2 \in \mathcal{A}$ *and* $concl(a_1) \nvdash concl(a_2)\}$
- *Undercut:* $R^{ucut} = \{(a_1,a_2) \mid a_1,a_2 \in \mathcal{A}$ *and* $concl(a_1) \nvdash p$ *for* $p \in prem(a_2)\}$
- *Attack:* $R^{att} = R^{rebut} \cup R^{ucut}$
- *Endorse:* $R^{rebut} = \{(a_1,a_2) \mid a_1,a_2 \in \mathcal{A}$ *and* $concl(a_1) \equiv concl(a_2)\}$
- *Backing:* $R^{back} = \{(a_1,a_2) \mid a_1,a_2 \in \mathcal{A}$ *and* $concl(a_1) \equiv p$ *for* $p \in prem(a_2)\}$
- *Support:* $R^{sup} = R^{end} \cup R^{back}$

The set $R = R^{att} \cup R^{sup}$ includes all types of relations. It is easily provable, that R^{att} and R^{sup} are disjoint. In this work we assume knowledge bases that consist of finite sets of arguments, conformed to the described theoretical concepts which are called dialogues.

Definition 6 (Dialogue). *A dialogue D is a knowledge base of $\mathcal{L}$ that consists of a finite set of arguments A. The infinite set of all possible such knowledge bases is denoted by $\mathcal{D}$.*

Relations between arguments are not explicitly expressed in the knowledge base, but exist in the satisfaction of conflicts and equivalences, in the definition 5. The abstract view of a dialogue, forms a graph, whose nodes are structured arguments of A and the edges are the four different types of structured sub-relations.

4 ArgQL Query Language

ArgQL is built on the idea of pattern matching, an inherently declarative approach. Provided is a number of well-defined patterns(motifs), especially designed to fulfill the informational requirements described in the introduction. Below, we present ArgQL syntax, by showing how each of the category of requirements is satisfied.

4.1 Syntax

Syntax has been influenced by Cypher [4], the language for the Neo4j database and SPARQL 1.1 [6], the standard query language for RDF. Both of them address graph data models. Initially, we assume an infinite set V of variables. Variables are prefixed with '?' and are used to bind with values from the data. For constant propositions values, we use the quotes "."(e.g. "p"). An interesting feature here is that, given a constant value "p", all of its equivalent propositions existing in the knowledge base are also valid matches.

Individual argument identification. To allow for identifying particular arguments from the knowledge base, we introduce a fundamental unity of the language, *argument pattern.* Syntactically, an argument pattern can be either a single variable $v_a \in V$, or have the form v_a:⟨*premisePattern, conclusionPattern*⟩. PremisePattern and conclusion-Pattern, are also patterns that restrict the premise and conclusion part of arguments, respectively. More precisely, an argument pattern can have one of the following forms

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

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

· $\langle ?v[/\{"p_1"\}], ?c \rangle$: match arguments whose premise include some equivalent proposition of "p_1".
· $\langle ?v, "c" \rangle$: match arguments with conclusion any equivalent proposition of *"c"*
· $\langle ?v[.\{"p_1","p_2"\}], ?c \rangle$: match arguments whose premise intersect with a set equivalent to $\{"p_1","p_2"\}$
· $\langle \{"p_1","p_2"\}, "c" \rangle$: instant arguments are also considered as argument patterns

The case that an argument pattern is a single variable and the one that it has the form $\langle ?pr, ?c \rangle$ are equivalent and they both match all arguments in the knowledge base.

Correlated argument identification. The correlations we discern here, mainly concern the relevant content in the arguments' premises. In particular, ArgQL allows to discover pairs of arguments, such that, one's premise is a subset of the other's, whose premises intersect, or even whose premises are completely irrelevant. For this purpose, we use the filtering mechanism, except that, here the restriction is based on a premise variable of an external argument pattern. Some representative examples are the following:

·$\langle ?pr_1, "c" \rangle$,$\langle ?pr_2[/?pr_1], ?c_2 \rangle$:match arguments with conclusion "c" and then, match arguments the premise of which are superset of the first matching arguments

· $\langle ?pr_1, ?c_1 \rangle$, $\langle ?pr_2[!?pr_1], ?c_2 \rangle$: finds pairs of arguments with disjoint premises.

Argument interactions discovery. ArgQL provides the possibility to detect implied relations between arguments, based on the existing conflicts and equivalents between the involved propositions. In particular, it provides six keywords, to discover the six different types of relations: `rebut, undercut, attack, endorse, back, support`.

· $\langle ?pr_1, ?c_1 \rangle$ `rebut` $\langle ?pr_2, "c" \rangle$: find arguments rebutting those with conclusion "c".

Dialogue traversing/Searching for sub-dialogues. ArgQL offers mechanisms that address the graph view of a dialogue, and give the ability to navigate in a graph of arguments. To do this, it uses *path patterns*. As their name reveal, path patterns are expressions that describe paths in a graph and they match sequences of arguments that satisfy these paths. Syntactically, they always lie between two argument patterns. They are recognized by sequences of relations separated by the character '/', (e.g.`attack/support/support`). The expression **n* is a syntactic sugar to express the "*n* repetitions of a path pattern". For example `rebut*2`, is an alternative of `rebut/rebut`. In addition, we can express the case "up to *n* repetitions", by using the notation *'+n'*. In particular `attack+3`, defines three different path patterns: {`attack, attack/attack, attack/attack/attack`}. The coexistence of multiple number indicators in the same pattern, gives a number of combinations, equal to the proliferation of the '+' indicators.

· *?a* `(attack*2/support)+2` $\langle ?pr_2, "c" \rangle$: matches with dialogue lines (sequences of arguments and relations), satisfying either the *attack/attack/support* or the *attack/attack/support/attack/attack/support* and result to an argument with conclusion *"c"*.

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

Definition 7 (ArgQL query.). *An ArgQL query q is defined as*

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

where varlist = $(v_1, v_2, \ldots, v_n)$, with $v_i \in V$ is a list of variables

The *match* clause of an ArgQL query may contain one or more dialogue patterns. A dialogue pattern can be either a single argument, or an alternating sequence between argument patterns and path patterns. The complete syntax is shown bellow. The highlighted symbols inside the quotes ' ' denote tokens of the language. Anything included in (.)? is optional while the notation (.)* implies zero or more occurrences. The language is not case sensitive, so keywords like *"attack"*, or *"ATTACK"* are recognized as equal. To unify the different patterns, we define the set *M* which includes all the supported patterns.

dialogue_pattern	←	argpattern |
		argpattern pathpattern dialogue_pattern
argpattern	←	variable |
		(variable:)? '**⟨**' premise_pattern '**,**' conclusion_pattern '**⟩**'
premise_pattern	←	propset_pattern | variable ('**[**'premise_filter'**]**')?
premise_filter	←	('**/**' | '**.**' | '**!**') (proposition_set | variable)
conclusion_pattern	←	variable | proposition
propset_pattern	←	'**{**' proposition ('**,**' proposition)* ' **}** '
pathpattern	←	relation |
		pathpattern '**/**' pathpattern **|**
		'**(** 'pathpattern'**)**' ('*****' | '**+**') num
relation	←	**'attack'** | **'rebut'** | **'undercut'** |
		'support' | **'endorse'** | **'back'**
variable	←	**'?'**(**'a'..'z'**—**'A'..'Z'**)(**'a'..'z'** —**'A'..'Z'**—'_'—**'0'..'9'**)*
num	←	(**'0' .. '9'**)+
proposition	←	'"' .*? '"'

Below we show some examples of complete ArgQL queries. Argument values are from dialogues in [1] where the topic of discussion is "For or Against Cloning".

Q1. Find arguments which "defend"(attack the attackers of) arguments with conclusion equivalent to "Cloning is going to be awesome" and return both of them. This query could capture specific cases from Dung semantics [**?**].

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

[1] http://www.createdebate.com/debate/show/Cloning_8

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

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

Q3. Find pairs of arguments whose premise sets join each other and return them.

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

4.2 Semantics

In this section we show formally how each particular component pattern is evaluated in a dialogue D. We assume an interpretation function I_D, which takes as input a pattern $t \in M$ and returns the set of binding values from D. Moreover, assuming the set $S = \mathcal{P} \cup PR \cup A$, where $PR = 2^{\mathcal{P}}$, we also define a variable replacement $\mu : V \mapsto S$, as a mapping that associates variables with elements from S. The logical content of D, denoted as $P(D)$, is the set of the involved propositions. I_D is defined as:

Definition 8 (Pattern Interpretation I_D). *Let a dialogue D, constituted by a finite set of arguments A and $P(D)$ its logical content. Interpretation function I_D is defined as:*

- *for t a constant proposition $t \in \mathcal{P}$. The interpretation of t is the set of all the equivalent propositions of t: $I_D(t) = \{p \in P(D) \mid t \equiv p\}$*
- *for t a proposition_set of the form $t = \{t_1, \ldots, t_n\}$, where t_i constant propositions:*
  $$I_D(t) = \{\{t'_1, \ldots t'_n\} \mid t'_i \in I_D(t_i)\}$$
- *for t a premise_pattern:*
  - *t is a proposition_set t': $I_D(t) = \{pr \in I_D(t') \mid prem(a) = pr \text{ for some } a \in A\}$*
  - *if t is of the form $v[f]$, where $v \in V$ and f a filter, then*
    $$I_D(t) = \{\mu(v) \subseteq lc(D) \mid \mu(v) = prem(a) \text{ for some } a \in A \text{ and } \mu(v) \vDash f\}$$
    ($s \vDash f$, where $s \in PR$, means that the set s satisfies the filter and its semantics is given separately after the current definition)
- *for t a conclusion_pattern:*
  - *if t is a constant proposition t', then:*
    $$I_D(t) = \{p \in I_D(t') \mid concl(a) = p \text{ for some } a \in A\}$$
  - *if t is a variable v: $I_D(t) = \{\mu(v) \in lc(D) \mid \mu(v) = concl(a), \text{for some } a \in A\}$*
- *for t an argpattern:*
  - *if t is a variable $v \in V$ then: $I_D(t) = A$*
  - *if t is of the form $v : \langle pr, c \rangle$, where $v \in V$, pr a premise_pattern and c a conclusion_pattern, then: $I_D(t) = \{a \in A \mid prem(a) \in I_D(pr) \text{ and } concl(a) \in I_D(c)\}$*
- *for t a relation pattern:*
  - *$I_D(rebut) = \{(a_1 a_2) \mid a_1, a_2 \in A \text{ s.t. } concl(a_1) \not\equiv concl(a_2)\}$*
  - *$I_D(undercut) = \{(a_1 a_2) \mid a_1, a_2 \in A \text{ s.t. } concl(a_1) \not\equiv p_i \text{ for some } p_i \in prem(a_2)\}$*
  - *$I_D(attack) = I_D(rebut) \cup I_D(undercut)$*
  - *$I_D(endorse) = \{(a_1 a_2) \mid a_1, a_2 \in A \text{ s.t. } concl(a_1) \equiv concl(a_2)\}$*
  - *$I_D(back) = \{(a_1 a_2) \mid a_1, a_2 \in A \text{ s.t. } concl(a_1) \equiv p_i \text{ for some } p_i \in prem(a_2)\}$*
  - *$I_D(support) = I_D(endorse) \cup I_D(back)$*
- *for t a pathpattern.*
  - *if t is a relation pattern t', then $I_D(t) = \{(a_1 a_2) \mid \in I_D(t')\}$*
  - *if t is of the form t_1/t_2, for t_1, t_2 pathpatterns then*
    $$I_D(t) = \{(a_1 \ldots a_m \ldots a_n) \mid (a_1 \ldots a_m) \in I_D(t_1) \text{ and } (a_m \ldots a_n) \in I_D(t_2)\}$$
  - *if t is of the form $(t') * n$, which is translated into $t'/\ldots/t'$ (n times) we have:*
    - *if $n = 1$, then $I_D(t) = I_D(t')$*
    - *if $n > 1$, then $I_D(t) = I_D(t'/(t') * (n-1))$*

- *if t is of the form $(t')+n$, which is translated into a set of pathpatterns $(t')+n = \bigcup_{k=1}^{n}(t' * k)$:*
  $I_D(t) = \bigcup_{k=1}^{n} I_D((t') * k)$

– *for t is a dialogue_pattern. A dialogue pattern t in general can be written as:*
  $ap \mid ap\ pp\ dp'$
  We use the name head(dp) to denote the first argument pattern that appears in the sequence of argument patterns in the dialogue pattern dp. We have:
  - *if t an argument pattern ap, then: $I_D(t) = \{a \mid a \in I_D(ap)\} = I_D(ap)$*
  - *if t is $ap\ pp\ dp'$, then:*
    $I_D(t) = \{(a_0 \ldots a_n) \mid a_0 \in I_D(ap),\ (a_0 \ldots a_x) \in I_D(pp),\ a_x \in I_D(head(dp'))$ *and* $(a_x \ldots a_n) \in I_D(dp')\}$

Regarding the expression $s \vDash f$ about the satisfiability of f by a set s, we have:

· for $f = f_{incl}$, written as $s[/w]$, we have that $s \vDash f$ if $w \subseteq s$
· for $f = f_{join}$, written as $s[.w]$, we have that $s \vDash f$ if $w \cap s \neq \varnothing$
· for $f = f_{disj}$, written as $s[!w]$, we have that $s \vDash f$ if $w \cap s = \varnothing$

Note here that set operators $\subseteq$ and $\cap$, obtain extra semantics with the insertion of the equivalence relation in def. 1, so that for example we can be able to say that two sets intersect, not only when they contain a common proposition, but also when they contain two equivalent ones. For this reason, we suggest the following operator overloading:

$A \subseteq B$: if $\forall x \in A, \exists y \in B$ s.t. $x \equiv y$
$A \cap B$: $\{x, .., xi, .., x_j, .., x_n\}$ such that $x_i \in A$, $x_j \in B$ and $x_i \equiv x_j$

5 Data Mapping

5.1 An Ontology about Argumentation

In this section, we describe a extended description of the AIF ontology scheme [2] [1]. Data in the Corpora of AIFdb are stored in the concepts defined by that scheme.

Description of the AIF Core Ontology

The classes of AIF are shown in figure 1 and they capture all the basic concepts of argumentation such as, arguments, premises, conclusions, relations between arguments emerging from conflicts between their parts etc.

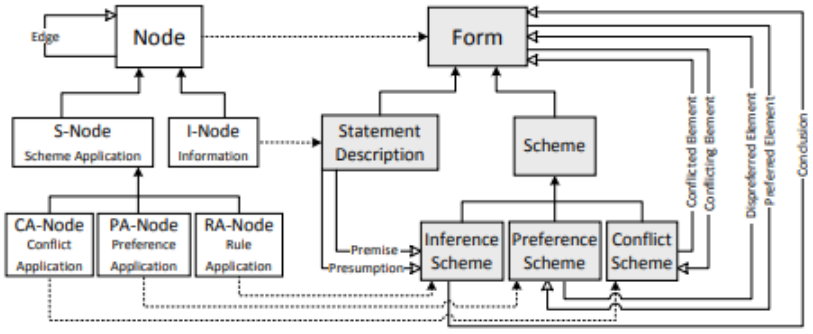


Fig. 1. AIF class taxonomy

In its current form, AIF is a two-level ontology, consisting of the Upper and the Forms ontology. The Upper ontology (white nodes) consists of the basic building blocks of argument graphs, while Forms ontology (grey

nodes), makes use of the classes in the Upper ontology and specializes them to represent contextual information. Arguments are represented as nodes in a directed graph called an *argument network*. Each node may also have a number of internal attributes that denote things like a title, creator, status etc. More specifically, the core ontology of AIF consists of two disjoint sets of nodes: *information nodes* (I-nodes) that hold the content of the argument and *scheme nodes* (S-nodes) that hold the relationships between the various parts. Scheme nodes are further specialized into three main types for representing *rule of inference application nodes* (RA-nodes), *preference application nodes* (PA-nodes) that express the concept of preference between the linked parts and *conflict application nodes* (CA-nodes), to capture the conflict between the linked parts. These nodes are further specialized in the Forms ontology according to the occasional application. For instance, inference application captured by RA-nodes, is a general concept, which can be either a deduction, a logical or defeasible inference etc. The edge "uses", maps the S-nodes of the Upper ontology, into the scheme it exploits. Notice that in Forms ontology, the edges are also typed to indicate what the role of one node is with respect to another node. For example, incoming edges of an inference scheme (RA) are typed with the label *premise*, the outcoming ones with *conlusion* and so on.

In general, the topology of AIF allows to model a variety of concepts in the area, such as: preferences and conflicts between simple information nodes or between whole arguments, by linking RA-nodes, preferences on conflicts, conflicts on preferences, meta-preferences, etc. In figure 5.1, the class taxonomy of AIF is depicted.

In this work we extend AIF by one class to capture the notion of equivalence between two information nodes. In particular, we further specialize S-Node, by adding the extra subclass *equivalence node* (MA-node) at the same level of hierarchy with RA-,CA-,PA-nodes. Like CA-nodes, MA-nodes lie between two I-nodes, except that they are used to denote that the information content of these two I-nodes is the same (although it may be expressed differently).

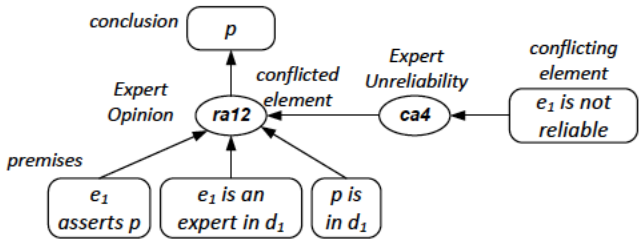


Fig. 2. Conflict from Unreliability

An example of an attack between two arguments in AIF, is shown in figure 2. It is based on Walton's argument schemes [9]. On the left side there is a typical structured argument. Node *ra12* represents the inference from the premises to conclusion *p*. The argument is an instantiation of the scheme "Argument from Expert Opinion". The node *ca4* expresses an attack to this argument by a position that undermines the expert's reliability. This is a type of asymmetric attack where an I-node attacks a CA-node. A conflict between two I-nodes is called symmetric.

According to [1], an *AIF argument graph* is defined as follows:

Definition 10 (AIF graph) *An AIF argument graph G_A is a simple digraph (V,E), where*

1. *$V = I \cup RA \cup CA \cup PA \cup MA$ is the set of nodes in G, where I are the I-nodes, RA are the RA-nodes, CA are the CA-nodes and PA are the PA-nodes*
2. *$E \subseteq V \times V \setminus I \times I$ is the set of edges in G_A. Any edge $e \in E$ is assumed to have exactly one type from among the following: premise, conclusion, preferred element, dis-preferred element, conflicting-element, conflicted-element, equal-element.*
3. *if $\upsilon \in V \setminus I$ then υ has at least one direct predecessor and one direct successor*
4. *if $\upsilon \in RA$ then υ has at least one direct predecessor via a premise edge and exactly one direct successor via a conclusion edge*
5. *if $\upsilon \in PA$ then υ has exactly one direct predecessor via a preferred edge and exactly one direct successor via a dis-preferred edge*

6. *if* $\upsilon \in CA$ *then* υ *has exactly one direct predecessor via a conflicting element and exactly one direct successor via a conflicted element edge*
7. *if* $\upsilon \in MA$ *then* υ *has exactly one direct predecessor and one direct successor via equal-element edges.*

We say that, given two nodes $\upsilon_1, \upsilon_2 \in N$, υ_1 is a predecessor of υ_2 and υ_2 is a successor of υ_1, if there is a path in G_A from υ_1 to υ_2, and υ_1 is a direct predecessor of υ_2 and υ_2 a direct successor of υ_1, if there is an edge $(\upsilon_1, \upsilon_2) \in E$. A node υ is called an initial node if it has no predecessor.

AIF-RDF: The Extended AIF Ontology in RDF Schema

In this section we describe the data in the AIFdb corpora, that use the AIF scheme. In particular, for each concept in an AIF graph(def.10), we present the set of triples that are required to describe it in RDF. Recall the notation from section 1 for the RDF representation of a concept x, written as $\|x\|$. The notation is also extended to a complete AIF graph (by writing $\|G_A\|$), to denote the set of RDF triples needed to represent it.

Each node from V is represented as an RDF resource, identifiable by a unique URI, and each edge from E creates an object property that connects the resources. We also assume as *aif* the namespace that will be used as prefix in the URIs for the classes of AIF.

- The RDFS code that defines the classes in AIF is shown below. Let *classname* be one of the I-,RA-,CA-,PA-,MA-Node:

```
<rdf:Description rdf:about=aif:classname>
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:label> classname </rdfs:label>
</rdf:Description>
```

  For the scheme nodes, the description also includes the row

```
<rdfs:subClassOf rdf:resource=aif:S-Node/>
```

- a node $i \in I$ must include a *string* data-property, say *claimText*, which will keep its information content. So assuming that the uri of i is uri_i, we have that $\|i\|$ consists of two RDF triples:

```
<NamedIndividual rdf:about=uri_i>
   <rdf:type rdf:resource=aif:I-node/>
   <aif:claimText>content</aif:claimText>
</NamedIndividual>
```

- a node $ra \in RA$ has at least one direct predecessor via a *premise* edge and exactly one direct successor via a *conclusion* edge. With *in(ra)*, we refer to the set of predecessors of *ra*, and with *out(ra)* the the successor of *ra*. Assuming that the URIs of its predecessors are $uri_1, \ldots uri_n$, its successor's uri_c and its uri uri_{ra}, we have that $\|ra\|$ is characterized by the following RDF code:

```
<NamedIndividual rdf:about=uri_ra>
    <rdf:type rdf:resource="aif:RA-node"/>
    <aif:conclusion rdf:resource=uri_c/>
</NamedIndividual>
<rdf:resource=uri_1 aif:premise rdf:resource=uri_ra/>
  ....
<rdf:resource=uri_n aif:premise rdf:resource=uri_ra/>
```

- a node $pa \in PA$ has exactly one direct predecessor via a *preferred* edge and exactly one direct successor via a *dis-preferred* edge. Assuming uri_i the predecessor's uri, uri_j the successor's and uri_{pa} the uri of *pa*, we have that $\|pa\|$ is equal to:

```
<NamedIndividual rdf:about=uri_pa>
    <rdf:type rdf:resource=aif:PA-node/>
    <aif:dis-preferred rdf:resource=uri_j/>
</NamedIndividual>
<rdf:resource=uri_i aif:preferred rdf:resource=uri_pa/>
```

- a node $ca \in CA$ has exactly one direct predecessor via a *conflicting-element* edge and exactly one direct successor via a *conflicted-element* edge. Assuming uri_i the predecessor's uri, uri_j the successor's and uri_{ca} the uri of ca, we have that $\|ca\|$ is equal to:

```
<NamedIndividual rdf:about=uri_ca>
    <rdf:type rdf:resource=aif:CA-node/>
    <aif:conflicted-element rdf:resource=uri_j/>
</NamedIndividual>
<rdf:resource=uri_i aif:conflicting-element rdf:resource=uri_ca/>
```

- Default rephrase (MA-node) is expressed in a more complex way in AIFdb corpora. In particular, the content equivalence, or otherwise a default rephrase between two I-nodes (i_1, i_2). The AIF+ ontology [8] is an extension of AIF to represent dialogic argumentation. In this work, the dialogic level is separated by the conceptual level that represents argumentative data and there is a representation mechanism to indicate the analogies between them. Default rephrase is defined in the level of dialogues and is described in figure 3

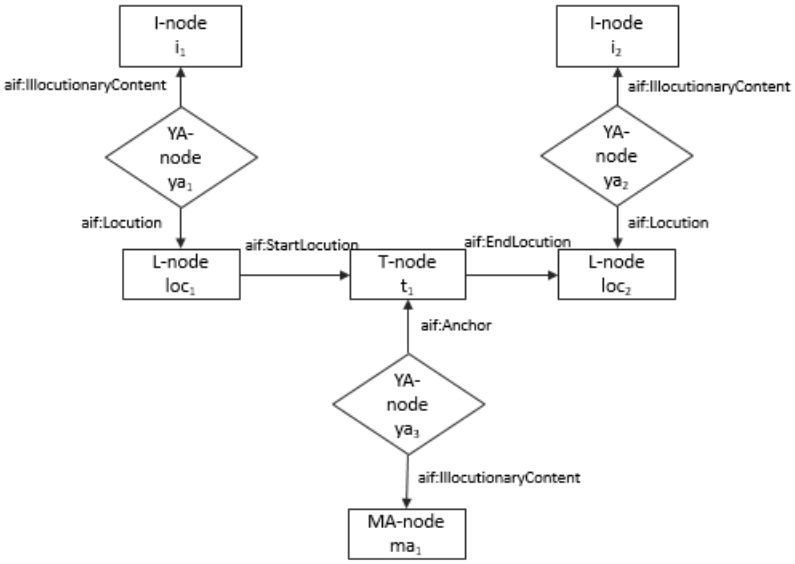


Fig. 3. Default rephrase in AIFdb

Assuming uri_{i1} , uri_{i2} the URIs of the I-nodes i_1, i_2 and uri_{ma} the uri of ma, we have that $\|ma\|$ is given by the following set of triples:

```
<NamedIndividual rdf:about=uri_ma>
    <rdf:type rdf:resource=aif:MA-node/>
</NamedIndividual>
<NamedIndividual rdf:about=uri_ya1>
    <aif:IllocutionayContent rdf:resource:uri_ma/>
    <aif:Anchor rdf:resource:uri_ta/>
```

```
</NamedIndividual>
<NamedIndividual rdf:about=uri_ta>
    <rdf:type rdf:resource=aif:TA-node/>
    <aif:StartLocution rdf:resource=uri_loc1/>
    <aif:EndLocution rdf:resource=uri_loc2/>
</NamedIndividual>
<NamedIndividual rdf:about=uri_loc1>
    <rdf:type rdf:resource=aif:L-node/>
    <aif:StartLocution rdf:resource=uri_ta/>
</NamedIndividual>
<NamedIndividual rdf:about=uri_loc2>
    <rdf:type rdf:resource=aif:L-node/>
    <aif:EndLocution rdf:resource=uri_ta/>
</NamedIndividual>
 <NamedIndividual rdf:about=uri_ya1>
    <aif:IllocutionayContent rdf:resource:uri_i1/>
    <aif:Locution rdf:resource:uri_loc1/>
</NamedIndividual>
<NamedIndividual rdf:about=uri_ya2>
    <aif:IllocutionayContent rdf:resource:uri_i2/>
    <aif:Locution rdf:resource:uri_loc2/>
</NamedIndividual>
```

5.2 From Argumentation data model to AIF

In this section, we will show the mapping between the data model defined in section 3 to the concepts of AIF and the respective RDF representation. The mapping is formalized at the level of the language $\mathcal{L} =< \mathcal{P}, \vdash >$. So we show the mappings for *a)*propositions in $\mathcal{P}$, *b)*arguments in $\mathcal{A}$, *c)* contraries between propositions and *d)* equivalences between propositions.

We adopt the following notation to formally present this translation. For any element x in the language $\mathcal{L}$, we denote by $\hat{x}$ the equivalent concept in AIF, which will be one of the *I, RA, CA, MA* types (our model does not support the type *PA*). In the continuity, we will use the $\|\hat{x}\|$, for the RDF representation of $\hat{x}$, as defined above for each different concept in AIF. Furthermore, we will need an auxiliary function $u : I \cup RA \cup CA \cup MA \rightarrow U$ which assigns URIs to the resources generated during the process. For example $u(\hat{x})$ is the URI of the concept $\hat{x}$.

Definition 9 (Mapping to AIF). *Let a dialogue D of the language $\mathcal{L} =< \mathcal{P}, \vdash >$, consisting of a set of arguments A and the corresponding equivalence ($\equiv$) and contrariness ($\not\vdash$) relations. An AIF graph produced from the dialogue D and denoted as G_D, is a graph $G_D = (V, E)$, for which $V = I \cup RA \cup CA \cup MA$, such that:*

- *for $p \in \mathcal{P}$, we have that $\hat{p} \in I$ is an I-node. The RDF representation of $\|\hat{p}\|$ is defined above and we have that $u(\hat{p})$ is the assigned URI and p is the value of the dataproperty claimText.*
- *for $a \in \mathcal{A}$, s.t. $a = \langle \{p_1, \ldots, p_n\}, c \rangle$, we have that $\hat{a}$ is a specific case of an $ra \in RA$, for which the predecessors via the premise edge are the I-nodes $\hat{p_1}, \ldots \hat{p_n}$ and the successor via the conclusion edge is the I-node $\hat{c}$. So, its RDF representation is $\|\hat{a}\| = \bigcup_{i=1}^{n} \|\hat{p_i}\| \cup \|\hat{c}\| \cup \|ra\|$, and $u(\hat{a}) = u(ra)$, $u(\hat{p_1}), \ldots, u(\hat{p_n})$ and $u(\hat{c})$, their respective URIs.*
- *for a conflict $cn = (p_1, p_2)$ with $p_1, p_2 \in \mathcal{P}$ and $p_1 \not\vdash p_2$, we have that $\hat{cn} \in CA$, for which the direct predecessor via a conflicting-element edge is the I-node $\hat{p_1}$, and its direct successor via a conflicted-element, is the I-node $\hat{p_2}$. The RDF representation $\|\hat{cn}\|$ is defined above and the respective URIs are $u(\hat{cn})$, $u(\hat{p_1})$ and $u(\hat{p_2})$. In our model, a conflict is symmetric, so $\|\hat{cn}\|$ also includes the opposite relation(p_2, p_1).*
- *for an equivalence $eq = (p_1, p_2)$ with $p_1, p_2 \in \mathcal{P}$ and $p_1 \equiv p_2$, we have that $\hat{eq} \in MA$, for which the direct predecessor and successor via the equal-element edge are the I-nodes $\hat{p_1}$ and $\hat{p_2}$.The RDF representation is $\|\hat{eq}\|$, as defined above, and the respective URIs $u(\hat{eq})$, $u(\hat{p_1}), u(\hat{p_2})$. The note about symmetry in the previous bullet, also holds here.*

The resulted AIF graph, constitutes a subset of what can be in general represented and expressed with the AIF scheme. It contains no PA-nodes, since our model does not take into account any kind of preferences and also it does not produce any of the more complex concepts like conflicts on RAs, or RAs on RAs etc.

Having formalized the mapping on the level of $\mathcal{L}$, the transition to a dialogue D as defined in 6 is trivial. Recall the quantity $P(D)$ being the logical content of D. We also assume the following notations:

- $cf(D) = \{(p_1, p_2) \mid p_1, p_2 \in P(D), s.t. p_1 \nvdash p_2\}$, is the set of conflicting pairs among the propositions in $P(D)$
- $eq(D) = \{(p_1, p_2) \mid p_1, p_2 \in P(D), s.t. p_1 \equiv p_2\}$, is the set of equivalent pairs among the propositions in $P(D)$

Definition 11 (Dialogue Mapping) *Given a dialogue D of the language $\mathcal{L}$, and A the set of arguments it consists of, the mapping of D into the AIF/RDF representation is given by the following:*

$$\|D\| = \bigcup_{a \in A} \|\hat{a}\| \cup \bigcup_{cn \in cf(D)} \|\hat{cn}\| \cup \bigcup_{eq \in eq(D)} \|\hat{eq}\|$$

In a few words, the above definition builds an AIF graph, for which the set of I-nodes, RA-nodes, CA-nodes and MA-nodes consist of all the formulas existing in the arguments in A, the arguments, the conflicts and the equivalences between the particular formulas, respectively.

5.3 ArgQL to SPARQL translation

At this point we define our ArgQL-to-SPARQL translation and afterwards we formally prove the correctness of this translation.

For the sake of the proof's clarity, we choose to show the translation in a declarative way instead of following some algorithmic approach. The whole process is based on the idea that each particular segment of ArgQL (and more specifically each different pattern) is translated to a unique graph pattern in SPARQL destined to match data in the AIF/RDF model defined in the previous section. These separate graph patterns are then combined, to form the final query. To this end, we adopt some notations that will help us show this translation.

First of all, assuming a pattern x in ArgQL from the set M, we denote with $\langle\langle x \rangle\rangle$ the SPARQL graph pattern, to which it corresponds. Regarding the variables, the idea is that we reuse the already existing variables from the ArgQL query and place them in particular positions in the graph pattern, in order to get the correct values. However, we will also need new variables to bind with RDF data that do not exist as values in our argumentation model, but are necessary for identifying the right information. Such data are basically RDF resource identifiers (URIs). We adopt a notation for the new created variables, which is defined as $\_v_x \in V$. Indicator x is one of the *(i, ra, ca, ma, pa)* and points to the AIF class type, which the particular URI refers to. With the symbol $\wedge$, we denote the conjunction between individual graph patterns.

Furthermore, there may be cases where, a variable of a graph pattern will need to be accessed by another one, in order to express a join between them, on this variable. We use the symbolism $\langle\langle x \rangle\rangle.v$, to access the variable v of the graph pattern $\langle\langle x \rangle\rangle$.

Having agreed on the naming conventions, we now move on presenting the translation. Let Q_A the infinite set of ArgQL queries and a query $q \in Q_A$, which according to the definition 7 has the form:

```
q ← match dialoguePattern1 , ..., dialoguePatternn
 return varlist
```

The SPARQL equivalent of q is denoted as $q* \in Q_S$, where Q_S is the infinite set of SPARQL queries, and is defined as:

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

The following list shows the translation for each particular pattern (or motif) and for all the different forms they may have.

- for a constant *proposition* pattern $p \in \mathcal{P}$, recall that the matching data is the same proposition p as well as any equivalent proposition in the knowledge base. Thus, the corresponding SPARQL graph pattern must also detect equivalent I-nodes. Assuming that p is the referred proposition. With $\sim p$ we indicate the random equivalent proposition of p. So we have:

$$\langle\!\langle p \rangle\!\rangle = (\_v_i \text{ rdf:type aif:I-node}) \wedge (\_v_i \text{ aif:claimText p}) \text{ and}$$

$$\begin{aligned}\langle\!\langle \sim p \rangle\!\rangle = \; & \_vya1 \text{ aif:IllocutionaryContent} \wedge \langle\!\langle p \rangle\!\rangle.\_v_i \\ & (\_vya1 \text{ rdf:type aif:YA-node}) \wedge \\ & (\_vya1 \text{ aif:Locution } \_v_{loc1}) \wedge \\ & (\_v_{loc1} \text{ rdf:type aif:L-node}) \wedge \\ & (\_v_i \text{ rdf:type aif:I-node}) \wedge \\ & (\_v_i \text{ aif:claimText } \_v_{i1}) \wedge \\ & (\_vya2 \text{ rdf:type aif:YA-node}) \wedge \\ & (\_vya2 \text{ aif:IllocutionaryContent } \_v_i) \wedge \\ & (\_vya2 \text{ aif:Locution } \_v_{loc2}) \wedge \\ & (\_v_{loc2} \text{ rdf:type aif:L-node}) \wedge \\ & (\_v_{ta} \text{ rdf:type aif:TA-node}) \wedge \\ & (\_v_{loc1} \text{ aif:StartLocution } \_v_{ta}) \wedge \\ & (\_v_{ta} \text{ aif:EndLocution } \_v_{loc2}) \wedge \\ & (\_vya3 \text{ rdf:type aif:YA-node}) \wedge \\ & (\_vya3 \text{ aif:Anchor } \_v_{ta}) \wedge \\ & (\_vya3 \text{ aif:IllocutionaryContent } \_v_{ma}) \wedge \\ & (\_vma \text{ rdf:type aif:MA-node})\end{aligned}$$

The graph pattern used to find the equivalent I-nodes, must be that complex to match with the set of triples required to represent an MA-node, as explained before.

– a *proposition_set* pattern is not translated separately, since, depending on where it is used in the query, it is translated differently.

– for a *premise_pattern pr*, of an argument pattern ap, we have:
  - if pr is a *proposition_set* pattern $ps = \{p_1, \ldots, p_n\}$, then:

$$\begin{aligned}\langle\!\langle pr \rangle\!\rangle = \bigwedge_{i=1}^{n} \Big(& \{\langle\!\langle p_i \rangle\!\rangle \wedge (\langle\!\langle p_i \rangle\!\rangle.\_v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle.\_v_{ra})\} \text{ UNION} \\ & \{\langle\!\langle \sim p_i \rangle\!\rangle \wedge (\langle\!\langle \sim p_i \rangle\!\rangle.\_v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle.\_v_{ra})\}\Big)\end{aligned}$$

    Intuitively, for the case of a constant proposition set as a premise pattern, it holds that for all propositions $\{p_1, .., p_n\}$, it must hold that either these, or their equivalent ones ($\sim p_i$), must be premises of the same argument. In case that there exists at least one p_i, for which, neither it, nor its equivalent belongs to the premises, the matching of the pattern fails.

  - if pr is of the form $v[f]$, where $v \in V$ and f a filter (which is optional), then:

$$\langle\!\langle pr \rangle\!\rangle = (\_v_i \text{ rdf:type aif:I-node}) \wedge (\_v_i \text{ aif:claimText } v) \wedge (\_v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle.\_v_{ra}) \wedge \langle\!\langle f \rangle\!\rangle$$

    In case that the premise pattern is a variable followed by a filter, they both must be translated. If the pattern contains no filter, the part $\langle\!\langle f \rangle\!\rangle$ is omitted. Variable v is put in the particular place, in order to get the value of the content of the I-node, which is essentially the value of the matching proposition in ArgQL.

– for f a *premise filter*, coming from the premise_pattern $pr = v[f]$, with $v \in V$ which, according to its definition has the form:

$$premise\_filter \leftarrow (\text{'/'} \mid \text{'.'} \mid \text{'!'})(proposition\_set \mid variable)$$

Each of the three different filters is based either on a constant proposition_set pattern (say $ps' = \{s_1, \ldots, s_m\}$), or on a variable (say v_p) with a set of propositions as a value. Since the query has been verified, variable v_p will refer to the premise pattern pr' of a different argument pattern, say ap'. We will write $\langle\!\langle pr' \rangle\!\rangle.\_v_i$, in order to refer to the variable that will match with the URIs of the premise in pr' and with $\langle\!\langle ap' \rangle\!\rangle.\_v_{ra}$ to the variable for the respective RA-node of the matching arguments of ap'. Totally these cases create 6 different cases, each one generating a different graph pattern. Next, we show the translation for these cases:

  - for $f_{incl} = [/ps']$ we have:

$$\langle\!\langle f_{incl} \rangle\!\rangle = \bigwedge_{i=1}^{m} \Big(\{ \langle\!\langle s_i \rangle\!\rangle \wedge (\langle\!\langle s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \text{ UNION } \{ \langle\!\langle \sim s_i \rangle\!\rangle \wedge (\langle\!\langle \sim s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \Big)$$

When the filter requires the inclusion of all propositions of ps' in the premise part of an argument, again it is also checked whether any of their equivalents $\sim s_i$ are also included.

- for $f_{incl} = [/v_p]$. This case of filter is a little peculiar because it requires that the candidate matching argument must contain *all* of the propositions in v_p in its premise set and not just some of them. SPARQL does not provide an explicit way to express the *"for all"* case. It allows though to write it as the double negation "there is no premise in argument ap' which is not a premise of the current one ap". We use the *NOT EXISTS* filter twice, as follows:

$$\langle\!\langle f_{incl} \rangle\!\rangle = \text{FILTER NOT EXISTS } \{ \; (\_v_i \text{ aif:Premise } \langle\!\langle ap' \rangle\!\rangle ._{-}v_{ra}) \wedge \text{FILTER NOT EXISTS } \{ \; (\_v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \; \} \; \}$$

- for $f_{join} = [.ps']$, which means the requirement for pr to intersect with the constant set $ps' = \{s_1, \ldots, s_m\}$. So:

$$\langle\!\langle f_{join} \rangle\!\rangle = \bigcup_{i=1}^{m} \Big(\{ \langle\!\langle s_i \rangle\!\rangle \wedge (\langle\!\langle s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \text{ UNION } \{ \langle\!\langle \sim s_i \rangle\!\rangle \wedge (\langle\!\langle \sim s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \Big)$$

Here, the $\bigcup$ symbol represents the UNION keyword in SPARQL which expresses that at least one of the s_i, or its equivalent one, suffice to be included in the premise part.

- for $f_{join} = [.v_p]$, which is translated to the requirement that pr must join with the premise variable v_p of the premise pattern of another variable $ap' = \langle pr', cp' \rangle$, with pr' and cp' its premise pattern and conclusion pattern, respectively. We have:

$$\langle\!\langle f_{join} \rangle\!\rangle = \big(\langle\!\langle pr' \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra} \big)$$

- for $f_{disj} = [!ps']$, which means the requirement of pr having no common elements with ps', we have:

$$\langle\!\langle f_{disj} \rangle\!\rangle = \text{FILTER NOT EXISTS } \{ \bigcup_{i=1}^{m} \Big(\{ \langle\!\langle s_i \rangle\!\rangle \wedge (\langle\!\langle s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \text{ UNION } \{ \langle\!\langle \sim s_i \rangle\!\rangle \wedge (\langle\!\langle \sim s_i \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra}) \} \Big) \; \}$$

That filter implements the *de Morgan* rule, according to which: $\neg(a \vee b) \equiv (\neg a \wedge \neg b)$. In particular, in order to express that *none of the p_i propositions is included in the current premise part*, written as a big conjunction where each proposition is *not* a premise for the current argument, we write that *it does not hold that at least one of the p_i is a premise*, expressed as the negation of the corresponding disjunction. Again here, the condition is written also for the equivalent propositions.

- for $f_{disj} = [!v_p]$, which means the requirement that the two premise parts pr and pr' from the argument patterns ap and ap' are disjoint.

$$\langle\!\langle f_{disj} \rangle\!\rangle = \text{FILTER NOT EXISTS } \{ \big(\langle\!\langle pr' \rangle\!\rangle ._{-}v_i \text{ aif:Premise } \langle\!\langle ap \rangle\!\rangle ._{-}v_{ra} \big) \}$$

– for a conclusion pattern cp of an argument pattern ap, we have:

- if cp is a constant *proposition* pattern $c \in \mathcal{P}$, then:

$$\langle\!\langle cp \rangle\!\rangle = \big\{ \langle\!\langle c \rangle\!\rangle \wedge (\langle\!\langle ap \rangle\!\rangle ._{-}v_{ra} \text{ aif:Conclusion } \langle\!\langle c \rangle\!\rangle ._{-}v_i) \big\} \text{ UNION } \big\{ \langle\!\langle \sim c \rangle\!\rangle \wedge (\langle\!\langle ap \rangle\!\rangle ._{-}v_{ra} \text{ aif:Conclusion } \langle\!\langle \sim c \rangle\!\rangle ._{-}v_i) \big\}$$

In order to access the variables $\langle\!\langle c \rangle\!\rangle.\_v_i$ and $\langle\!\langle \sim c \rangle\!\rangle.\_v_i$ of cp from an outer pattern, we will write respectively $\langle\!\langle cp \rangle\!\rangle.\_v_i$ and $\langle\!\langle c \rangle\!\rangle.\_v'_i$, respectively.

- if cp is a variable $v_c \in V$, then:

$$\langle\!\langle cp \rangle\!\rangle = (\_v_i \text{ rdf:type aif:I-node }) \wedge (\_v_i \text{ aif:claimText } v_c) \wedge (\langle\!\langle ap \rangle\!\rangle.\_v_{ra} \text{ aif:Conclusion } \_v_i)$$

– for an argument pattern $ap =< pr, cp >$, with pr a premise pattern and cp a conclusion pattern, we have:

$$\langle\!\langle ap \rangle\!\rangle = \langle\!\langle pr \rangle\!\rangle \wedge \langle\!\langle cp \rangle\!\rangle \wedge (\_v_{ra} \text{ rdf:type aif:RA-node })$$

We don't examine separately the case where the argument pattern is a single variable, since this case generates the same translation with an argument pattern, for which, the premise part is a single variable with no filters and the conclusion part is also a single variable.

– Regarding a relation pattern r, we assume it appears between two argument patterns $ap_1 =< pr_1, cp_1 >$ and $ap_2 =< pr_2, cp_2 >$. For each of the 6 different relation types we have:

- if r = *rebut* then, for each of the conclusion patterns cp_1, cp_2, we have to check both for the main I-node variable, and for the variable that points to its equivalent one, whether there is a conflict. This results to 4 different cases as follows:

$$\begin{aligned}\langle\!\langle \text{rebut} \rangle\!\rangle = &(\langle\!\langle cp_1 \rangle\!\rangle.\_v_i \text{ aif:conflicting-element/aif:conflicted-element } \langle\!\langle cp_2 \rangle\!\rangle.\_v_i) \text{ UNION} \\ &(\langle\!\langle cp_1 \rangle\!\rangle.\_v'_i \text{ aif:conflicting-element/aif:conflicted-element } \langle\!\langle cp_2 \rangle\!\rangle.\_v_i) \text{ UNION} \\ &(\langle\!\langle cp_1 \rangle\!\rangle.\_v_i \text{ aif:conflicting-element/aif:conflicted-element } \langle\!\langle cp_2 \rangle\!\rangle.\_v'_i) \text{ UNION} \\ &(\langle\!\langle cp_1 \rangle\!\rangle.\_v'_i \text{ aif:conflicting-element/aif:conflicted-element } \langle\!\langle cp_2 \rangle\!\rangle.\_v'_i)\end{aligned}$$

- if r = *undercut*, again here, we have to check for all the combinations between conclusion pattern cp_1 and the premise pattern pr_2, and check for conflicts between the respective main variables and their equivalent ones. For simplicity we will write only the first case.

$$\langle\!\langle \text{undercut} \rangle\!\rangle = \left(\langle\!\langle cp_1 \rangle\!\rangle.\_v_i \text{ aif:conflicting-element/aif:conflicted-element } \langle\!\langle pr_2 \rangle\!\rangle.\_v_i\right)$$

- if r = *attack*:

$$\langle\!\langle \text{attack} \rangle\!\rangle = \langle\!\langle \text{rebut} \rangle\!\rangle \text{ UNION } \langle\!\langle \text{undercut} \rangle\!\rangle$$

- for the cases where r is one of the *endorse, back, support*, we will need the graph pattern described in the case of translation of a constant *proposition* pattern, used to detect equivalent I-nodes. This increases a lot the complexity to express the respective graph patterns. In order not to weary the reader with even complex notations we will give a brief description for each case. For the *endorse*, the two I-node variables that need to be checked for equivalence, are from the conclusion patterns cp_1, cp_2. For the *back*, these are the variables from the cp_1 and the pr_2. The *support* is the union of these two.

– for a *path_pattern* pp, assuming that it also appears between two argument patterns $ap_1 =< pr_1, cp_1 >$ and $ap_2 =< pr_1, cp_1 >$, we have

- if pp is a relation pattern r, then:

$$\langle\!\langle pp \rangle\!\rangle = \langle\!\langle r \rangle\!\rangle$$

- if pp is of the form pp_1/pp_2, where pp_1, pp_2 path_patterns, assuming that it can be written equally, as ap_1 pp_1 ap_2 pp_2 ap_3, we have:

$$\langle\!\langle pp_1/pp_2 \rangle\!\rangle = \langle\!\langle pp_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle pp_2 \rangle\!\rangle$$

- if pp is of the form $pp * n$, which is written alternatively as $pp/\ldots/pp$ (n repetitions), we have:
  - if $n = 1$, then $\langle\!\langle pp * n \rangle\!\rangle = \langle\!\langle pp \rangle\!\rangle$

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

- if pp is of the form $pp+n$, which means, "at most n repetitions of pp", we have:

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

– for a dialogue pattern dp, we have:
- if $dp = ap$, where ap an argument pattern, then: $\langle\langle dp \rangle\rangle = \langle\langle ap \rangle\rangle$
- if $dp = ap\ pp\ dp'$, then: $\langle\langle dp \rangle\rangle = \langle\langle ap \rangle\rangle \wedge \langle\langle pp \rangle\rangle \wedge \langle\langle dp' \rangle\rangle$

Information and Semantics preservation

In this section we will show that the suggested translation is semantics preserving, which essentially means that given the data mapping of section 9, the SPARQL query generated by the translation of ArgQL query will give the correct (expected) results.

Here we will need some notations described in the section 5.1 in the description of the RDF representation of AIF. In particular, with *in(x)* where x an S-node, we refer to the set of the predecessor nodes of x, and with *out(ra)* to the the successor nodes of x.

We state the following lemma, which is necessary for the general proof.

Lemma 1. *A matching instance of the translation of an argument pattern $\langle\langle ap \rangle\rangle$ in an AIF-graph G_A is a ra node, for which in(ra) is a set of I-nodes and out(ra) is also an I-node.*

Proof. The lemma is trivially proved. Roughly, for the general case of the premise pattern ($v[f]$, where $v \in V$, f a filter and $\_v \in V$ the temporarily created variable for v), a mapping instance of the translated graph pattern consists of the triples

$\{(\mu(\_v)$ *rdf:type I-node),* $(\mu(\_v)$ *aif:claimText* $\mu(v))$, $(\mu(\_v)$ *aif:Premise* $u_{ra})\}$

$in(ra)$ is created by the I-nodes, the uris of which are included in the set of mappings of $\mu(\_v)$ for the given u_{ra}, and which precede the *ra* via the *premise* edge.

In the case where the premise pattern is a constant set of propositions $\{p_1, \ldots p_n\}$, with $p_i \in \mathcal{P}$, a mapping instance of the translated graph pattern consists of the triples

$\{(\mu(\_p_1)$ *rdf:type I-node),* $(\mu(\_p_1)$ *aif:claimText* $p_1)$, $(\mu(\_p_1)$ *aif:Premise* $u_{ra})$

$\ldots$

$(\mu(\_p_n)$ *rdf:type I-node),* $(\mu(\_p_n)$ *aif:claimText* $p_n)$, $(\mu(\_p_n)$ *aif:Premise* $u_{ra})\}$

In this case, $in(ra)$ consists of the I-nodes to which the propositions p_i will map, and $\mu(\_p_i)$ will bind to their uris.

Respectively, for the translated conclusion pattern (let $c \in \mathcal{P} \cup V$ and $\_c \in V$ the temporarily created variable) a mapping instance will include the following triples:

$\{(\mu(\_c)$ *rdf:type I-node),* $(\mu(\_c)$ *aif:claimText c),*$(u_{ra}$ *aif:Conclusion* $\mu(\_c))\}$

The mapping for of $\mu(\_c)$ for each different *ra* includes a unique value, which is the uri of the successor I-node via the *conclusion* edge. □

The main theorem with which we claim the correctness of our proposed translation methodology is the following:

Theorem 1 *Given a dialogue pattern dp of ArgQL, a dialogue D, the mappings $\|\cdot\|$ and $\langle\langle\cdot\rangle\rangle$, for the data model and ArgQL patterns respectively, $I_D(\cdot)$ the evaluation function of the ArgQL patterns in D and the evaluation function $\mathcal{E}(\cdot,\cdot)$ of SPARQL graph patterns in an RDF graph, the ArgQL-to-SPARQL translation is semantics preserving, which means that $\|I_D(dp)\| = \mathcal{E}(\langle\langle dp \rangle\rangle, \|D\|)$*

Proof. In order to prove the equivalence $\|I_D(dp)\| = \mathcal{E}(\langle\langle dp \rangle\rangle, \|D\|)$, we will show that $\|I_D(dp)\| \subseteq \mathcal{E}(\langle\langle dp \rangle\rangle, \|D\|)$ and $\mathcal{E}(\langle\langle dp \rangle\rangle, \|D\|) \subseteq \|I_D(dp)\|$. Assume a dialogue D under the language $\mathcal{L}$ consisting of a set of arguments A, as well as the equivalence and contrariness relations among the propositions in $P(D)$.

According to the syntax a *dialogue pattern* dp is defined as:

$$dp = ap \mid ap\ pp\ dp'$$

where ap is an argument pattern, pp is a path pattern and dp' another dialogue pattern.

First we prove the relation $\|I_D(dp)\| \subseteq \mathcal{E}(\langle\!\langle dp \rangle\!\rangle, \|D\|)$ for both forms of a dp:

– For the first form, where dp is a single argument pattern ap, we will examine two cases:
a) $ap = \langle \{s_1, \ldots, s_m\}, cp \rangle$, with $s_i \in \mathcal{P}$ and $cp \in \mathcal{P} \cup V$
b) $ap = \langle pr[f], cp \rangle$, with $pr \in V$, $cp \in \mathcal{P} \cup V$ and f a filter
(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.)
a) Let an argument $a \in A$, s.t. $a \in I_D(ap)$ and $a = \langle \{s_1, \ldots, s_m\}, c \rangle$. This means that $\{s_1, \ldots, s_m\} \in I_D(\{s_1, \ldots, s_m\})$ and $c \in I_D(cp)$. By extension, $\|\hat{a}\| \in \|I_D(ap)\|$.
By the definition 9, it is known that

$$\|\hat{a}\| = \bigcup_{i=1}^{m} \|\hat{s_i}\| \cup \|\hat{c}\| \cup \|ra\|$$

At the same time, the translation of ap, according to the section 5.3 gives the following graph pattern :

$\langle\!\langle ap \rangle\!\rangle =$

?_s_1	rdf:type	aif:I-Node.
?_s_1	aif:claimText	s_1
	…	
?_s_m	rdf:type	aif:I-Node.
?_s_m	aif:claimText	s_m
?_c	rdf:type	aif:I-Node.
?_c	aif:claimText	cp.
?_ra	rdf:type	aif:RA-node.
?_s_1	aif:Premise	?_ra.
	…	
?_s_m	aif:Premise	?_ra.
?_ra	aif:Conclusion	?_c

Using the lemma 1, we have that a matching instance of $\langle\!\langle ap \rangle\!\rangle$ is a particular ra, for which $in(ra)$ includes I-nodes with values $s_1, \ldots s_m$ and $out(ra)$ is the I-node that matches $\mu(\_c)$. By the mapping in def 9 it is obvious that these I-nodes are the $\hat{s_1}, \ldots \hat{s_m}$ and $\hat{c}$, that come from the $prem(a)$ and $concl(a)$, respectively. So we can safely conclude, that this particular ra is the same with that included in the $\|\hat{a}\|$ and therefore $\|\hat{a}\| \in \mathcal{E}(\langle\!\langle ap \rangle\!\rangle, \|D\|)$

b) For the case when $ap = \langle v[f], cp \rangle$, for which, the premise pattern $pr : v[f]$ and the conclusion pattern cp the proof goes as follows.
Let an argument $a \in A$, s.t. $a \in I_D(ap)$ and $a = \langle \{p_1, \ldots, p_n\}, c \rangle$, with $\{p_1, \ldots p_n\} \in I_D(pr)$ and $c \in I_D(cp)$.
The translation of ap, according to the section 5.3 gives the following graph pattern :

$\langle\!\langle ap \rangle\!\rangle =$

?_v	rdf:type	aif:I-Node.
?_v	aif:claimText	v
?_v	aif:Premise	?_ra.
?_c	rdf:type	aif:I-Node.
?_c	aif:claimText	cp.
?_ra	aif:Conclusion	?_c
?_ra	rdf:type	aif:RA-node.
$\langle\!\langle f \rangle\!\rangle$		

Again using the lemma 1, we have that a matching instance of $\langle\!\langle ap \rangle\!\rangle$ is a particular ra, for which $in(ra)$ includes the set of I-nodes specified by the set of mapping of $\mu(\_v)$ and $out(ra)$ is the I-node that matches $\mu(\_c)$. Here, it is necessary to show that the translation of the filter $\langle\!\langle f \rangle\!\rangle$ resricts correctly the matching *ras*. For the case that the filter is based on a variable v_2, it will refer to the premise part of a different argument pattern, let ap_2. For each of the eight different filter cases, the proof goes as follows:

- $f_{incl} = [/\{s_1,..,s_k\}]$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{incl}$. This means that $\{s_1,..,s_k\} \subseteq \{p_1,..p_n\}$. Without loss of generaliyy, we assume that $prem(a) = \{s_1,..,s_k,..,p_n\}$.
  By definition 9, we have that
  $$\|\hat{a}\| = \|\hat{s_1}\| \cup .. \cup \|\hat{s_k}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$
  On the other hand, $\langle\langle f_{incl} \rangle\rangle$ requires for the matching ra', the set $in(ra')$ to include I-nodes with values $s_1,..s_k$. It is obvious that the ra of $\|\hat{a}\|$ satisfies that requirement, since these I-nodes are the $\hat{s_1},..,\hat{s_k}$, so $ra = ra'$ and $\|\hat{a}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$.

- $f_{incl} = [/v_2]$, with $v_2 \in V$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{incl}$. This means that ap_2 has been evaluated and $\exists a' \in I_D(ap_2)$, s.t. $\mu(v_2) = prem(a')$ and $prem(a') \subseteq prem(a)$. In other words, if $a' = \langle\{p'_1,..,p'_m\}, c'\rangle$, then $a = \langle\{p'_1,..,p'_m,..,p_n\}, c\rangle$
  By definition 9, we have that
  $$\|\hat{a'}\| = \|\hat{p'_1}\| \cup .. \cup \|\hat{p'_m}\| \cup \|\hat{c'}\| \cup \|ra_1\|$$
  $$\|\hat{a}\| = \|\hat{p'_1}\| \cup .. \cup \|\hat{p'_m}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$
  On the other hand, $\langle\langle f_{incl} \rangle\rangle$ requires for the matching ra', that there is another ra'', such that $\forall p : p \in in(ra'')$, it also holds that $p \in in(ra')$. That condition is satisfied by the ra and ra_1 of the $\|\hat{a}\|$ and $\|\hat{a'}\|$, respectively, since $in(ra_1) \subseteq in(ra)$. So $ra' = ra$ and $ra'' = ra_1$, and $\|\hat{a}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$

- $f_{join} = [.\{s_1,..,s_k\}]$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{join}$. This means that $\exists s, s \in \{s_1,..,s_k\}$, s.t. $s \in prem(a)$, or in other words $prem(a) = \{p_1,..,s,..,p_n\}$.
  By definition 9, we have that
  $$\|\hat{a}\| = \|\hat{p_1}\| \cup .. \cup \|\hat{s}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$
  On the other hand, $\langle\langle f_{join} \rangle\rangle$ requires for the matching ra', that the set $in(ra)$ will include at least one I-node, which has value one of the $s_1,..,s_k$ (expressed with the UNION operator). It is obvious that the ra of $\|\hat{a}\|$ satisfies that requirement, since the I-node $\hat{s} \in in(ra)$, so $ra = ra'$ and $\|\hat{a}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$.

- $f_{join} = [.v_2]$, with $v_2 \in V$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{join}$. This means that ap_2 has been evaluated and $\exists a' \in I_D(ap_2)$, s.t. $\mu(v_2) = prem(a')$ and $prem(a') \cap prem(a) \neq \varnothing$. In other words, if $a' = \langle\{p'_1,..s,..,p'_m\}, c'\rangle$, then $a = \langle\{p_1,..,s..,p_n\}, c\rangle$.
  By definition 9, we have that
  $$\|\hat{a'}\| = \|\hat{p'_1}\| \cup .. \cup \|\hat{s}\| \cup .. \cup \|\hat{p'_m}\| \cup \|\hat{c'}\| \cup \|ra_1\|$$
  $$\|\hat{a}\| = \|\hat{p_1}\| \cup .. \cup \|\hat{s}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$
  On the other hand, $\langle\langle f_{join} \rangle\rangle$ requires for the matching ra', that there is another ra'', with which there is a join between $in(ra')$ and $in(ra'')$. That condition is satisfied by the ra and ra_1 of the $\|\hat{a}\|$ and $\|\hat{a'}\|$, respectively, since $\exists x : x \in in(ra_1)$ and $x \in in(ra)$. That x is the $\hat{s}$. So $ra' = ra$ and $ra'' = ra_1$, and $\|\hat{a}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$

- $f_{disj} = [!\{s_1,..,s_k\}]$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{join}$. This is translated into $\nexists s : s \in \{s_1,..,s_k\}$ and $s \in prem(a)$.

  By definition 9, we have that
  $$\|\hat{a}\| = \|\hat{p_1}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$
  On the other hand, $\langle\langle f_{disj} \rangle\rangle$ requires for the matching ra', that the set $in(ra')$ will not include any I-node, which has value one of the $s_1,..,s_k$. It is obvious that the ra of $\|\hat{a}\|$ satisfies that requirement, since $in(ra)$ includes only the I-nodes $\hat{p_1},..,\hat{p_n}$ and none of them has value one of the $s_1,..,s_k$. So $ra = ra'$ and $\|\hat{a}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$.

- $f_{disj} = [!v_2]$, with $v_2 \in V$. Since $a \in I_D(ap)$, it holds that $a \vDash f_{incl}$. This means that ap_2 has been evaluated and $\exists a' \in I_D(ap_2)$, s.t. $\mu(v_2) = prem(a')$ and $prem(a') \cap prem(a) = \varnothing$. Let $a' = \langle \{p'_1, .., p'_m\}, c' \rangle$, then $a = \langle \{p_1, .., p_n\}, c \rangle$.
  By definition 9, we have that

  $$\|\hat{a'}\| = \|\hat{p'_1}\| \cup .. \cup \|\hat{p'_m}\| \cup \|\hat{c'}\| \cup \|ra_1\|$$

  $$\|\hat{a}\| = \|\hat{p_1}\| \cup .. \cup \|\hat{p_n}\| \cup \|\hat{c}\| \cup \|ra\|$$

  On the other hand, $\langle\!\langle f_{disj} \rangle\!\rangle$ requires for the matching ra', that there is another ra'', with which there is no join between $in(ra')$ and $in(ra'')$. That condition is satisfied by the ra and ra_1 of the $\|\hat{a}\|$ and $\|\hat{a'}\|$, respectively, since $\nexists\, x : x \in in(ra_1)$ and $x \in in(ra)$. So $ra' = ra$ and $ra'' = ra_1$, and $\|\hat{a}\| \in \mathcal{E}(\langle\!\langle ap \rangle\!\rangle, \|D\|)$

Generalizing we have proved that $\forall a \in I_D(ap)$, it happens that $\|\hat{a}\| \in \mathcal{E}(\langle\!\langle ap \rangle\!\rangle, \|D\|)$.
Consequently $\|I_D(ap)\| \subseteq \mathcal{E}(\langle\!\langle ap \rangle\!\rangle, \|D\|)$

– Now, we examine the second form, in which $dp : ap\ pp\ dp'$, with pp a path pattern and dp' another dialogue pattern.
A matching instance of a dialogue pattern, is a sequence of arguments that satisfy the sequence of relations that appear in the path patterns of dp or dp'.

Let $(a_1 ... a_n) \in I_D(dp)$. In other words, and according to the semantics, this means that:

$a_1 \in I_D(ap)$, $(a_1 .. a_x) \in I_D(pp)$ for some a_x in the sequence, $a_x \in I_D(head(dp'))$, where head(dp') the first argument pattern appearing in dp' and finally, that $(a_x .. a_n) \in I_D(dp')$.

This sequence of arguments, is being mapped to the following sequence of ra nodes, according to the definition 9 ($\|\hat{a_1}\| ... \|\hat{a_n}\|$). We have to show that this sequence of ra nodes, satisfies the translation of the dialogue pattern which is:

$$\langle\!\langle dp \rangle\!\rangle = \langle\!\langle ap \rangle\!\rangle \wedge \langle\!\langle pp \rangle\!\rangle \wedge \langle\!\langle dp' \rangle\!\rangle$$

We have already shown that since $a_1 \in I_D(ap)$, it also holds that $\|\hat{a_1}\| \in \mathcal{E}(\langle\!\langle ap \rangle\!\rangle, \|D\|)$. Respectively, given that $a_x \in I_D(head(dp'))$, we also know that $\|\hat{a_x}\| \in \mathcal{E}(\langle\!\langle head(dp') \rangle\!\rangle, \|D\|)$. As a consequence, what remains to be proved is that $(\|\hat{a_1}\| ... \|\hat{a_x}\|) \in \mathcal{E}(\langle\!\langle pp \rangle\!\rangle, \|D\|)$
We will present the proof in a recursive way, to capture all the possible forms a path pattern may have.

- if pp is a single relation then $(a_1 .. a_x) \in I_D(pp)$ means that the sequence consists of two arguments $(a_1 a_2) \in I_D(pp)$. We will focus on the *rebut* relation type. The rest cases are trivial and follow the same methodology. The relation a_1 *rebuts* a_2 implies that $concl(a_1) \not\dashv concl(a_2)$. Assuming $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$, it means that there is the conflict $cn = (c_1, c_2)$. Invoking the definition 9 for one more time, we have that $\hat{cn}$ is a *CA-node* for which the predecessor via the *conflicting-element* is the I-node $\hat{c_1}$ and the successor via the *conflicted-element* is the I-node $\hat{c_2}$, so the RDF representation $\|\hat{cn}\|$ will connect the uri $u(\hat{cn})$ with the $u(\hat{c_1})$ and $u(\hat{c_2})$ with the respective properties.
  On the other hand the translation of the particular pp is

  $$\langle\!\langle rebut \rangle\!\rangle = \left(\langle\!\langle cp_1 \rangle\!\rangle ._{v_i}\ \textit{aif:conflicting-element/aif:conflicted-element}\ \langle\!\langle cp_2 \rangle\!\rangle ._{v_i}\right)$$

  where cp_1 is the conclusion pattern of ap, and cp_2 the conclusion pattern of $head(dp')$.

  It is obvious that $\mu(\langle\!\langle cp_1 \rangle\!\rangle ._{v_i}) = u(\hat{c_1})$ and $\mu(\langle\!\langle cp_2 \rangle\!\rangle ._{v_i}) = u(\hat{c_2})$. So we conclude that $(\|\hat{a_1}\| \|\hat{a_1}\|)$ satisfies the $\mathcal{E}(\langle\!\langle rebut \rangle\!\rangle, \|D\|)$

- if pp is pp_1/pp_2 then $(a_1 .. a_x) \in I_D(pp)$ means that for some a_i within the sequence, it holds that $(a_1 .. a_i) \in I_D(pp_1)$, $(a_i .. a_x) \in I_D(pp_2)$ and $a_i \in I_D(ap_i)$, for some random argument pattern ap_i, between pp_1, pp_2. According to the definition 9, that sequence is mapped to the following :

$(\|a_1\|..\|ai\|..\|a_x\|) \in \|I_D(pp)\|$

At the same time, the translation of this pp is :

$\langle\langle pp_1/pp_2\rangle\rangle = \langle\langle pp_1\rangle\rangle \wedge \langle\langle ap_i\rangle\rangle \wedge \langle\langle pp_2\rangle\rangle$

We have already proved that $\|a_i\| \in \mathcal{E}(\langle\langle ap_i\rangle\rangle, \|D\|)$. We have to prove that $(\|a_1\|..\|ai\|) \in \mathcal{E}(\langle\langle pp_1\rangle\rangle, \|D\|)$ and that $(\|a_i\|..\|ax\|) \in \mathcal{E}(\langle\langle pp_2\rangle\rangle, \|D\|)$

The proof progresses recursively, until both of them conclude to be a single relation. Then the procedure was described in the previous bullet.

- if pp is of the form $pp' * n$, with $n \geq 1$, which is translated into $pp'/../pp'$ (n repetitions), then $(a_1..a_x) \in I_D(pp)$ means $(a_1..a_x) \in I_D(pp'/pp' * (n-1))$. That case is reduced in the previous one, with $pp_1 = pp'$ and $pp_2 = pp' * (n-1)$, so the proof follows the same procedure.
  So it holds that $(\|a_i\|..\|ax\|) \in \mathcal{E}(\langle\langle pp' * n\rangle\rangle, \|D\|)$

- if pp is of the form $pp' + n$, with $n \geq 1$. Let $(a_1..a_x) \in I_D(pp)$. This means that $(a_1..a_x) \in \bigcup_{k=1}^{n} I_D(pp * k)$, or alternatively:
  $(a_1..a_x) \in I_D(pp * i)$, for some $1 \leq i \leq n$.

  Bu the mapping in definition 9, we know that
  $(\|a_1\|..\|a_x\|) \in \left(\|I_D(pp * 1)\| \cup .. \cup \|I_D(pp * n)\|\right)$
  By the translation of pp, we have that $\langle\langle pp + n\rangle\rangle = \langle\langle pp * 1\rangle\rangle$ *UNION .. UNION* $\langle\langle pp * n\rangle\rangle$

  By the previous bullet, we proved that $(\|a_1\|..\|a_x\|) \in \mathcal{E}(\langle\langle pp' * i\rangle\rangle, \|D\|)$ for some i.

  So, if holds that $(\|a_1\|..\|a_x\|) \in \mathcal{E}(\langle\langle pp + n\rangle\rangle, \|D\|)$

Summing up, we conclude that since $(a_1...a_n) \in I_D(dp)$, it also holds that $(\|\hat{a_1}\|...\|\hat{a_n}\|) \in \mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|)$, for all the different forms of a dp, so $\|I_D(dp)\| \subseteq \mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|)$

- For the opposite direction, that is $\mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|) \subseteq \|I_D(dp)\|$ we also prove it for both forms of a dp:

– if dp an argument pattern ap, we have again two cases:
  a) $ap = \langle\{s_1, \ldots, s_m\}, cp\rangle$, with $s_i \in \mathcal{P}$ and $cp \in \mathcal{P} \cup V$
  b) $ap = \langle pr[f], cp\rangle$, with $pr \in V$, $cp \in \mathcal{P} \cup V$ and f a filter

  a) Using the lemma 1, we have that a matching instance of $\langle\langle ap\rangle\rangle$ is a particular RA-node, whose set of predecessors as well as its unique successor, are all of type *I*. Let *ra* such an node and $\|\hat{ra}\| \in \mathcal{E}(\langle\langle ap\rangle\rangle, \|D\|)$
  The translation of ap, according to the section 5.3 gives the following graph pattern :

$\langle\langle ap\rangle\rangle =$

?_s_1	rdf:type	aif:I-Node.
?_s_1	aif:claimText	s_1
	...	
?_s_m	rdf:type	aif:I-Node.
?_s_m	aif:claimText	s_m
?_c	rdf:type	aif:I-Node.
?_c	aif:claimText	cp.
?_ra	rdf:type	aif:RA-node.
?_s_1	aif:Premise	?_ra.
	...	
?_s_m	aif:Premise	?_ra.
?_ra	aif:Conclusion	?_c

Since $\|\hat{ra}\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$, it means that $in(ra)$ consists of m I-nodes with values $s_1, ..s_m$, and $out(ra)$, of an I-node with value $\mu(cp)$. Furthermore, given that $\|\hat{ra}\| \in \|D\|$, we conclude that $\exists a, a \in A$ for which $\|\hat{a}\|$ includes that ra.

By the mapping definition 9, we conclude that this argument will have value $a = \langle \{s_1, \ldots, s_m\}, c \rangle$.

It is obvious that a constitutes a valid match for the argument pattern ap. Regarding the $\mu(cp)$, in case that in the initial argument pattern, cp was a constant value, then $\mu(cp) = cp$, which is correct, while in the case it was a variable $\mu(cp) = c$ which is also correct. As a result, we conclude that $a \in I_D(ap)$.

b) For the case when $ap = \langle v[f], cp \rangle$, for which, the premise pattern $pr = v[f]$ we have:
The translation of ap, according to the section 5.3 gives the following graph pattern :

$$\langle\langle ap \rangle\rangle = \begin{array}{lll} ?\_v & \text{rdf:type} & \text{aif:I-Node.} \\ ?\_v & \text{aif:claimText} & v \\ ?\_v & \text{aif:Premise} & ?\_ra. \\ ?\_c & \text{rdf:type} & \text{aif:I-Node.} \\ ?\_c & \text{aif:claimText} & cp. \\ ?\_ra & \text{aif:Conclusion} & ?\_c \\ ?\_ra & \text{rdf:type} & \text{aif:RA-node.} \\ \langle\langle f \rangle\rangle & & \end{array}$$

Let an RA-node $ra \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$ a matching instance of $\langle\langle ap \rangle\rangle$. For this ra, the set $in(ra)$ will include all the I-nodes of the mapping set of the variable $\_v$ and they will have the values lying in the bindings of the variable v. Let $\{p_1, ..p_n\}$ these values. Respectively, the $out(ra)$, will include the I-node that will bind with the variable $?\_c$, and its value will be the $\mu(cp)$. Let c this value. In addition, since ra is a valid match, it means that it will also satisfy the $\langle\langle f \rangle\rangle$.

Given that $\|\hat{ra}\| \in \|D\|$, we infer that $\exists a, a \in A$, for which $\|\hat{a}\|$ includes that ra. The mapping definition 9, imposes that $a = \langle \{p_1, \ldots, p_n\}, c \rangle$. In order to hold that $a \in I_D(ap)$, it must also hold that $a \vDash f$.

- Let $f_{incl} = [/\{s_1, .., s_k\}]$. $\langle\langle f_{incl} \rangle\rangle$ requires for the matching RA-node ra', the set $in(ra')$ to include I-nodes with values $s_1, ..s_k$. Since ra is a valid match, it means that $\{s_1, ..s_k\} \subseteq in(ra)$, hence, without loss of generality, we assume that $in(ra) = \{s_1, ..s_k, .., p_n\}$. Consequently, the argument a will be $a = \langle \{s_1, ..s_k, .., p_n\}, c \rangle$, which satisfies the f_{incl} that requires $\{s_1, ..s_k\} \subseteq prem(a)$. As a result we conclude that $a \vDash f_{incl}$ and $a \in I_D(ap)$.

- Let $f_{incl} = [/v_2]$, with $v_2 \in V$. $\langle\langle f_{incl} \rangle\rangle$ requires for the matching ra', that there is another ra'', such that $\forall p : p \in in(ra'')$, it also holds that $p \in in(ra')$. This join happens between the variables that bind with these $in(.)$ sets and which are originated by the translation of the current argument pattern, and a second one, let ap_2.
  ra is a valid match, which means that there exists such an ra'', s.t $\|ra''\| \in \|D\|$, which is a matching instance of ap_2 and for which $in(ra'') \subseteq in(ra)$. In correspondence with ra, it also holds that $\exists a', a' \in A$, for which $\|\hat{a'}\|$ includes ra''. As a result, we have that $a' \in I_D(ap_2)$, $prem(a') \subseteq prem(a)$, and thus $a \vDash f_{incl}$. So $a \in I_D(ap)$.

- The proof for the rest of the filters proceed in the same way and we are not going to show them separately.

Summing up, we have shown that for both forms of an argument pattern ap, the following holds:

$\forall \|ra\| \in \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$, $\exists a \in I_D(ap)$, for which $\langle\langle \hat{a} \rangle\rangle$ includes that ra.

Consequently $\|I_D(ap)\| \subseteq \mathcal{E}(\langle\langle ap \rangle\rangle, \|D\|)$

– Let the second form of a dialogue pattern, in which $dp : ap\ pp\ dp'$, with pp a path pattern and dp' another dialogue pattern.
The translation of dp, according to section 5.3 gives the following graph pattern:

$$\langle\langle dp\rangle\rangle = \langle\langle ap_1\rangle\rangle \wedge \langle\langle pp\rangle\rangle \wedge \langle\langle dp'\rangle\rangle$$

The path pattern pp lies between two argument patterns, ap and $head(dp')$, where $head(dp')$ is the first argument pattern appearing in dp'.

Let $(\|ra_1\|, \|ra_2\|,.. \|ra_n\|)$ a sequence of *ras* that satisfies $\langle\langle dp\rangle\rangle$. This means that $\|ra_1\| \in \mathcal{E}(\langle\langle ap_1\rangle\rangle, \|D\|)$, there is some ra_k in the sequence, for which $\|ra_k\| \in \mathcal{E}(\langle\langle head(dp')\rangle\rangle, \|D\|)$, the sequence $(\|ra_1\|,.. \|ra_k\|) \in \mathcal{E}(\langle\langle pp\rangle\rangle, \|D\|)$ and the sequence $(\|ra_k\|,.. \|ra_n\|) \in \mathcal{E}(\langle\langle dp'\rangle\rangle, \|D\|)$.

By the previous bullet, we proved that $\exists a \in A$, s.t. $a \in I_D(ap)$, for which $\|\hat{a}\|$ includes the ra_1 and $\exists a_k \in A$, s.t. $a_k \in I_D(head(dp'))$. Here, given that $(\|ra_1\|,.. \|ra_k\|) \in \mathcal{E}(\langle\langle pp\rangle\rangle, \|D\|)$, we need to prove that the sequence $(a_1,..,a_k) \in I_D(pp)$

We will present the proof in a recursive way, to capture all the possible forms a path pattern may have.

- for the case when pp is a single relation we will focus on the *rebut* relation type. As before, the rest cases are proved in the same way. The translation of the *rebut* relation is

  $$\langle\langle rebut\rangle\rangle = \left(\langle\langle cp_1\rangle\rangle.\_v_i \textit{ aif:conflicting-element/aif:conflicted-element } \langle\langle cp_2\rangle\rangle.\_v_i\right)$$

  where cp_1 is the conclusion pattern of ap, and cp_2 the conclusion pattern of $head(dp')$.

  Since $\|ra_1\|$ and $\|ra_k\|$ satisfy the translated argument patterns $\langle\langle ap_1\rangle\rangle$ and $\langle\langle head(dp')\rangle\rangle$ respectively, and the sequence $(\|ra_1\|..\|ra_k\|)$ satisfies the $\langle\langle pp\rangle\rangle$, it means that there is some *ca* node, between the I-nodes $out(ra_1)$ and $out(ra_k)$, with which they are linked with the edges *conflicting-element* and *conflicted-element*, respectively. However, according to definition 9, that *ca* originates from the conflict $c_1 \nmid c_2$ between two propositions c_1, c_2, for which $c_1 = concl(a)$ and $c_2 = concl(a_k)$, and $\hat{c_1} = out(ra_1)$ and $\hat{c_k} = out(ra_k)$. It is obvious then, that $(a_1 a_k) \in I_D(rebut)$.

- for $pp = pp_1/pp_2$ we assume that there is a radom argument pattern ap' between pp_1 and pp_2 and we have that it is equal to:

  $$\langle\langle pp_1/pp_2\rangle\rangle = \langle\langle pp_1\rangle\rangle \wedge \langle\langle ap'\rangle\rangle \wedge \langle\langle pp_2\rangle\rangle$$

  Since $\|ra_1\|$ and $\|ra_k\|$ satisfy the translated argument patterns $\langle\langle ap_1\rangle\rangle$ and $\langle\langle head(dp')\rangle\rangle$ respectively, and the sequence $(\|ra_1\|..\|ra_k\|)$ satisfies the $\langle\langle pp\rangle\rangle$ it means that for some $\|ra_m\|$ it will hold that $\|ra_1\|..\|ra_m\|$ will satisfy $\langle\langle pp_1\rangle\rangle$ and $\|ra_m\|..\|ra_k\|$ will satisfy $\langle\langle pp_2\rangle\rangle$. Now we need to prove that $(a_1..a_m) \in I_D(pp_1)$ and $(a_m..a_k) \in I_D(pp_2)$. The proof progresses recursively in this way, until both of pp_1 and pp_2 conclude to be a single relation. This case was proved in the previous bullet. So we conclude that $(a_1..a_k) \in I_D(pp_1/pp_2)$.

- the case $pp = pp' * n$, with $n \geq 1$, is written alternatively as $pp'/../pp'$ (n repetitions). Given that the sequence$(\|ra_1\|..\|ra_k\|)$ satisfies the $\langle\langle pp' * n\rangle\rangle$ it means that it satisfies the $\langle\langle pp'/pp' * (n-1)\rangle\rangle$ That case is reduced in the previous one, with $pp_1 = pp'$ and $pp_2 = pp' * (n-1)$, so the proof results to be the same as before. So we conclude that $(a_1..a_k) \in I_D(pp' * n)$

- for the case $pp = pp' + n$, with $n \geq 1$, we know that

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

  Since $(\|ra_1\|,.. \|ra_k\|) \in \mathcal{E}(\langle\langle pp\rangle\rangle, \|D\|)$, it means that $(\|ra_1\|,.. \|ra_k\|) \in \mathcal{E}(\langle\langle pp' * i\rangle\rangle, \|D\|)$ for some $1 \geq i \geq n$. This case is reduced to the previous bullet and the proof continues there.
  So we have proved that $(a_1..a_k) \in I_D(pp' + n)$

Summing up, we conclude that since$(\|\hat{a_1}\|...\|\hat{a_n}\|) \in \mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|)$, for all the different forms of a dp, it also holds that $(a_1...a_n) \in I_D(dp)$, so $\mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|) \subseteq \|I_D(dp)\|$.

Finally, we have showed that $\mathcal{E}(\langle\langle dp\rangle\rangle, \|D\|) = \|I_D(dp)\|$

References

1. 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.
2. Carlos Chesñevar, Sanjay Modgil, Iyad Rahwan, Chris Reed, Guillermo Simari, Matthew South, Gerard Vreeswijk, Steven Willmott, et al. Towards an argument interchange format. *The knowledge engineering review*, 21(4):293–316, 2006.
3. Brickley Dan and Guha R. V. Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/rdf-primer/, 2004.
4. Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrs Taylor. *Cypher: An Evolving Query Language for Property Graphs*. ACM, 1 2018.
5. Manola Frank and Miller Eric. RDF primer. http://www.w3.org/TR/rdf-primer/, 2004.
6. Steve H. Garlik, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/.
7. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.
8. Chris Reed, Simon Wells, Joseph Devereux, and Glenn Rowe. Aif+: Dialogue in the argument interchange format. *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, 172:311, 2008.
9. Douglas N. Walton. *Argumentation Schemes for Presumptive Reasoning*. L. Erlbaum Associates, 1996.