Freie Universität Berlin

# WIQA-PL Web Information Quality Assessment Policy Language Specification

**Chris Bizer**
chris@bizer.de

November 2006

# Contents

# Chapter 1

# The WIQA Framework

The *WIQA - Information Quality Assessment Framework* is a set of software components for filtering information using different quality-based information filtering policies. The WIQA framework can be employed by applications which process information of uncertain quality and want to enable users to filter information using different policies. The framework has been designed to fulfill the following requirements:

**Flexible Representation of Quality-Related Meta-information.** Information quality assessment relies on a wide range of different quality indicators. Which quality indicators are relevant depends on the application domain and the quality dimensions to be assessed. Important quality indicators in the context of web-based information systems are provenance information, ratings, and background information about information providers. The WIQA framework uses Named Graphs as a flexible data model for representing information together with quality related meta-information.

**Support for Information Filtering Policies.** The relevance of different quality dimensions and the metrics used to assess these dimensions depend on the application domain, the quality indicators available, the task at hand, and the subjective preferences of the information consumer. Therefore, information consumers use a wide range of different policies for determining whether to accept or reject information. The WIQA framework allows various policies to be employed for filtering information. Policies are expressed using a declarative policy language and can combine context-, content-, and rating-based quality assessment metrics.

**Explaining Filtering Decisions.** The accuracy of assessment results is often uncertain due to the limited availability of quality indicators and the often uncertain quality of the quality indicators themselves. Therefore, the final subjective decision of an information consumer whether to trust or distrust assessment results depends on his understanding of the quality indicators and the assessment metrics that have been used in the assessment process. In order to support information consumers in their trust decision, the WIQA framework can generate detailed explanations about filtering decisions.

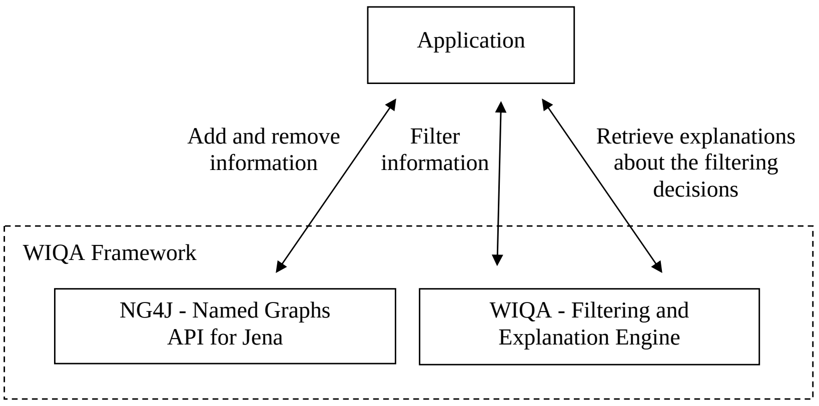
Application
Add and remove information
Filter information
Retrieve explanations about the filtering decisions
WIQA Framework
NG4J - Named Graphs API for Jena
WIQA - Filtering and Explanation Engine

Figure 1.1: Overview about the WIQA framework.

Figure 1.1 gives an overview about the components of the WIQA framework and illustrates how applications interact with the framework. The WIQA framework consists of the NG4J - Named Graphs API for Jena and the WIQA - Filtering and Explanation Engine.

*NG4J - Named Graphs API for Jena* is a software toolkit for creating, manipulating, persisting, and exchanging sets of named graphs. Graph sets can be stored in memory or in a relational database. The API provides parsers and serializers for the TriX, TriG, and RDF/XML syntaxes and allows graph sets to be signed using the Semantic Web Publishing vocabulary. The WIQA framework uses NG4J to store information together with quality-related meta-information.

The *WIQA - Filtering and Explanation Engine* determines the subset of the triples contained in a set of named graphs that match a given filtering policy. Filtering policies are expressed using the WIQA-PL policy language. Applications present a set of named graphs and a WIQA-PL policy to the WIQA - Filtering and Explanation Engine. Based on the policy, the engine generates a view on the graph set containing all triples that fulfill the policy.

This *set of accepted triples* is returned to the application. Figure 1.2 illustrates the process of promoting triples from the set of named graphs into the set of accepted triples.

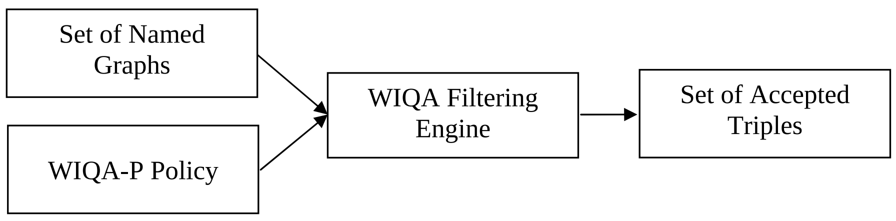

Figure 1.2: Overview about the filtering process.

The WIQA - Filtering and Explanation Engine can generate explanations about filtering decisions. An application can present an accepted triple to the engine which returns an explanation why the triple satisfies the given policy. The engine can generate two types of explanations: Textual explanations and RDF explanations. Textual explanations can be displayed directly to the end-user. RDF explanations may be used by the application for further processing.

The following chapters describe the WIQA framework in detail:

**Chapter 2: Expressing Information Filtering Policies.** Within the WIQA framework, information filtering policies are expressed using the WIQA-PL policy language. This chapter introduces the WIQA-PL language constructs and explains how the language is used to express filtering policies.

**Chapter 3: Explaining Assessment Results.** This chapter describes the capabilities of the WIQA framework to explain filtering decisions.

**Chapter 4: Implementation.** This chapter describes the implementation of the WIQA framework and explains how applications interact with the framework.

**Chapter ??: The WIQA Browser** is an example application that uses the WIQA framework. The browser demonstrates how information quality filtering capabilities can be integrated into a standard Web browser. The browser enables users to extract RDF data from Web pages. The extracted data is stored together with provenance information in a local repository. The content of the local repository can be filtered using WIQA-PL policies. The user may retrieve explanations why displayed information satisfies the selected policy.

**Chapter 5: Related Work.** This chapter compares the WIQA Framework with related approaches.

# Chapter 2

# Expressing Information Filtering Policies

In general, a policy can be seen as a set of declarative rules which governs the behavior of an information system [CLW02, FWS05]. Instead of hard-coding system behavior at design time, policies allow to dynamically alter system behavior at run-time. Policy-based approaches to system management have been applied in various application domains such as network management, authentication, access control, privacy, digital rights management, and quality of service assurance [FWS05, Pie04]. Policies are usually expressed using a declarative policy language. Examples of standardized policy languages are the eXtensible Access Control Markup Language (XACML) [Mos05], an OASIS standard for defining access control policies, and the Platform for Privacy Preferences (P3P) [Mar02], a W3C standard for expressing privacy policies.

WIQA-PL policies define which information is filtered positive by the WIQA filtering engine. WIQA policies are expressed using the WIQA-PL policy language. This chapter introduces the WIQA-PL language constructs and explains how the language is used to express filtering policies.

## 2.1 Basic Grammar

Figure 2.1 shows the Extended Backus-Naur Form (EBNF) [ISO96] definition for the basic grammar of the WIQA-PL policy language. The complete WIQA-PL grammar is given in appendix **??**.

Policies can be grouped into policy suites. A policy suite consists of a block of namespace prefix declarations and a set of policies. The namespace prefixes may be used later in PATTERN clauses to abbreviate URI references

```
1.  PolicySuite          ::= PrefixDeclaration*
2.                           Policy+
3.  PrefixDeclaration    ::= 'PREFIX' PrefixID Uriref
4.  Policy               ::= PolicyName
5.                           PolicyDescription?
6.                           PolicyPattern
7.                           RDFExplanationClause
8.  PolicyName           ::= 'NAME' Literal
9.  PolicyDescription    ::= 'DESCRIPTION' Literal
10. PolicyPattern        ::= 'PATTERN' PatternSet
```

Figure 2.1: EBNF grammar for the basic structure of a WIQA policy suite.

using the QName abbreviation mechanism [BHL06]. The PREFIX keyword associates a prefix label with a URI. A QName is mapped into an URI reference by concatenating its local part to the URI corresponding to its prefix.

```
1.  PatternSet           ::= '{' ExplanationClause?
2.                           GraphPattern*
3.                           FilterClause* '}'
4.  FilterClause         ::= 'FILTER' FilterExpression '.'
5.  ExplanationClause    ::= 'EXPL' ExplanationTemplates '.'
7.  GraphPattern         ::= GraphName '{'
8.                           ExplanationClause?
9.                           TriplePattern+
10.                          FilterClause* '}'
11. GraphName            ::= 'GRAPH' VariableOrUriOrANY
12. TriplePattern        ::= URIOrBnodeOrVariableOrReference
13.                          URIOrVariableOrReference
14.                          URIOrBnodeOrLiteralOrVariableOrReference '.'
15. VariableOrUriOrANY   ::= Variable | URI | 'ANY'
16. URIOrBnodeOrVariableOrReference
17.                      ::= URI | Bnode | Variable | Reference
18. URIOrVariableOrReference
19.                      ::= URI | Variable | Reference
20. URIOrBnodeOrLiteralOrVariableOrReference
21.                      ::= URI | Bnode | Literal | Variable |
22.                          Reference
23. Variable             ::= '?' String
24. Reference            ::= '?GRAPH' | '?SUBJ' |'?PRED' | '?OBJ'
```

Figure 2.2: EBNF grammar of the PATTERN clause.

Each policy consists of a NAME, a DESCRIPTION, and a PATTERN clause. The NAME clause specifies a display name for the policy. The

DESCRIPTION clause specifies a description for the policy that contains details about the quality indicators and assessment metrics that are used by the policy. The PATTERN clause specifies a set of conditions that triples have to satisfy in order to be filtered positive. The grammar of the PATTERN clause is shown in figure 2.2. The grammar is based on the grammar of the SPARQL query language [PS05] in order to make it easier for people who already know SPARQL to learn WIQA-PL. A PATTERN clause may contain:

**Graph Patterns** which refer to the triples of the graph set to be filtered using a set of special, referring variables. The WIQA filtering engine matches the graph patterns against the set of named graphs to be filtered. The set of accepted triples is constructed from the matching results afterward. Chapter 2.2 describes how graph patterns are matched against graph sets. Chapter 2.3 describes how referring variables are used to link graph patters to the triples to be filtered, and how the set of accepted triples is constructed from the matching solutions.

**Filter Clauses** may be used to further restrict matching solutions. Filter clauses contain boolean-valued expressions consisting of variables, RDF terms, comparison operators, and function calls. Multiple expressions may be combined using logical operators. Chapter 2.5 explains filter clauses in detail.

**Functions Calls.** Quality-based information filtering policies may involve complex rating algorithms and statistical calculations. The WIQA filtering engine provides an extension mechanism for including arbitrary, application domain specific assessment functions into policies. Chapter 2.6 explains how function calls are used within filter clauses and describes the extension functions that have been implemented for the WIQA framework so far.

**Explanation Clauses.** Graph patterns may contain explanation clauses which define explanation templates. The templates consist of text fragments and variables. When a user requests an explanation why a triple satisfies a given policy, these templates are instantiated with variable bindings from the matching solutions. Chapter 3 describes the generation of explanations.

## 2.2 Graph Pattern Matching

A graph pattern consists of a *graph name pattern* and a set of *triple patterns*. The graph name pattern may either consist of a URI reference, a variable, or the keyword ANY. The graph name pattern ANY matches all graph names. Triple patterns consist of a subject, a predicate, and an object. The subject of a triple pattern may contain a URI, a bNode, or a variable. The predicate of a triple pattern has to be URI. The object of a triple pattern may contain a URI, a bNode, a literal, or a variable.

The variables contained in graph patterns are bound to RDF terms by matching the graph patterns against a set of named graphs. Let $NG$ be a set of named graphs and $GP$ a set of graph patterns. Let $V$ be the set of all variables contained in $GP$, let $RT$ be the set of all RDF terms contained in $NG$, and let $GN$ be the set of all graph name URIs in $NG$. A matching solution $s$ assigns an RDF term from $RT$ to each variable in $V$. $GP$ matches $NG$ with the matching solution $s$ if each variable in $GP$ may be substituted with its value from $s$ and if the keyword ANY may be substituted with a graph name from $GN$ so that each graph pattern in $GP$ is equal to or is a subgraph of a graph in $NG$.

Matching a set of graph patterns against a set of named graphs results in a *solution set* as multiple solutions may satisfy the condition above. This set is empty if no solution fulfills the condition above.

Figure 2.3 shows an example graph pattern. The WIQA-PL syntax for graph patterns requires each pattern to be introduced with the keyword GRAPH. The set of triple patterns is enclosed with curly brackets. Variable names are prefixed with a question mark. The graph pattern shown in figure 2.3 consists of a graph name pattern and one triple pattern. The graph name pattern requires the matching graph to be named `fd:BackgroundInformation`. The triple pattern matches all triples with the predicate `foaf:name`. The subjects and objects of these triples are bound to the variables `?var1` and `?var2`. Matching the graph pattern against the example graph set from chapter **??** results in the solution set shown in table 2.1.

```
GRAPH fd:BackgroundInformation
  { ?var1 foaf:name ?var2 . }
```

Figure 2.3: Graph pattern 1.

Figure 2.4 shows another example graph pattern. The graph pattern consist of the graph name pattern ANY and two triple patterns. The graph pattern matches all graphs, independent of their graph name, that contain a triple having a `foaf:name` predicate and a triple having a `fin:country` predicate and an `iso:DE` object. Both triple patterns contain the variable `?var1` as subject. Therefore pairs of matching triples have to share the same subject.

| | *?var1* | *?var2* |
|---|---|---|
| 1. | `<mailto:peterSmith@deutsche-bank.de>` | `"Peter Smith" dtype:string` |
| 2. | `<urn:x-DUNS:332907323>` | `"Deutsche Bank" dtype:string` |
| 3. | `<mailto:reynolds@ft.com>` | `"John Reynolds" dtype:string` |
| 4. | `<urn:x-DUNS:42307553>` | `"Financial Times" dtype:string` |
| 5. | `<mailto:mark@scott.com>` | `"Mark Scott" dtype:string` |

Table 2.1: Solution set from matching graph pattern 1 against the example graph set.

Matching the graph pattern against the example graph set from chapter **??** results in the solution set shown in table 2.2.

```
1. GRAPH ANY
2.   { ?var1 foaf:name ?var2 .
3.     ?var1 fin:country iso:DE . }
```

Figure 2.4: Graph pattern 2.

| | *?var1* | *?var2* |
|---|---|---|
| 1. | `<mailto:peterSmith@deutsche-bank.de>` | `"Peter Smith" xsd:string` |
| 2. | `<urn:x-DUNS:332907323>` | `"Deutsche Bank" xsd:string` |

Table 2.2: Solution set from matching graph pattern 2 against the example graph set.

Figure 2.5 shows a graph pattern set consisting of two graph patterns. The pattern set demonstrates how the condition, that information should originate from John Reynolds, is expressed by combining two graph patterns. The first graph pattern consists only of variables and therefore matches every triple. The second graph pattern matches only triples in the graph `fd:GraphFromAggregator` that use the Semantic Web Publishing vocabulary to describe the origin of the graphs that have been asserted by Dave Reynolds. The names of these graphs are bound to the variable `?graph`. When both patterns are matched simultaneously against a set of named graphs, the second pattern works like an additional filter for the first pattern: The second pattern binds the names of all graphs that have been asserted by John Reynolds to the variable `?graph`. With this constraint for the variable `?graph`, the first pattern matches only triples within graphs from John Reynolds and the subjects, predicates, and objects of these triples are bound to the variables `?var1`,

`?var2`, and `?var3`. Matching both graph patterns together against the example graph set from chapter **??** results in the solution set shown in table 2.3.

```
GRAPH ?graph
  { ?var1 ?var2 ?var3 . }

GRAPH fd:GraphFromAggregator
  { ?graph   swp:assertedBy ?warrant .
    ?warrant swp:authority <mailto:reynolds@ft.com> . }
```

Figure 2.5: Graph pattern set.

| *Variable* | *Solution 1* | *Solution 2* |
|---|---|---|
| *?var1* | `<urn:x-DUNS:316067164>` | `<urn:x-DUNS:047897855>` |
| *?var2* | `fin:news` | `fin:news` |
| *?var3* | `"Siemens AG ..."@EN` | `"Intel has ..."@EN` |
| *?graph* | `fd:GraphFromJohnReynolds` | `fd:GraphFromJohnReynolds` |
| *?warrant* | `fd:JrWarrant` | `fd:JrWarrant` |

Table 2.3: Solution set from matching the graph pattern set from figure 2.5 against the example graph set.

## 2.3 Accepting Triples

The WIQA-PL policy language uses graph patterns to represent conditions that triples have to satisfy in order to be filtered positive. When a policy is applied, the WIQA filtering engine checks for each triple in the graph set to be filtered whether it satisfies the conditions given in the pattern clause of the current policy. The triples that satisfy the conditions are included into the set of accepted triples.

Conditions are expressed as graph patterns which refer to the triples in the graph set to be filtered using a set of special variables. These *referring variables* connect the graph patterns in the pattern clause with the triples in the graph set to be filtered. The referring variables are shown in table 2.4. If the variable `?SUBJ` is used in any graph pattern, then only triples with a subject that equals a binding of the variable `?SUBJ` are accepted. If the variable `?PRED` is used in any pattern, then only triples are filtered positive which have a predicate matching a binding of the variable `?PRED`. If the variable `?OBJ` is used, then only triples are filtered positive with an object matching a `?OBJ`

| *Variable* | *Description* |
|---|---|
| ?SUBJ | Reference to subject of a triple. |
| ?PRED | Reference to predicate of a triple. |
| ?OBJ | Reference to object of a triple. |
| ?GRAPH | Reference to the graph containing a triple. |

Table 2.4: WIQA-PL referring variables.

value. If the variable `?GRAPH` is used in any pattern, then only those triples are accepted that occur in a graph that is named with an URI which equals a binding of the variable `?GRAPH`. If multiple referring variables are used within the same pattern clause then triples are accepted only if they match values of all referring variables and these values occur in a single matching solution.

When a policy is applied against a graph set, the WIQA engine generates the set of accepted triples by conducting the following steps:

1. The filtering engine adds the graph pattern `GRAPH ?GRAPH {` `?SUBJ, ?PRED, ?OBJ` `}` to the set of graph patterns given by the pattern clause of the policy. In the following, this graph pattern will be called *root pattern*.

2. The engine matches the extended pattern set against the graph set to be filtered. This results into a solution set.

3. The engine generates an accepted triple from each distinct set of values of the variables `?SUBJ, ?PRED and ?OBJ` in the solution set.

Figure 2.6 shows the WIQA-PL representation of the policy "Accept only information which has been asserted by German analysts". Lines 1-5 define the namespace prefixes which are used in the pattern clause later. Line 7 and 8 specify the policy name and policy description. The pattern clause restricts information to originate from German analysts. It consists of two graph patterns. The first graph pattern requires provenance information about graphs to be contained in the graph `fd:GraphFromAggregator`. It contains two triple patterns which require provenance information to be expressed using the SWP properties `swp:assertedBy` and `swp:authority`. The first pattern binds the names of asserted graphs to the referring variable `?GRAPH`. The second pattern binds URIs that identify authorities to the variable `?authority`. The second triple pattern is connected with the first one by sharing the variable `?warrant`.

The second graph pattern requires authorities to be an instance of the class `fin:Analyst` and to have a `fin:country` property with the value

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swp: <http://www.w3.org/2004/03/trix/swp-2/> .
@prefix iso: <http://www.daml.org/2001/09/countries/iso-3166-ont#> .
@prefix fin: <http://www.fu-berlin/suhl/bizer/2006/FinVoc/> .
@prefix fd: <http://www.fu-berlin/suhl/bizer/exampleDataset#> .

NAME "Information from German analysts"
DESCRIPTION "Use only information which has been asserted by
             German analysts."
PATTERN
    {
      GRAPH fd:GraphFromAggregator
        { ?GRAPH swp:assertedBy ?warrant .
          ?warrant swp:authority ?authority . }

      GRAPH fd:BackgroundInformation
        { ?authority rdf:type fin:Analyst .
          ?authority fin:country iso:DE . }
    }
```

Figure 2.6: WIQA-PL policy: Use only information which has been asserted by German analysts.

`iso:DE`. The triples that describe authorities have to occur in the graph `fd:BackgroundInformation`.

When the policy is applied against the example graph set from chapter **??**, the second graph pattern matchs the analyst Peter Smith and the variable `?authority` is bound to `<mailto:peterSmith@deutsche-bank.de>`. With this binding of the variable `?authority`, the first graph pattern matches the graph `fd:GraphFromAggregator` and binds the value `fd:GraphFromPeterSmith` to the referring variable `?GRAPH`. With this binding of the variable `?GRAPH`, the WIQA engine filters all triples that are contained in `fd:GraphFromPeterSmith` positive. The engine would therefore return the set of accepted triples shown in figure 2.7 to the application.

```
<urn:x-ISIN:DE0007236101> fin:positiveAnalystReport "As Siemens agrees
  partnership with Novell unit SUSE ..."@EN .
<urn:x-ISIN:US4581401001> fin:negativeAnalystReport "Chiphersteller
  Intel will nach Firmenangaben mit milliardenschweren ..."@DE .
```

Figure 2.7: Accepted triples.

## 2.4 Context Variables

Filtering policies may rely on information about the application context in which they are applied. Subjective policies might, for instance, require information about the user who applies them; time-dependent policies might require the current time.

Applications can provide the WIQA engine with information about the application context by setting context variables at run-time. Context variables can be used within WIQA-PL policies. The names of context variables are written in uppercase letters in order to distinguish them from other variables. Before applying a policy, the WIQA engine substitutes all context variables within the policy with their values set by the application. As the variable names ?GRAPH, ?SUBJ, ?PRED, ?OBJ are already reserved for the referring variables, these names cannot be used for context variables.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swp: <http://www.w3.org/2004/03/trix/swp-2/> .
@prefix iso: <http://www.daml.org/2001/09/countries/iso-3166-ont#> .
@prefix fin: <http://www.fu-berlin/suhl/bizer/2006/FinVoc/> .
@prefix fd: <http://www.fu-berlin/suhl/bizer/exampleDataset#> .

NAME "Information from positively rated information providers"
DESCRIPTION "Use only information from information providers that
             I have rated positive."
PATTERN
    {
      GRAPH fd:GraphFromAggregator
        { ?GRAPH swp:assertedBy ?warrant .
          ?warrant swp:authority ?authority . }

      GRAPH ?myGraph
        { ?USER fin:positiveRating ?authority . }

      GRAPH fd:GraphFromAggregator
        { ?myGraph swp:assertedBy ?warrant2 .
          ?warrant2 swp:authority ?USER . }
    }
```

Figure 2.8: WIQA-PL policy: Use only information from information providers that I have rated positive.

Figure 2.8 shows the WIQA-PL representation of the policy “Use only information from information providers that I have rated positive”. In order to determine which information providers have been rated positive by the current user, the WIQA engine requires the URI reference identifying the current user. This URI reference is represented by the context variable `?USER` within the policy.

The PATTERN clause of the policy contains three graph patterns. The first pattern binds the URI references identifying information providers to the variable `?authority`. The second graph pattern restricts `?authority` bindings to information providers that have been rated positive by the current user. The names of the graphs that contain the ratings are bound to the variable `?myGraph`. The third graph pattern ensures that ratings originate from the current user by requiring the graph `fd:GraphFromAggregator` to contain the statements that `?myGraph` was asserted by the current user.

## 2.5 Filters

FILTER clauses restrict solution sets according to a given expression. They eliminate any solution from the solution set that, when substituted into the expression, results in a boolean value of false or produces an error.

```
FilterClause       ::= 'FILTER' Expression | FunctionCall
Expression         ::= '(' ConditionalOrExpression ')'
ConditionalOrExpression ::= ConditionalAndExpression ( '||'
                       ConditionalAndExpression )*
ConditionalAndExpression ::= LogicalValue ( '&&' LogicalValue )*
LogicalValue     ::= RelationalExpression
RelationalExpression ::= NumericExpression ( '=' NumericExpression |
                       '!=' NumericExpression |
                       '<'  NumericExpression  |
                       '>'  NumericExpression |
                       '<=' NumericExpression |
                       '>=' NumericExpression )?
NumericExpression  ::= AdditiveExpression
AdditiveExpression ::= MultiplicativeExpression (
                       '+' MultiplicativeExpression |
                       '-' MultiplicativeExpression )*
MultiplicativeExpression ::= UnaryExpression ( '*' UnaryExpression |
                       '/' UnaryExpression )*
UnaryExpression    ::= '!' PrimaryExpression |
                       '+' PrimaryExpression |
                       '-' PrimaryExpression |
                       PrimaryExpression
PrimaryExpression  ::= Expression | FunctionCall | URIref |
                       RDFLiteral | NumericLiteral | BooleanLiteral |
                       BlankNode | Variable
```

Figure 2.9: Grammar of the FILTER clause.

Figure 2.9 shows the grammar of the FILTER clause. Filter expressions may consist of relational, logical, and numeric expressions and may include function calls:

**Relational Expressions** compare variables, RDF terms, and numeric expressions to each other. Relational expressions may use the following comparison operators: = (equal), != (not equal), > (greater than), < (less than), >= (greater than or equal to) and <= (less than or equal to). The evaluation of a relational expressions results either in the boolean value true or false. WIQA-PL uses the same rules for comparing RDF terms as the SPARQL query language. These rules are defined in section 11.3 of the SPARQL specification [PS05]. An example of a relational expression is `?authority = <mailto:chris@bizer.de>`, meaning that the value of the variable `?authority` has to be the URI `<mailto:chris@bizer.de>`.

**Logical Expressions** connect multiple expressions using the && (logical AND) and || (logical OR) operators or negate the result of an expression using the ! (logical NOT) operator. An example of a logical expression using the && operator is `?authority != <mailto:chris@bizer.de> && ?price < 20`, meaning that the value of the variable `?authority` has to be different from the URI `<mailto:chis@bizer.de>` and the value of the variable `?price` has to be smaller than 20.

**Numeric Expressions** conduct calculations. Numeric expressions may use the + (add), - (substract), * (multiply), and / (divide) operators. WIQA-PL uses the same functions for evaluating numeric expressions as the SPARQL query language. These functions are defined in section 11.3 of the SPARQL specification [PS05]. An example of a relational expression that includes a numeric expression is `?priceYen < ?priceDollar * 112`.

Figure 2.10 shows the WIQA-PL policy "Accept only information that has been asserted after January 1st, 2006 by analysts who achieved a StarMine score above 80". The policy uses two filter clauses: The filter clause in line 10 restricts bindings of the variable `?date` to values that are greater than 2006-01-01. The FILTER clause in line 19 restricts bindings of the variable `?benchmark` to integer values greater than 80.

```
NAME "New information from highly rated analysts"
DESCRIPTION "Accept only information that has been asserted
             after January 1st, 2006 by analysts who achieved
             a StarMine score above 80"
PATTERN
      {
       GRAPH fd:GraphFromAggregator
          { ?GRAPH swp:assertedBy ?warrant .
            ?warrant swp:authority ?authority .
            ?warrant dc:date ?date .
            FILTER (?date > "2006-01-01"^^xsd:date) . }

       GRAPH fd:BackgroundInformation
         { ?authority rdf:type fin:Analyst .
           ?authority fin:benchmark ?benchmark .
           FILTER (?benchmark > "80"^^xsd:integer) . }
      }
```

Figure 2.10: WIQA-PL policy: Accept only information that has been asserted after January 1st, 2006 by analysts who achieved a StarMine score above 80.

```
FunctionCall       ::=  RDFrelatedFunction | CastingOrExtensionFunction
CastingOrExtensionFunction  ::=  URIref ArgList
ArgList            ::=  ( '(' NIL | Expression ( ',' Expression )* ')' )
RDFrelatedFunction ::=  'str' '(' Expression ')' |
                        'lang' '(' Expression ')' |
                        'datatype' '(' Expression ')' |
                        'isUri' '(' Expression ')' |
                        'isBlank' '(' Expression ')' |
                        'isLiteral' '(' Expression ')' |
                        'regex' '(' Expression ',' Expression (
                        ',' Expression )? ')'
```

Figure 2.11: Grammar for WIQA-PL function calls.

## 2.6 Functions

Filter expressions may include function calls. A function takes some number of RDF terms as arguments and returns an RDF term or a boolean value as result. Figure 2.11 shows the grammar for invoking functions within filter clauses. There are three types of functions in WIQA-PL:

**Basic RDF-Related Functions** are used to test RDF-specific properties of variable bindings. RDF-related functions can, for instance, be used to check if a variable is bound to a URI reference or to a literal, or to test if a literal has a specific language tag. WIQA-PL provides the same RDF-related functions as the SPARQL query language. Table 2.5 contains a short description of each function. The functions are specified in detail in chapter 11.4 of the SPARQL specification [PS05]. The policy suite that is shown in figure 2.12 contains two policies that use RDF-related functions: The policy "Accept only German or English information" uses the `lang()` function to check whether the language tag of RDF literals has the value DE or EN. The policy "Accept only information from Deutsche Bank" uses the the `str()` and the `regex()` functions to check whether the URI that identifies an authority contains the domain name `deutsche-bank.de`.

**Constructor Functions** are used to cast literals to a specific datatype. Casting is performed by calling a constructor function for the target type on an operand of the source type. WIQA-PL provides the same constructor functions as the SPARQL query language. Table 2.6 gives an overview about these functions. The functions are defined in detail in chapter 11.5 of the SPARQL specification [PS05]. Each constructor function can cast only a specific set of datatypes into the target type. For instance, a `xsd:float` cannot be casted into a `xsd:dateTime`. The datatypes that are allowed for the operand of each constructor function are also specified in chapter 11.5 of the SPARQL specification [PS05]. Calling a constructor function with an operand that has a disallowed datatype raises an error. Constructor functions are used within filter clauses to make literals that have different datatypes comparable. For instance, the filter clause `FILTER (xsd:dateTime(?date) > "2005-11-20T17:22:10"^^`xsd:dateTime) uses the constructor function `xsd:dateTime()` to cast the value of the variable `?date` to the datatype `xsd:dateTime` before comparing it to the given value.

**Extension Functions.** WIQA-PL provides an extension mechanism for invoking arbitrary, application domain specific functions. Extension functions are implemented as plug-ins for the WIQA filtering and explanation engine. An extension function is named by a URI and takes some number of RDF terms as arguments. The result of an extension function is an RDF term. Extension functions are called within policies by their URI followed by a list of arguments. The list of arguments is enclosed with parentheses and arguments are separated by commas.

Quality-based information filtering policies rely on a wide range of different, application domain specific assessment metrics. For instance, rating-

| *Function* | *Description* |
|---|---|
| isUri() | Returns true if the argument is a URI. Returns false otherwise. |
| isBlank() | Returns true if the argument is a blank node. |
| isLiteral() | Returns true if the argument is a literal. |
| lang() | Returns the language tag of a literal, if it has one. |
| datatype() | Returns the datatype URI of a literal. |
| str() | Returns an string representation of a URI reference. |
| regex() | Invokes the Xpath [CD99] regular expression function to match a string against a regular expression. |

Table 2.5: Basic RDF-related functions.

```
NAME "Only German or English information"
DESCRIPTION "Accept only German or English information.
      The language is determined by testing the RDF language tag."
PATTERN
      { FILTER( lang(?OBJ) = 'DE' || lang(?OBJ) = 'EN' ) . }

NAME "Accept only information from Deutsche Bank"
DESCRIPTION "Checks if information has been asserted by an
             authority identified with a email address within
             the domain 'deutsche-bank.de'."
PATTERN {
      GRAPH ANY {
        ?GRAPH swp:assertedBy ?warrant .
        ?warrant swp:assertedBy ?authority .
        FILTER(regex(str(?authority), 'deutsche-bank\.de')) . } }
```

Figure 2.12: WIQA-PL policy suite containing two policies using WIQA-PL build in functions.

based filtering policies use various scoring algorithms to calculate the score for an entity from a network of ratings. Content-based filtering policies may rely on linguistic methods to analyze text or may use various statistical methods to compare a piece of information with related information. By including domain specific functions, the WIQA framework can be extended to fit the requirements of different application domains.

Three example extension functions have been implemented so far: The More Positive Ratings and the Tidal Trust functions implement different rating-based scoring algorithms; the `wiqa:count` function allows the formulation of quantity constraints. The functions are named with URIs in the namespace `http://www.wiwiss.fu-berlin.de/suhl/bizer/WIQA/`, which is abbre-

| *Operator* | *Description* |
|---|---|
| xsd:boolean() | Produces a typed literal with the datatype `xsd:boolean` from the operand. |
| xsd:double() | Produces a literal with the datatype `xsd:double`. |
| xsd:float() | Produces a literal with the datatype `xsd:float`. |
| xsd:decimal() | Produces a literal with the datatype `xsd:decimal`. |
| xsd:integer() | Produces a literal with the datatype `xsd:integer`. |
| xsd:dateTime() | Produces a literal with the datatype `xsd:dateTime`. |
| xsd:string() | Produces a literal with the datatype `xsd:string`. |

Table 2.6: Constructor functions.

viated using the `wiqa:` prefix. The extension functions will be described in the following sections.

## 2.6.1 More Positive Ratings Function

The More Positve Ratings extension function implements a simple rating-based scoring algorithm. The function counts all positive and negative ratings for a resource within the graph set to be filtered. It returns true, if the resource received more positive than negative ratings, and returns false otherwise.

The function assumes that ratings are expressed using the terms `fin:positiveRating` and `fin:negativeRating` from the financial vocabulary introduced in section **??**. The function has one operand that determines the resource for which the ratings are counted.

Figure 2.13 shows the WIQA-PL policy "Only accept information from information providers who have received more positive than negative ratings". The filter clause in line 8 uses the `wiqa:MorePositiveRatings` function to check whether an `?authority` has received more positive than negative ratings.

An advantage of the `wiqa:MorePositiveRatings()` function is that the evaluation process is easy to understand for the information consumer. A disadvantage of the function is that it is very susceptible to ballot stuffing and bad mouthing attacks (see section **??**), as it takes all ratings into account and does not differentiate between ratings from trustworthy and less trustworthy raters.

## 2.6.2 Tidal Trust Function

The Tidal Trust extension function implements a more complex rating-based scoring algorithm. The Tidal Trust algorithm was developed by Jennifer

```
NAME "More positive Ratings"
DESCRIPTION "Only accept information from information providers who
             have received more positive than negative ratings."
PATTERN
     {  GRAPH fd:GraphFromAggregator
             { ?GRAPH swp:assertedBy ?warrant .
               ?warrant swp:authority ?authority .
               FILTER wiqa:MorePositiveRatings(?authority) . }
    }
```

Figure 2.13: WIQA-PL policy: Only accept information from information providers who have received more positive than negative ratings.

Golbeck at the University of Maryland [Jen05]. The algorithm takes only ratings from information providers into account who are on the information consumer's web-of-trust. Ratings from other information providers are ignored. The ratings are weighted with the degree of trust the information consumer has in the information provider. The Tidal Trust algorithm is therefore more robust against ballot stuffing and bad mouthing attacks than the algorithm described in the last section. By weighting the ratings, the algorithm can take the personal bias of the information consumer into account and is therefore suitable for situations where ratings are subjective [GM06].

The algorithm operates on a network of ratings in which each node has rated several other nodes. The meaning of the ratings may differ between application scenarios. Within the financial information integration scenario from chapter **??**, a rating may, for instance, represent an investor's opinion about the quality of discussion forum postings from another investor. Figure 2.14 shows an example rating network. The nodes represent information providers, the edges represent ratings on a scale from 1 (low quality) to 10 (high quality).

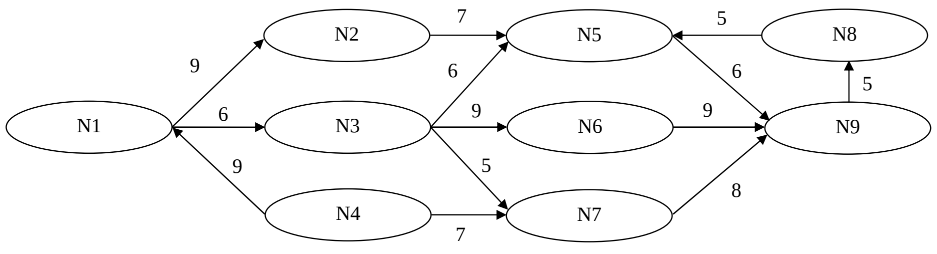

Figure 2.14: Example rating network.

The Tidal Trust algorithm determines the rating of a node in the network (called *sink*) from the perspective of another node (called *source*). If the rating network contains a direct rating by the source for the sink, then the algorithm returns this rating as result. If the network does not contain such a rating, the algorithm infers an approximated rating from the paths connecting the two nodes.

The inference is based on two assumptions: It is expected that people who the user rates highly will tend to agree with the user more about the rating of others than people whom the user gives a low rating. Second, the accuracy of inferred ratings is expected to decrease with length of the paths that connect two individuals. These assumptions are motivated in [GM06] with the experience from several real-world rating networks.

Based on these assumptions, the Tidal Trust algorithm infers a missing rating by conducting the following steps:

1. It searches for all minimum length paths in the network that connect the source with the sink. The length of a path is understood as the number of edges that form a path. Let's assume, for example, that node N1 wants to infer a rating for node N9 from the example network shown in figure 2.14. There are four minimal length paths from N1 to N9: The first path connects the source with the sink over the nodes N2 and N5; the second path over the nodes N3 and N5; the third path over the nodes N3 and N6 and the forth path over nodes N3 and N7.

2. The algorithm determines the *strength* of each path. The strength of a path equals the lowest rating on the path. Within our example, the strength of paths 1-3 is 6, the strength of the fourth path is 5.

3. Afterwards, the algorithm establishes the threshold *max*. This threshold is used to determine which ratings are taken into account in the final calculation. The threshold *max* is set to the maximum strength of all minimal length paths leading to the sink. Within our example, *max* has the value 6.

4. With the *max* value established, each node on a path which does not have a rating for the sink can calculate the rating as the weighted average of the ratings for the sink from its successors using the formula shown in figure 2.15. $t_{ij}$ stands for the rating of a node $i$ for node $j$. The function $suc(i)$ returns all nodes that are successors of node i. The formula takes only ratings from nodes into account which are rated at or above the max threshold by their predecessor. Each rating is weighted by the rating a successor received from its predecessor. Within our example network, nodes N5, N6 and N7 already have ratings for the

$$t_{is} = \frac{\displaystyle\sum_{j \in suc(i) \mid t_{ij} \geq max} t_{ij}\, t_{js}}{\displaystyle\sum_{j \in suc(i) \mid t_{ij} \geq max} t_{ij}}$$

Figure 2.15: Formula for inferring the rating for the sink node s from the perspective of the source node i.

sink N9. Node N3 infers its rating for the sink from the ratings of nodes N5 and N6. It does not take the rating from node N7 into account as its rating for node N7 is below the max threshold of 6. Formula 2.15 results in the value 7.8 for the ratings of N5 and N6 and the ratings of N3 for both nodes. Node N2 infers the rating 6 from the rating of node N3. The source node N1 infers a rating of 6.72 for the sink from the ratings of nodes N2 and N3.

The WIQA implementation of the Tidal Trust algorithm assumes that ratings are represented using the FOAF Trust Module [Jen05]. Ratings have to be contained in the graph set to be filtered. The extension function has two arguments: The first argument identifies the source node; the second argument identifies the sink node. Figure 2.16 shows a WIQA-PL policy that uses the Tidal Trust extension function. The policy accepts information only from information providers who have a Tidal Trust rating that is greater than 5.

```
NAME "TidalTrust rating above 5"
DESCRIPTION "Only accept information from information providers with
             a Tidal Trust rating above 5."
PATTERN
     {  GRAPH fd:GraphFromAggregator
              { ?GRAPH swp:assertedBy ?warrant .
                ?warrant swp:authority ?authority .
                FILTER (wiqa:TidalTrust(?USER, ?authority) > 5) }
     }
```

Figure 2.16: WIQA-PL policy: Only accept information that originates from information providers with TidalTrust rating above 5.

### 2.6.3 Count Function

Filtering policies may rely on quantity constraints. For example, a policy might require information to be asserted by a number of independent information sources. Other policies might require information sources to have received a certain number of positive ratings; or might accept information only from information providers that are believed to be experts on a specific topic because they have worked for a certain number of projects involving that topic.

The WIQA extension function `wiqa:count()` is used to express quantity constraints within WIQA policies. `wiqa:count()` takes one variable as operand and returns the number of different RDF terms that are bound to this variable within a group of matching solutions. The grouping is defined by the position of the filter clause containing the `wiqa:count()` function in the pattern clause. The filter clause can either be included into a graph pattern or it can be positioned after the graph patterns.

If the filter clause is positioned after the graph patterns, then the solution set is grouped by the variables `?SUBJ`, `?PRED`, and `?OBJ`. A group of solutions is formed by all solutions within the solution set that assign the same values to these variables.

```
1.  NAME "Asserted by two different analysts"
2. DESCRIPTION "Only accept information that has been asserted by
3.             at least two different analysts."
4. PATTERNS
5.      {
6.         GRAPH ANY { ?GRAPH swp:assertedBy ?warrant .
7.                     ?warrant swp:authority ?authority . }
8.
9.         GRAPH ANY { ?authority rdf:type fin:Analyst . }
10.
11.        FILTER (wiqa:count(?authority) >= 2) .
12.     }
```

Figure 2.17: WIQA-PL policy: Only accept information that has been asserted by at least two different analysts.

Figure 2.17 shows the policy “Only accept information that has been asserted by at least two different analysts”. The pattern clause of the policy consists of two graph patterns that are followed by a filter clause using the `wiqa:count()` function (line 11).

Figure 2.18 shows an example graph set consisting of three graphs. `ex:Graph3` contains provenance information about `ex:Graph1` and `ex:Graph2`

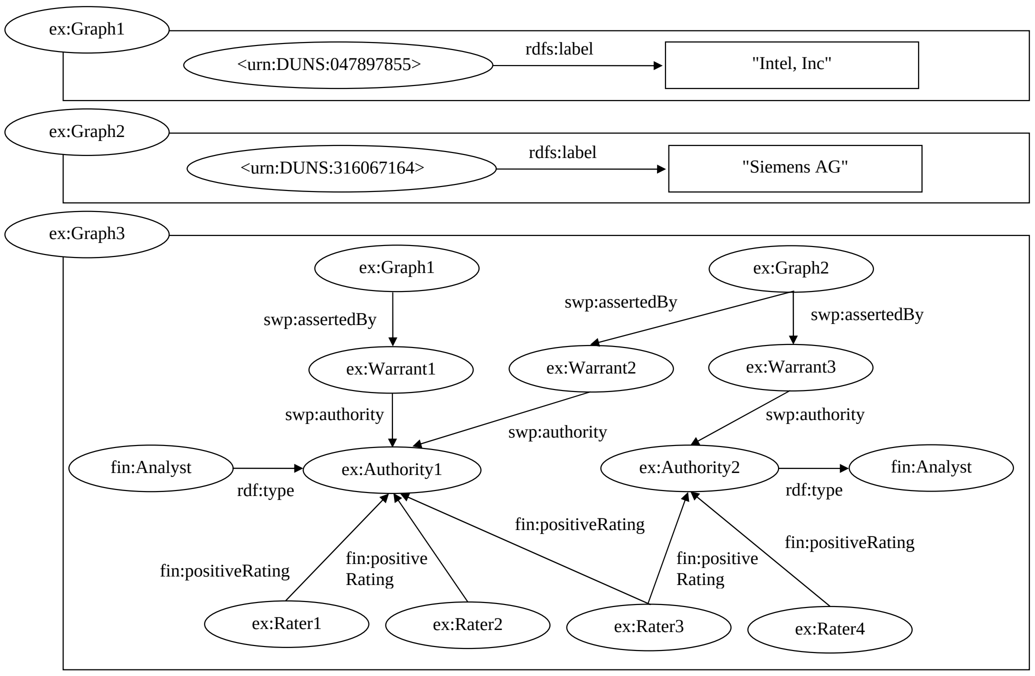

Figure 2.18: Example graph set.

and background information about the authorities that have asserted the graphs: `ex:Authority1` has asserted `ex:Graph1` and `ex:Graph2`. `ex:Authority1` is a `fin:Analyst` and has received three positive ratings. `ex:Authority2` has asserted `ex:Graph2`. `ex:Authority2` is also a `fin:Analyst` but has received only two positive ratings.

When the policy is applied against the example graph set, the WIQA filtering engine conducts the following steps:

1. The filtering engine adds the root pattern `GRAPH ?GRAPH { ?SUBJ, ?PRED, ?OBJ . }` to the set of graph patterns given by the pattern clause. Matching all three patterns against the example graph set results in the solution set shown in table 2.7.

2. As the filter clause, that contains the `wiqa:count()` function, is positioned after the graph patterns, the solution set is grouped by the variables `?SUBJ`, `?PRED` and `?OBJ`. This results in two groups: The first group contains solution 1; the second group contains solution 2 and 3, as these solutions assign the same values to all three variables.

3. The operand of the `wiqa:count()` function is the variable `?authority`.

The function therefore counts the number of different values of this variable in each group. It returns the value 1 for solutions in the first group, and the value 2 for solutions in the second group.

4. The filter clause requires solutions to have a `wiqa:count()` result greater or equal to 2. Solution 1 is therefore removed from the solution set and the accepted triple `<urn:x-DUNS:316067164> rdfs:label "Siemens AG"` is constructed from the remaining second solution set.

| *Variable* | *Solution 1* | *Solution 2* | *Solution 3* |
|---|---|---|---|
| *?SUBJ* | `<urn:x-DUNS: 047897855>` | `<urn:x-DUNS: 16067164>` | `<urn:x-DUNS: 316067164>` |
| *?PRED* | `rdfs:label` | `rdfs:label` | `rdfs:label` |
| *?OBJ* | `"Intel, Inc"` | `"Siemens AG"` | `"Siemens AG"` |
| *?GRAPH* | `ex:Graph1` | `ex:Graph2` | `ex:Graph2` |
| *?warrant* | `ex:Warrant1` | `ex:Warrant2` | `ex:Warrant3` |
| *?authority* | `ex:Authority1` | `ex:Authority1` | `ex:Authority2` |

Table 2.7: Solution set from matching all three patterns against the example graph set.

The `wiqa:count()` function may also be used within graph patterns. Figure 2.19 shows the policy “Only accept information that has been asserted by analysts who have received at least 3 positive ratings”. The policy consists of three graph patterns. The third graph pattern contains the filter clause `FILTER (wiqa:count(?rater) > 2)` (line 11).

```
1.  NAME "Asserted by analysts with at least 3 positive ratings."
2.  DESCRIPTION "Only accept information that has been asserted by
3.               analysts who have received at least 3 positive ratings."
4.  PATTERNS {
5.           GRAPH ANY { ?GRAPH swp:assertedBy ?warrant .
6.                       ?warrant swp:authority ?authority . }
7.
8.           GRAPH ANY { ?authority rdf:type fin:Analyst . }
9.
10.          GRAPH ANY { ?rater fin:positiveRating ?authority .
11.                      FILTER (wiqa:count(?rater) > 2) . }
12.          }
```

Figure 2.19: WIQA-PL policy: Only accept information that has been asserted by analysts who have received at least 3 positive ratings.

The graph patterns in the pattern clause are connected to the root pattern by the referring variables `?GRAPH`, `?SUBJ`, `?PRED`, or `?OBJ`. The graph patterns may also share variables between each other. All variables that occur in more than on graph pattern are called *shared variables*. The first graph pattern (line 5-6) in figure 2.19 refers to the root pattern by using the variable `?GRAPH`. All three graph patterns share the variable `?authority`.

Graph patterns form a pattern tree by sharing variables. Figure 2.20 shows the pattern tree for the policy shown in figure 2.19.

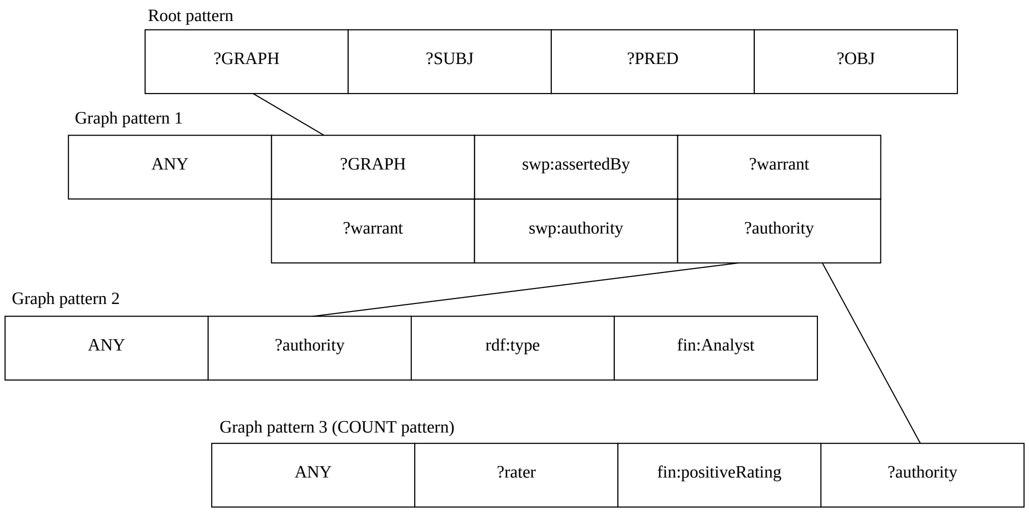

Figure 2.20: Graph pattern tree for the policy: Only accept information that has been asserted by analysts who have received at least 3 positive ratings.

When the `wiqa:count()` function is used within a graph pattern, then the solution set is grouped by the variables `?SUBJ`, `?PRED`, and `?OBJ` and by all shared variables that lie on the path between the root pattern and the graph pattern containing the `wiqa:count()` function.

For our example policy, the shortest path is formed by the variables `?GRAPH` and `?authority`. The solution set is therefore grouped by the variables `?SUBJ`, `?PRED`, `?OBJ`, `?GRAPH`, and `?authority`.

Applying the example policy against the graph set shown in figure 2.18 leads to a solution grouping that consists of three groups: The first group contains all solutions resulting from the triple in `ex:Graph1`, its assertion by `ex:Authority1` and the three ratings of `ex:Authority1`. The second group contains all solutions resulting from the triple in `ex:Graph2`, its assertion by `ex:Authority1` and the three ratings of `ex:Authority1`. The third group contains all solutions resulting from the triple in `ex:Graph2`, its assertion by `ex:Authority2` and the two ratings of `ex:Authority2`.

The function call `wiqa:count(?rater)` returns the number of different values of the variable `?rater` within the group to which a solution belongs. The number is required to be greater than two by the filter clause. `ex:Authority2` was rated by two raters only, therefore all solutions from the third group are removed from the solution set. As `ex:Authority1` was rated by three different raters, the solutions from the first and second group remain in the solutions set, which finally leads to the acceptance of all triples from `ex:Graph1` and `ex:Graph2`.

## 2.7 Summary

This section introduced the WIQA-PL Policy Language and explained how the language is used to express different filtering policies. Within WIQA-PL, filtering policies are expressed as a set of conditions that a piece of information has to satisfy in order to be filtered positive. The language assumes that information is represented as a set of named graphs. Conditions are expressed as graph patterns which refer to the triples to be filtered using a set of referring variables. Graph patterns may contain filter clauses, which further restrict pattern matches. Filter clauses may consist of relational, logical, and numeric expressions and may include function calls. WIQA-PL policies can invoke application domain specific assessment metrics through an extension function mechanism.

The design of the language was lead by the following goals:

**Flexibility.** Quality-based information filtering policies combine a wide range of different context-, content-, and rating-based assessment metrics. A language for expressing policies should therefore provide a high degree of flexibility. WIQA-PL tries to achieve this flexibility by relying on constructs from RDF query languages, such as graph patterns and filters, which have already proven their general applicability.

**Extensibility.** Information quality assessment often requires domain-specific metrics. In order to be applicable across different domains, an information quality assessment framework should be extensible with domain-specific metrics. WIQA-PL provides this extensibility through its extension function mechanism.

**Standard Conformance.** Whenever possible, newly introduced languages should be based on well-known concepts. WIQA-PL adopts the concepts of graph patterns and filters and the syntax for representing them from the SPARQL query language, the standard query language for RDF. This may make it easier for users who are already familiar with SPARQL to learn WIQA-PL.

# Chapter 3

# Explaining Assessment Results

The accuracy of information quality assessment results is often uncertain due to the limited availability of quality indicators and due to the uncertain quality of the quality indicators themselves. Therefore, the user's final decision whether to trust or distrust assessment results depends on her understanding of the assessment metrics and quality indicators that were used in the assessment process. Information systems can support users in this trust decision by providing explanations *why* information satisfies a given filtering policy.

Making information filtering decisions comprehensible and traceable requires diverse forms of explanations. The content of suitable explanations depends on the assessment metrics that are used within a policy and on the current task of the user. For less important tasks, the user will be contented with short, simple to comprehend explanations. For others, more important tasks the user will require explanations to contain detailed information about the assessment process and the quality indicators that were used in the process.

**Explanations for Rating-Based Metrics.** The accuracy of rating-based assessment metrics depends on the the quality of the ratings that are used in the calculation as well as on the scoring algorithm. Ratings might be subjective and raters may try to influence rating-systems by providing unfair ratings. Therefore, explanations for rating-based assessment metrics should contain the ratings that were used in the evaluation and explain the calculation steps of the scoring algorithm. More detailed explanations might provide provenance information about ratings and background information about raters.

**Explanations for Context-Based Metrics.** Context-based assessment metrics rely on meta-information about the circumstances in which information has been claimed. An explanation for a policy that relies on provenance information should list the information providers. The explanation might also contain additional background information about information providers in order to support information consumers in judging their trustworthiness.

**Explanations for Content-Based Metrics** detail why information content itself satisfies the requirements of an assessment metric. For instance, an explanation for a statistical metric should contain the data that was used in the calculation and describe the calculation process. An explanation for a text analysis method might list relevant keywords and explain how the overall score for a text was calculated.

The WIQA framework can generate explanations why an accepted triple satisfies a WIQA-PL policy. The WIQA framework can produce two types of explanations: Textual and RDF explanations. RDF explanations consist of an RDF graph and may be used by applications for further processing. Textual explanations are aimed at direct human consumption. They consist of natural language text fragments.

In order to provide a high degree of flexibility, the WIQA framework combines two explanation generation mechanisms: First, a template mechanism is used to generate the parts of an explanation which explain why constraints that are expressed as graph patterns are satisfied. Afterwards, the template generated explanation parts are supplemented with custom explanation parts that explain why constraints that are expressed using WIQA extension functions are satisfied.

Figure 3.1 shows a visualization of a textual explanation. The explanation details why a triple fulfills the policy "Accept only information that has been asserted by analysts who have received at least 3 positive ratings". Lines 11-15 explain why Peter Smith is considered to be an analyst. Lines 16-19 explain why Peter Smith satisfies the second part of the policy by listing all raters who have rated him positive.

The content of the "because"-part of the explanation is defined using *explanation templates*. When a user requests an explanation why an accepted triple fulfills the policy, these templates are instantiated with variable bindings from the matching solutions that led to the acceptance of the triple. Chapter 3.1 will explain the template mechanism in detail.

Extension functions may conduct complex calculations and may retrieve additional information from the graph set. In order to make their calculations comprehensible, extension functions can generate custom, function-specific explanations. Chapter 3.2 describes how extension function generated and template generated explanations are combined and discusses the

```
The triple:

  Siemens AG has positive analyst report: "As Siemens agrees
  partnership with Novell unit SUSE ..."

fulfills the policy:

  Accept only information that has been asserted by
  analysts who have received at least 3 positive ratings.

because:

  it was asserted by Peter Smith and
     - Deutsche Bank claims that Peter Smith is an analyst.
     - Financial Times claims that Peter Smith is an analyst.
  Peter Smith has received positive ratings from
     - Mark Scott who works for Siemens.
     - David Brown who works for Intel.
     - John Maynard who works for Financial Times.

```

Figure 3.1: Example explanation.

custom explanations that are generated by the `wiqa:MorePositiveRatings` and the `wiqa:TidalTrust` extension functions.

## 3.1 Explaining Pattern Matches

WIQA-PL uses a template mechanism to define the content and the structure of explanations. When a user requests an explanation why an accepted triple fulfills the policy, the explanation templates are instantiated with variable bindings from the matching solutions that led to the acceptance of the triple. This section describes the WIQA explanation template mechanism. As a running example, it is explained how the example explanation shown in figure 3.1 is generated.

Technically, WIQA textual explanations consist of a set of explanation parts. Each explanation part is a tuple $(fragments, children, details)$, where $fragments$ is an ordered list of RDF nodes (usually literals). $fragments$ is created by instantiating an explanation template. *children* is a set of explanation parts which are displayed as children of the current explanation part. *details* may contain an explanation part which contains additional details about the content of $fragments$. As the *details* part may have child parts of its own, this mechanism allows explanations to be divided into different levels of abstraction.

The content of $fragments$ is specified by an *explanation template*. Explanation templates are defined within the pattern clause of a WIQA-PL policy. Figure 3.2 shows the WIQA-PL grammar for defining explanation templates. An explanation template consists of an ordered list of literals, variables, and `ExtensionFunctionURI`s. A template is instantiated by replacing the variables within the template with their values from the set of matching solutions that led to the acceptance of the triple.

The structure of an explanation is determined by the position of the explanation templates within the pattern clause of a WIQA-PL policy.

```
1. ExplanationClause    ::= 'EXPL' ExplanationTemplate '.'
2. ExplanationTemplate  ::= ( Literal | Variable | ExtensionFunctionURI )+
```

Figure 3.2: EBNF grammar of the WIQA-PL explanation clause.

Figure 3.3 shows the WIQA-PL policy "Accept only information that has been asserted by analysts who have received at least 3 positive ratings". The pattern clause contains the four explanation templates (lines 9, 17-18, 23, 27) which were used to generate the example explanation shown in figure 3.1. A WIQA policy suite containing explanation templates for all example policies from chapter 2 is available on the WIQA website[1].

A WIQA explanation is generated in a two step process: First, graph patterns and explanation templates are arranged into a graph pattern tree. Afterwards, the matching solutions that led to the acceptance of the triple are matched against the explanation templates. This two-step process is neccessary to arrange the table-like solution set into a tree-like explanation. Both steps are described below.

## 3.1.1 Building the Graph Pattern Tree

Graph patterns share variables and contain variables that refer to the `GRAPH ?GRAPH {?SUBJ ?PRED ?OBJ}` root pattern which is added in the matching process to the set of graph patterns given by the policy (see section 2.3). For example, the graph pattern in lines 6-9 of figure 3.3 shares the variable `?GRAPH` with the root pattern and the variable `?authority` with the pattern in lines 11-12 and the pattern in lines 20-23.

The structure of the graph pattern tree is determined by the relations of graph patterns to each other through shared variables. Starting from the

[1]http://www.wiwiss.fu-berlin.de/suhl/bizer/wiqa/financialscenario/FinancialScenarioPolicies.wiqa (retrieved 09/25/2006)

```
NAME "Asserted by analysts with at least 3 positive ratings."
DESCRIPTION "Accept only information that has been asserted by
         analysts who have received at least 3 positive ratings."
PATTERNS {

  GRAPH fd:GraphFromAggregator
        { ?GRAPH swp:assertedBy ?warrant .
          ?warrant swp:authority ?authority .
          EXPL "it was asserted by " ?authority " and " . }

  GRAPH ?graph2
        { ?authority rdf:type fin:Analyst . }

  GRAPH fd:GraphFromAggregator
        { ?graph2 swp:assertedBy ?warrant2 .
          ?warrant2 swp:authority ?authority2 .
          EXPL ?authority2 " claims that " ?authority
               " is an analyst." . }

  GRAPH ANY
        { ?rater fin:positiveRating ?authority .
          FILTER (wiqa:count(?rater) > 2) .
          EXPL ?authority "has received positive ratings from" . }

  GRAPH fd:BackgroundInformation
        { ?rater fin:affiliation ?company .
          EXPL ?rater "who works for" ?company . }
  }
```

Figure 3.3: WIQA-PL policy including explanation templates.

root pattern, graph patterns are arranged into a graph pattern tree by the following rule: A graph pattern $B$ becomes a child of another graph pattern $A$ if both patterns share at least one variable, but not if a variable is already shared by pattern $A$ and its parent or between two ancestors of pattern $A$. The second part of the rule assures that multiple graph patterns that share the same variable with a single parent pattern only become children of this pattern and do not appear a second time in the tree as children of each other.

Applying this rule to the graph pattern set given by our example policy results in the graph pattern tree shown in figure 3.4. The root pattern shares the variable `?GRAPH` with `Graph Pattern 1`. `Graph Pattern 1` shares the variable `?authority` with the `Graph Pattern 2` and `Graph Pattern 4`. `Graph Pattern 2` shares the variable `?graph2` with `Graph Pattern 3`, and `Graph Pattern 4` the variable `?rater` with `Graph Pattern 5`. `Graph Pattern 4` is not a child pattern of `Graph Pattern 2` as the variable `?authority` is already shared between pattern

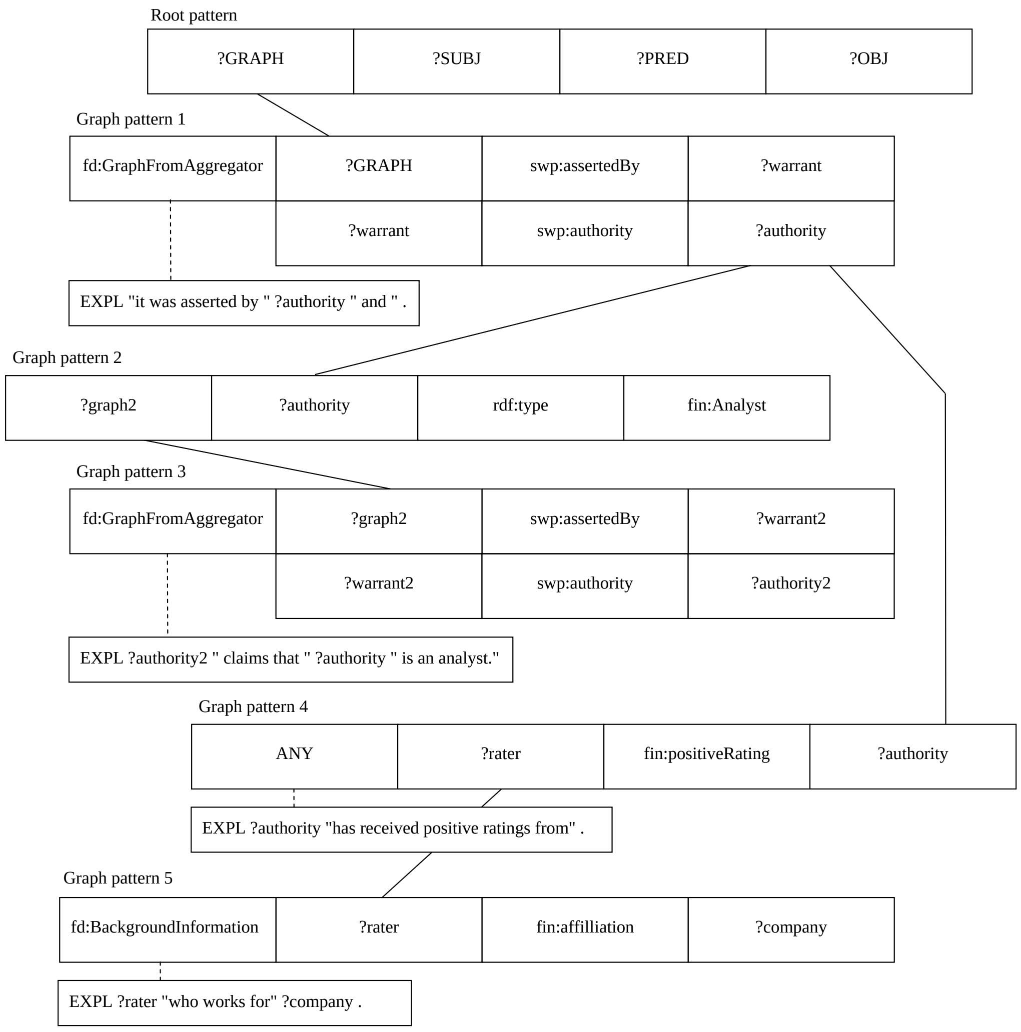

Figure 3.4: Graph pattern tree with attached explanation templates.

one and two.

### 3.1.2 Instantiating the Graph Pattern Tree

When a user requests an explanation why a triple satisfies a given policy, then the explanation templates in the graph pattern tree are instantiated with variable bindings from the set of matching solutions that led to the acceptance of the triple. This section describes the algorithm for creating an explanation for a given triple from a graph pattern tree and the solution set that led to the acceptance of the triple. Section 2.3 explains how the solution set that leads to the acceptance of a triple is determined.

Let $p$ be the policy that was applied to filter the graph set $GS$. Let $tree$ be the graph pattern tree for policy $p$, and let $root$ be the root pattern of $tree$. Let $t$ be an accepted triple from $GS$. Let $S$ be the set of matching solutions which led to the acceptance of the triple $t$.

Figure 3.5 shows the algorithm that is used by the WIQA engine for creating explanations. `explain(S, root, tree)` creates a set of explanation parts why $t$ matches $p$. The condition in line 6 of the algorithm checks whether $gp$ has an attached explanation template $tpl$. If $gp$ has an explanation template, then the set $V$ of all variables that are contained in $tpl$ is determined. Lines 8-11 determine all distinct sets of bindings of these variables in the solution set $S$. A binding is a tuple $(variablename, variablevalue)$. The function `projection(s, V)` determines the set of bindings of all variables in $V$ from solution $s$. The function `instantiate(tpl, bindingset)` in line 13 instantiates $tpl$ with a $bindingset$ from $bindingsets$ by replacing all variables in $tpl$ with their values from $bindingset$.

Lines 14-18 create the child parts of the current explanation part. Line 14 determines all solutions in the solution set $S$ that assign the same values to the variables in $V$ as the current $bindingset$. The function `children(gp, tree)` determines the set of child graph patterns of $gp$ from the graph pattern tree $tree$. Lines 16-18 recursively call `explain()` for each child pattern $c$. The resulting explanation parts are added to the set $childparts$. Finally, in line 19, a new explanation part, consisting of the template instance $fragments$ and the set $childparts$, is created and added to the set $parts$.

Lines 22-24 are executed if $gp$ does not have an attached explanation template. For each child graph pattern of $gp$, the function `explain()` is recursively called and the resulting set of explanation parts is added to to the set $parts$.

Note that the function `instantiate(tpl, bindingset)` replaces the variables in $tpl$ with their values from $bindingset$ but does not replace variable values that are URIs with the name or label of the resource that is identified by the URI. URIs are not replaced with labels in order to enable applications to display custom labels for resources and to provide functionality for retrieving background information about resources.

## 3.2 Explaining Extension Function Results

The extension function mechanism introduced in chapter 2.6 enables the WIQA framework to be extended with domain-specific assessment metrics.

```
explain(S, gp, tree)
  input: S, a solution set
  input: gp, a graph pattern
  input: tree, a graph pattern tree
  parts = {}
  if gp has an explanation template tpl
    V = set of all variables in tpl
    bindingsets = {}
    for each s in S
      bindingsets = bindingsets U { projection(s, V) }
    end for
    for each bindingset in bindingsets
      fragments = instantiate(tpl, bindingset)
      solutiongroup = { s in S where bindingset is subset of s }
      childparts = {}
      for each c in children(gp, tree)
        childparts = childparts U explain(solutiongroup, c, tree)
      end for
      parts = parts U { (fragments, childparts) }
    end for
  else
    for each c in children(gp, tree)
      parts = parts U explain(S, c, tree)
    end for
  end if
  return parts
```

Figure 3.5: Algorithm for generating an explanation from a graph pattern tree and the solution set that led to the acceptance of a triple.

Making the results of extension functions comprehensible often requires extensive, function-specific explanations. For instance, an explanation for a rating-based extension function should contain a description of the scoring algorithm and should list all ratings that were used in the calculation.

The template mechanism described in the last section is suitable for explaining graph pattern matches, but is often too limited for explaining extension function results. Therefore, WIQA-PL allows template generated explanations to be supplemented with custom, function-specific explanation parts. Function-specific explanations are generated directly by the extension function plug-ins. Each plug-in that wants to provide function-specific explanations has to implement the `returnExplanation()` method (see section 4.2.1). The returned function-specific explanation has to consist of a tree of explanation parts.

This chapter describes how function-specific explanation parts are combined with template generated parts. Afterwards, the custom explanations that are generated by the `wiqa:MorePositiveRatings` and `wiqa:TidalTrust` extension functions are discussed.

Extension function generated explanation trees are included as branches into the template generated explanation tree. The position, where function generated explanation trees are included, is specified by URI references to extension functions in the explanation template. Figure 3.6 shows the WIQA-PL policy "Only accept information from information providers who have received more positive than negative ratings". The policy uses the `wiqa:MorePositiveRatings` extension function (line 10). Line 5-6 contain an explanation template. The template contains the reference `wiqa:MorePositiveRatings`. Each time the template is instantiated, the `returnExplanation()` method of the `wiqa:MorePositiveRatings` extension function plug-in is called with the values of the current matching solution. The resulting explanation tree is added to the explanation part that is generated by the template.

```
1. NAME "More positive ratings"
2. DESCRIPTION "Only accept information from information providers who
3.             have received more positive than negative ratings."
4. PATTERN
5.      {  EXPL "The information was asserted by "
6.              ?authority " and " wiqa:morePositiveRatings .
7.         GRAPH fd:GraphFromAggregator
8.               { ?GRAPH swp:assertedBy ?warrant .
9.                 ?warrant swp:authority ?authority .
10.                FILTER wiqa:morePositiveRatings(?authority) . }
11.     }
12.
```

Figure 3.6: WIQA-PL policy: Only accept information from information providers who have received more positive than negative ratings.

### 3.2.1 More Positive Ratings Function

The `wiqa:MorePositiveRatings` extension function, introduced in chapter 2.6.1, implements a simple rating-based scoring algorithm. The function returns true if the graph set contains more positive than negative ratings for a specific resource, and returns false otherwise.

In order to understand the results of the `wiqa:MorePositiveRatings` function and to be able to assess its accuracy, a user requires to know the sums of positive and negative ratings and the origin of the ratings. Therefore, the `wiqa:MorePositiveRatings` function generates function-specific explanations that consist of three parts: The first part contains the sums of ratings. The second and third part list the information providers that rated the resource positive or negative.

Figure 3.7 shows a visualization of an explanation why a triple satisfies the "More positive ratings" policy shown in figure 3.6. Line 13 of the explanation is generated by the explanation template from the policy. Line 14-25 are generated by the extension function plug-in. The explanation uses the *details* mechanism to divide the explanation into two different levels of detail: Line 15-16 sum up the positive and negative ratings. Detail 1 (line 18-21) lists all positive ratings. Detail 2 (line 23-25) lists all negative ratings.

An application which displays explanations to the user might only show the main part of an explanation by default, and display the details only if they are explicitly requested by the user.

```
1.  The triple:
2.
3.   Siemens AG has positive analyst report: "As Siemens agrees
4.   partnership with Novell unit SUSE ..."
5.
6.  fulfills the policy:
7.
8.   Only accept information from information providers who
9.   have received more positive than negative ratings.
10.
11. because:
12.
13.  The information was asserted by Peter Smith and
14.  Peter Smith received the following numbers of ratings:
15.        - 3 positive ratings (see detail 1)
16.        - 2 negative ratings (see detail 2)
17.
18.  Detail 1: Peter Smith received positive ratings from:
19.        - John Reynolds
20.        - Mary O'Conner
21.        - Elisa Armstoen
22.
23.  Detail 2: Peter Smith received negative ratings from:
24.        - Dave Berser
25.        - Colin Marwick
26.
```

Figure 3.7: Explanation why a triple matches the policy: Only accept information from information providers who have received more positive than negative ratings.

### 3.2.2 Tidal Trust Function

The `wiqa:TidalTrust` extension function, introduced in chapter 2.6.2, implements a rating-based scoring algorithm that takes only ratings from information providers into account who are on the information consumer's web-of-trust [Jen05]. The algorithm operates on a network of ratings in which each node has rated several other nodes. The score for a resource is calculated in a three step process: First, the algorithm determines all minimum length paths in the network between the information consumer (source node) and the resource (sink node). Then, the threshold *max* is set to the maximum strength of these paths. Afterwards, each node on the paths calculates its rating for the sink. A rating is calculated by taking the weighted average of all ratings for the sink from the successors of a node, that are rated above the threshold *max* by the node. Each rating is weighted with the rating of the node for its successor (see formula 2.15 in section 2.6.2).

A custom explanation for the `wiqa:TidalTrust` extension function therefore has to explain which ratings were taken into account and describe how the rating for the sink was calculated from these ratings.

The `wiqa:TidalTrust` extension function generates explanations consisting of a summary and three blocks of details. Figure 3.8 show an example explanation for the policy "Only accept information from information providers with a Tidal Trust rating above 5". Lines 13-15 sum up the calculation result. Detail number 1 (lines 19-24) gives an overview about the calculation process. Detail number 2 (lines 26-31) lists all minimum length path from the source to the sink. Detail number 3 (lines 33-46) explains how each score is calculated.

## 3.3 RDF Explanations

Beside of displaying explanations to the end-user, applications might require to process explanations in other application-specific ways. For instance, applications could use explanations as input for reasoning processes, perform different actions depending on the content of an explanation, or attach explanations as evidence to information that is exchanged with other applications [MdS03].

In order to support these use cases, the WIQA framework can generate pure RDF explanations. An RDF explanation consist of an RDF graph. The content of the graph is specified using a *construct template*. Figure 3.9 shows the grammar for defining construct templates within WIQA-PL policies. A construct template is introduced why the keywords CONSTRUCT

EXPLANATION and consists of a set of *construct triple patterns*.

An RDF explanation is generated by taking each matching solution in the solution set that led to the acceptance of the triple, substituting each variable in the construct template with its value from the matching solution and combining the resulting triples into a single RDF graph.

Figure 3.10 shows a WIQA-PL policy containing a construct template in lines 11-14. The construct template consists of two construct triple patterns. The template generates an RDF explanation containing the author and the publication date of each graph in the graph set to be filtered that is published after January 1st, 2006. Figure 3.11 shows an example RDF explanation which was generated with the construct template shown in figure 3.10.

```
The Triple:

  Intel Share has discussion forum posting: As we have already seen
  in the past, investing into this company is no good idea.

fulfills the policy:

  Only accept information from information providers with a
  Tidal Trust rating above 5.

because:

   It was asserted by Mark Scott. The WIQA extension function
   Tidal Trust inferred a rating of 6.7 from Chris Bizer for Mark
   Scott (see detail 1).

Details:

Detail 1: The inferred rating arises from the following calculation:
  - The shortest path between Chris Bizer and Mark Scott has length 3.
    There are 4 different paths with that length (see detail 2).
  - The maximum strength of the paths is 6.0. Therefore, ratings
    below 6.0 are ignored.
  - The calculation yielded a result of 6.7 (see detail 3).

Detail 2: Paths from the source to the sink:
  - Chris Bizer -6.0-> Anne Richards -9.0-> John Gevner -9.0->
    Mark Scott (Strength of the path: 6.0)
  - Chris Bizer -9.0-> Siddhartha Kataki -7.0-> Mary Louis -6.0->
    Mark Scott (Strength of the path: 6.0)
  - ...

Detail 3: Chris Bizer -6.7-> Mark Scott was calculated from these
  ratings:
  - Siddhartha Kataki -6.0-> Mark Scott, weighted with Chris Bizer's
    rating of 9.0 for Siddhartha Kataki.
  - Anne Richards -7.8-> Mark Scott, weighted with Chris Bizer's
    rating of 6.0 for Anne Richards.
  - Anne Richards -7.8-> Mark Scott was calculated from these ratings:
    - Mary Louis -6.0-> Mark Scott, weighted with Anne Richards's
      rating of 6.0 for Mary Louis.
    - John Gevner -9.0-> Mark Scott, weighted with Anne Richards's
      rating of 9.0 for John Gevner.
  - ...
  - John Gevner -9.0-> Mark Scott is a direct rating.
  - Mary Louis -6.0-> Mark Scott is a direct rating.
```

Figure 3.8: Explanation why a triple matches the policy: Only accept information from information providers with a Tidal Trust rating above 5 (shortened).

```
RDFExplanationClause ::= 'CONSTRUCT' 'EXPLANATION' ConstructTemplate
ConstructTemplate    ::= '{' ConstructPattern+ '}'
ConstructPattern     ::= URIOrBnodeOrVariableOrReference
                         URIOrBnodeOrVariableOrReference
                         URIOrBnodeOrLiteralOrVariableOrReference '.'
URIOrBnodeOrVariableOrReference
                     ::= URI | Bnode | Variable | Reference
URIOrBnodeOrLiteralOrVariableOrReference
                     ::= URI | Bnode | Variable | Literal | Reference
```

Figure 3.9: EBNF grammar for defining WIQA-PL construct templates.

```
NAME "Information that has been asserted after January 1st, 2006"
DESCRIPTION "Accept only information that has been asserted
             after January 1st, 2006"
PATTERN
      { GRAPH fd:GraphFromAggregator
          { ?GRAPH swp:assertedBy ?warrant .
            ?warrant swp:authority ?authority .
            ?warrant dc:date ?date .
            FILTER (?date > "2006-01-01"^^xsd:date) }
      }
CONSTRUCT EXPLANATION
      {     ?GRAPH dc:creator ?authority .
            ?GRAPH dc:date ?date .
      }
```

Figure 3.10: WIQA-PL policy containing a construct template.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix fd: <http://www.fu-berlin/suhl/bizer/exampleDataset> .

fd:GraphFromPeterSmith dc:creator <mailto:peterSmith@deutsche-bank.de> ;
                       dc:date "2005-11-20T12:40:44"^^xsd:dateTime .

fd:GraphFromMarkScott dc:creator <mailto:mark@scott.com> .
                      dc:date "2005-11-20T17:22:10"^^xsd:dateTime .
```

Figure 3.11: Example RDF explanation generated with the construct template shown in figure 3.10.

# Chapter 4

# Implementation

This chapter describes the implementation and the usage of the WIQA Information Quality Assessment Framework. The WIQA framework is implemented in Java. The implementation consists of two parts: *NG4J - Named Graph API for Jena* which is a general purpose extension to the Jena Semantic Web framework [CDD$^+$04] for handling Named Graphs. Second, the *WIQA Filtering and Explanation Engine* which enables applications to filter a set of named graphs using WIQA-PL policies and to retrieve explanations about filtering decisions.

## 4.1 NG4J - Named Graph API for Jena

NG4J - Named Graph API for Jena [BCW05][BC06] is a software toolkit for creating, manipulating, persisting, and exchanging sets of named graphs.

The toolkit provides an API for manipulating a set of named graphs using graph-centric and quad-centric methods. Graph sets can be stored in memory or in a relational database. NG4J provides parsers and serializers for the TriX and TriG syntaxes introduced in section **??**.

NG4J implements convenience methods for using the Semantic Web Publishing vocabulary (SWP) introduced in chapter **??**. The toolkit enables users to sign graph sets and to verify signatures without the need for detailed knowledge about signature methods and the SWP vocabulary.

NG4J builds on the Jena Semantic Web framework [CDD$^+$04], a leading Semantic Web programming environment which is maintained by the Hewlett Packard Laboratories in Bristol. NG4J is available under the terms of the Berkeley Software Distribution (BSD) license [Reg99] and can be downloaded from the NG4J website[1].

[1] http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/ (retrieved 09/25/2006)

The followings sections give an overview about NG4J's public interface and illustrate the usage of the toolkit with a code example. The complete documentation of the toolkit is available on the NG4J website[2].

### 4.1.1 Public Interface

The UML diagram shown in figure 4.1 gives an overview about NG4J's public interface.

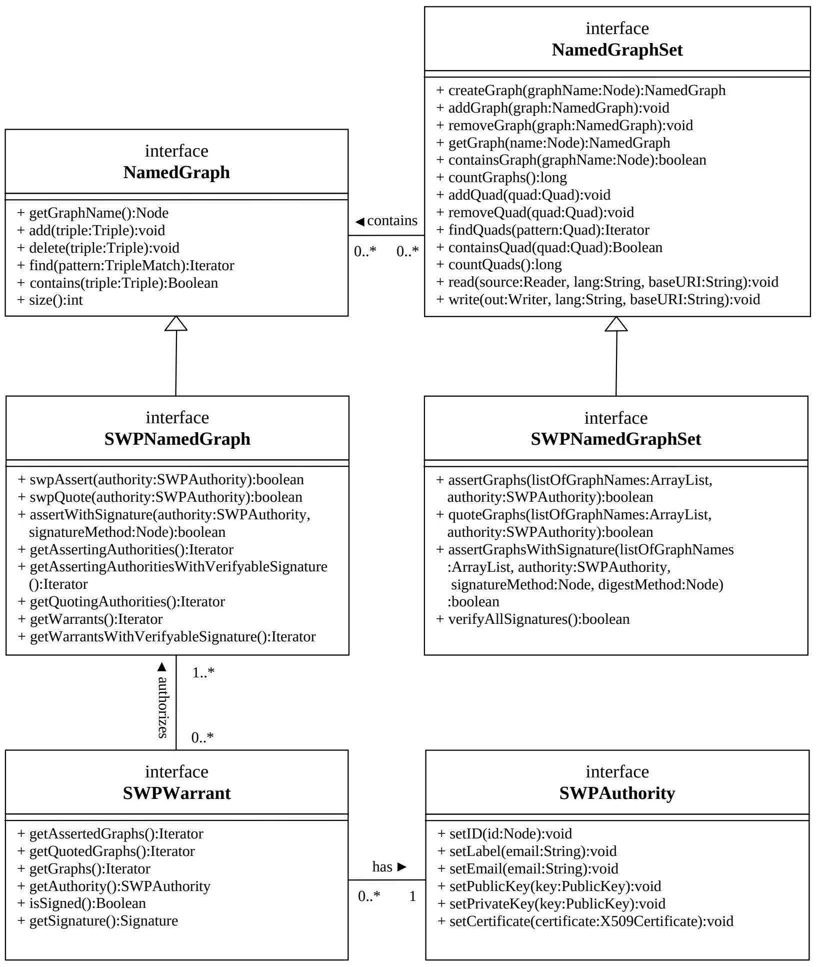
interface
NamedGraphSet
+ createGraph(graphName:Node):NamedGraph
+ addGraph(graph:NamedGraph):void
+ removeGraph(graph:NamedGraph):void
+ getGraph(name:Node):NamedGraph
+ containsGraph(graphName:Node):boolean
+ countGraphs():long
+ addQuad(quad:Quad):void
+ removeQuad(quad:Quad):void
+ findQuads(pattern:Quad):Iterator
+ containsQuad(quad:Quad):Boolean
+ countQuads():long
+ read(source:Reader, lang:String, baseURI:String):void
+ write(out:Writer, lang:String, baseURI:String):void
interface
NamedGraph
+ getGraphName():Node
+ add(triple:Triple):void
+ delete(triple:Triple):void
+ find(pattern:TripleMatch):Iterator
+ contains(triple:Triple):Boolean
+ size():int
◄contains
0..*
0..*
interface
SWPNamedGraph
+ swpAssert(authority:SWPAuthority):boolean
+ swpQuote(authority:SWPAuthority):boolean
+ assertWithSignature(authority:SWPAuthority, signatureMethod:Node):boolean
+ getAssertingAuthorities():Iterator
+ getAssertingAuthoritiesWithVerifyableSignature ():Iterator
+ getQuotingAuthorities():Iterator
+ getWarrants():Iterator
+ getWarrantsWithVerifyableSignature():Iterator
interface
SWPNamedGraphSet
+ assertGraphs(listOfGraphNames:ArrayList, authority:SWPAuthority):boolean
+ quoteGraphs(listOfGraphNames:ArrayList, authority:SWPAuthority):boolean
+ assertGraphsWithSignature(listOfGraphNames :ArrayList, authority:SWPAuthority, signatureMethod:Node, digestMethod:Node) :boolean
+ verifyAllSignatures():boolean
authorizes ►
1..*
0..*
interface
SWPWarrant
+ getAssertedGraphs():Iterator
+ getQuotedGraphs():Iterator
+ getGraphs():Iterator
+ getAuthority():SWPAuthority
+ isSigned():Boolean
+ getSignature():Signature
has ►
0..*
1
interface
SWPAuthority
+ setID(id:Node):void
+ setLabel(email:String):void
+ setEmail(email:String):void
+ setPublicKey(key:PublicKey):void
+ setPrivateKey(key:PublicKey):void
+ setCertificate(certificate:X509Certificate):void

Figure 4.1: Overview about NG4J's public interfaces.

[2]http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/javadoc/ (retrieved 09/25/2006)

## NamedGraph and NamedGraphSet

The `NamedGraph` interface defines basic methods for manipulating a named graph. The `add()` and `delete()` methods add and remove triples from the graph. The `find()` method returns an iterator over all triples that match a given triple pattern.

The basic information container in NG4J is the `NamedGraphSet`. The `NamedGraphSet` interface defines methods for manipulating and serializing a set of named graphs. A graph set can be manipulated by adding and removing entire graphs, or by working with individual *quads*. Within NG4J, quads are not understood as separate entities beside named graphs but as a different view on a graph set. The first item in a quad is a graph name; the other three constitute an RDF triple. If a quad, having a graph name that does not exists in a graph set yet, is added to the set, then a new graph with this name is created in the set. If a quad with a graph name that already exists in a graph set is added to the set, then the triple of the quad is added to the corresponding named graph.

The `write()` method serializes the graph set to a file. The `lang` argument defines the serialization syntax. NG4J supports the TRIX, TRIG, RDF/XML, N-TRIPLE, and N3 syntaxes. If the specified serialization syntax does not support graph naming, then the union graph is serialized, and knowledge about the partitioning of triples into graphs is lost.

NG4J provides two implementations of the `NamedGraphSet` and `NamedGraph` interfaces: The classes `NamedGraphSetImpl` and `NamedGraphImpl` store graph sets in memory. The classes `NamedGraphSetDB` and `NamedGraphDB` persist graph sets in a relational database. `NamedGraphSetDB` is instantiated for a given JDBC `Connection`. Database tables for storing graph sets are automatically created by NG4J if they do not exist in the database yet.

## The Semantic Web Publishing API

NG4J provides a convenience API for asserting and quoting graphs using Semantic Web Publishing vocabulary introduced in chapter ??. The API consists of the `SWPNamedGraphSet`, `SWPNamedGraph`, `SWPWarrant` and `SWPAuthority` interfaces.

`SWPAuthority` stores the URI reference, label, email address, public and private key, and certificate of an information provider. Keys and certificates are represented using the `java.security` package. NG4J supports RSA [KS98] and DSA [FIP95b] keys, as well as X.509 certificates [HPFS02].

`SWPWarrant` represents a warrant which may assert or quote several graphs. The methods `getAssertedGraphs()` and `getQuotedGraphs()` return an iterator over all graphs that are asserted or quoted by the warrant. The method `getAuthority()` returns the authority of the warrant. The method `isSigned()` can be used to check whether a warrant is signed.

`SWPNamedGraph` extends the NamedGraph interface. `SWPNamedGraph` defines convenience methods for asserting and signing graphs and for verifying graph signatures.

The method `swpAssert(authority:SWPAuthority)` turns the graph into a self-asserting warrant graph by adding the triples `<GraphName> swp:assertedBy <GraphName>` and `<GraphName> swp:authority <Authority>` to the graph. `<GraphName>` stands for the URI reference identifying the current graph; `<Authority>` stands for the URI reference identifying the authority that is passed as argument to the method.

The method `assertWithSignature()` turns the graph into a self-asserting warrant graph and signs the graph with the private key of the authority. The argument `signatureMethod` specifies the combination of canonicalization, digest, and signature algorithms that should be used to calculate the signature. Signature methods are identified by the URI references listed in table **??**. NG4J supports the RSA [KS98] and DSA [FIP95b] signatures algorithms, SHA1 [FIP95a] and MD5 [Riv92] digests, and the RDF canonicalization algorithm proposed by Carroll in [Car03]. The `assertWithSignature()` method adds a `swp:assertedBy`, a `swp:authority`, a `swp:signatureMethod`, and a `swp:signature` triple to the graph.

The method `getWarrantsWithVerifyableSignature()` returns an iterator over all warrant that assert or quote the graph and are signed with a verifiable signature. For verifying graph signatures, NG4J requires the public keys or certificates of information providers and root certification authorities that are trusted by the current user. NG4J expects trusted keys and certificated to be contained in the graph `<http://localhost/trustedinformation>`, which has to be added to a graph set before calling any of NG4J's signature verification methods. NG4J traces certification chains up to a certificate in this trusted graph. Certification chains are constructed using all certificates that are contained in the graph set.

The `SWPNamedGraphSet` interface extends the NamedGraphSet interface. `SWPNamedGraphSet` defines convenience methods to assert and sign multiple graphs at once and to verify all signatures that are contained in a graph set.

The method `assertGraphsWithSignature()` asserts and signs multiple graphs. The resulting warrant is added as a new graph to the graph set. The method takes a list of graph names, an `SWPAuthority`, a signature method, and digest method as argument. The method adds a new warrant graph to the graph set which asserts all graphs from the list. The new graph is named with a Universally Unique IDentifier (UUID) [LMS05] that is created using the Java package `com.eaio.uuid.UUID` [Bur06]. Then, the method digests all graphs from the list and adds the digests to the new warrant graph. Afterwards, the warrant graph is signed and the signature is added to it.

The method `verifyAllSignatures()` verifies all signatures in the graph set. The method expects trusted keys and certificates to be contained in the graph `<http://localhost/trustedinformation>`. The verification result is added to the graph set as a new graph called `<http://localhost/verifiedSignatures>` The graph contains a `<Warrant> swp:signatureVerification swp:sucessful` triple for each warrant with a verifyable signature. The graph can be used within WIQA-PL filtering policies for checking whether content is digitally signed.

### 4.1.2 Usage Example

The example code shown in figure 4.3 illustrates how NG4J is used to create a graph set, add information to the graph set, retrieve information from the graph set and finally serialize the graph set using the TriX syntax [CS04].

## 4.2 WIQA - Filtering and Explanation Engine

The WIQA - Filtering and Explanation Engine enables applications to filter a set of named graphs using WIQA-PL policies and to retrieve explanations about the filtering decisions. WIQA implementation builds on NG4J and the ARQ SPARQL query engine [Sea06].

The WIQA - Filtering and Explanation Engine is available under the terms of the GNU General Public License [Fre91] and can be downloaded from the WIQA website[3]. The following sections give an overview about the public classes of the engine and illustrate its usage with a code example. The complete documentation of the engine is available on the WIQA website[4].

### 4.2.1 Public Interface

The UML diagram shown in figure 4.1 gives an overview about the public interface of the WIQA Filtering- and Explanation Engine.

#### Policy and PolicyParser

WIQA-PL policies are represented as instances of the class `Policy`. The class `PolicyParser` provides a collection of static methods for parsing WIQA-PL

[3]http://www.wiwiss.fu-berlin.de/suhl/bizer/wiqa/ (retrieved 09/25/2006)
[4]http://www.wiwiss.fu-berlin.de/suhl/bizer/wiqa/javadoc/ (retrieved 09/25/2006)

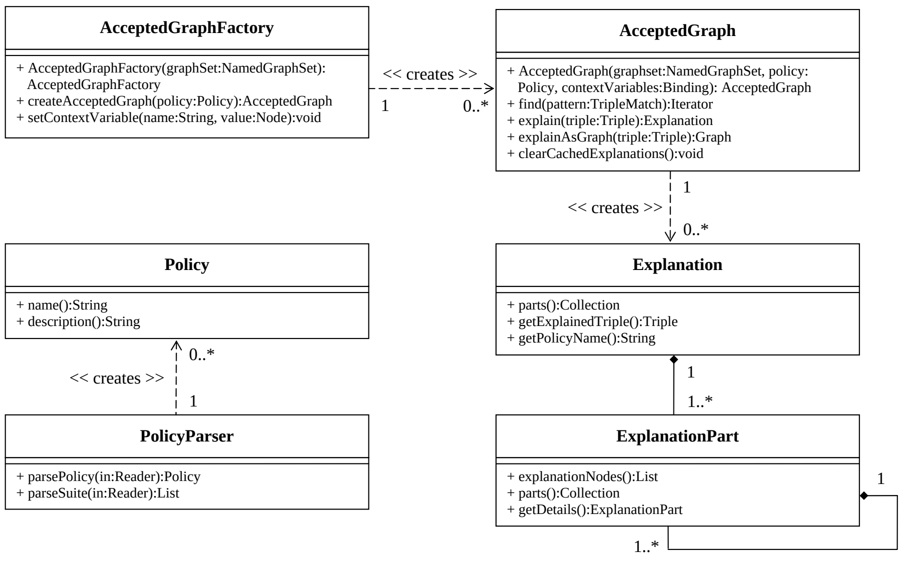
AcceptedGraphFactory
+ AcceptedGraphFactory(graphSet:NamedGraphSet): AcceptedGraphFactory
+ createAcceptedGraph(policy:Policy):AcceptedGraph
+ setContextVariable(name:String, value:Node):void
<< creates >>
1
0..*
AcceptedGraph
+ AcceptedGraph(graphset:NamedGraphSet, policy: Policy, contextVariables:Binding): AcceptedGraph
+ find(pattern:TripleMatch):Iterator
+ explain(triple:Triple):Explanation
+ explainAsGraph(triple:Triple):Graph
+ clearCachedExplanations():void
<< creates >>
1
0..*
Policy
+ name():String
+ description():String
Explanation
+ parts():Collection
+ getExplainedTriple():Triple
+ getPolicyName():String
<< creates >>
0..*
1
1
1..*
PolicyParser
+ parsePolicy(in:Reader):Policy
+ parseSuite(in:Reader):List
ExplanationPart
+ explanationNodes():List
+ parts():Collection
+ getDetails():ExplanationPart
1
1..*

Figure 4.2: Overview about the public interface of the WIQA Filtering- and Explanation Engine.

policies from files and strings into `Policy` objects. The method `parsePolicy()` creates a single `Policy` object from a string or a Java `Reader`. The method `parseSuite()` parses all policies in a policy suite and returns a list of `Policy` objects.

### AcceptedGraph and AcceptedGraphFactory

The class `AcceptedGraph` is the main interface of the WIQA - Filtering and Explanation Engine. An `AcceptedGraph` represents a filtered view on a set of named graphs. Only statements matching a WIQA-PL policy are in the graph. The method `find()` is used to extract information from the graph. The method returns an iterator over all accepted triples that match a given triple pattern.

`AcceptedGraphs` are created using an `AcceptedGraphFactory`. The method `createAcceptedGraph()` of the factory class returns an `AcceptedGraph` for a given `Policy`. The method `setContextVariable()` is used to set WIQA-PL context variables (see section 2.4) which may be used within policies afterwards.

### Retrieving Explanations

`AcceptedGraphs` can generate textual explanations and RDF explanations why a triple was accepted into the graph. A textual `Explanation` is created by calling the method `explain()` of an `AcceptedGraph`. The method takes an accepted triple as argument.

An `Explanation` consists of a collection of `ExplanationParts`. The method `parts()` returns the collection of `ExplanationParts`. An `ExplanationPart` represents a text fragment and has zero or more children which are also `ExplanationParts`. The text fragment is represented as list of RDF nodes. Most of these are literals, but some may be URI references or blank nodes which refer to some entity involved in the explanation. Applications may want to choose to make URI references or blank nodes click-able or retrieve an appropriate label for them. The method `explanationNodes()` returns the list of RDF nodes. The method `parts()` returns a collection containing the child explanation parts of the current part.

An explanation part can provide additional details about itself. These details are represented as a further explanations part. As any explanation part, this part may also have child parts. The method `getDetails()` returns this explanations part, or null if none exists.

Beside of textual explanations, the WIQA engine can also generate RDF explanations (see section 3.3). These are simply RDF graphs that contain information about the filtering process in a vocabulary chosen by the policy. RDF explanations are retrieved by calling the `explainAsGraph()` method of an `AcceptedGraph`.

Textual and RDF explanations are cached inside the `AcceptedGraph`. This consumes memory over time. The `clearCachedExplanations()` method discards all cached explanations. The next call to the `find()` method will start to fill the cache again. The WIQA framework provides an `ExplanationToHTMLRenderer` which can be used to generate basic HTML representations of explanations.

### Implementing and Registering Extension Functions

The WIQA-PL policy language can be extended with domain-specific extension functions (see section 2.6). Extension functions are implemented as plug-ins for the WIQA Filtering and Explanation Engine. At implementation level, two different types of extension functions are distinguished: Basic functions and extensions. The input of a basic function is a single matching solution and a number of arguments which are RDF nodes. The output is a boolean value. The input of an extension is a stream of matching solutions and a number of arguments which are RDF nodes. The output is a modified version of the input stream. The `wiqa:MorePositiveRatings` and the `wiqa:TidalTrust` extension functions are implemented as basic functions. The `wiqa:count` extension function is implemented as an extension as it requires access to the complete solution set in order to count solutions.

Basic functions have to extend the abstract class `ExplainableFunction`. A basic function has to implement the following methods: `exec()` which does the actual calculation, and `returnExplanation()` which returns a custom explanation for the function.

Extensions have to extend the abstract class `ExplainableExtension`. An extension has to implement the `exec()`, `finish()`, and the `returnExplanation()` methods. The function `exec()` is called once for every solution in the solution stream. `finish()` is called after the last solution of the stream. The expected return values of both functions are iterators over matching solutions. An extension can choose when it returns the matching solutions, it might return some for each `exec()` call, or all for the `finish()` call.

Basic functions and extensions have access to the unfiltered graph set and can use it to retrieve additional information, like ratings or background information about information providers. The graph set is accessed from within a basic function or extension by calling the method `getDataset()`.

Basic functions and extensions must be registered with the `FunctionRegistry` and `ExtensionRegistry` respectively, before they can be used in WIQA-PL policies. A function is registered by calling `FunctionRegistry.get().put("functionURI", Implementation.class)`, where `functionURI` is the URI identifying the function and `Implementation.class` is a Java class implementing the function. Extensions are registered by the same call on `ExtensionRegistry`.

### 4.2.2 Usage Example

The example code shown in figure 4.4 illustrates the usage of the WIQA - Filtering and Explanation Engine. Lines 10-11 create a new `NamedGraphSet` and load a TriX file into the graph set. Lines 14-17 load a policy suite and select a policy from the suite. Line 19-22 create an `AcceptedGraphFactory` and assign a value to the context variable `?USER`. Line 25 creates an `AcceptedGraph` by applying the selected policy against the graph set. Lines 28-30 create an iterator over all triples in the accepted graph that have the subject `http://richard.cyganiak.de/foaf.rdf#RC`. In line 37 an explanation why the first triple satisfies the policy is created. This explanation is rendered to HTML and written to `System.Out` in lines 40-42.

```java
import java.util.Iterator;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.Triple;
import de.fuberlin.wiwiss.ng4j.*

// Create a new graphset
NamedGraphSet graphset = new NamedGraphSetImpl();

// Create a new NamedGraph in the NamedGraphSet
NamedGraph graph =
  graphset.createGraph("http://example.org/persons/123");

// Add information to the NamedGraph
graph.add(new Triple(
   Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
   Node.createURI("http://xmlns.com/foaf/0.1/name") ,
   Node.createLiteral("Richard Cyganiak", null, null)));

// Create a quad
Quad quad = new Quad(
  Node.createURI("http://www.bizer.de/InformationAboutRichard"),
  Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
  Node.createURI("http://xmlns.com/foaf/0.1/mbox") ,
  Node.createURI("mailto:richard@cyganiak.de"));

// Add the quad to the graphset. This will create a new NamedGraph
// in the graphset.
graphset.addQuad(quad);

// Find information about Richard across all graphs in the graphset
Iterator it = graphset.findQuads(
   Node.ANY,
   Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
   Node.ANY,
   Node.ANY);

// Output the results of findQuads()
while (it.hasNext()) {
   Quad q = (Quad) it.next();
   System.out.println("Source: " + q.getGraphName());
   System.out.println("Statement: " + q.getTriple());
}

// Serialize the graphset to System.out, using the TriX syntax
graphset.write(System.out, "TRIX", null);

```

Figure 4.3: NG4J - Named Graph API for Jena usage example.

```java
import java.util.Iterator;
import java.util.List;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.Triple;
import de.fuberlin.wiwiss.ng4j.NamedGraphSet;
import de.fuberlin.wiwiss.ng4j.impl.NamedGraphSetImpl;
import de.fuberlin.wiwiss.wiqa.*;

// Create a new graph set and read a TRIG file into the graph set
NamedGraphSet graphset = new NamedGraphSetImpl();
graphset.read("file:graphset.trig", "TRIG");

// Read a WIQA policy suite and get a policy from the policy suite
List policysuite =
   PolicyParser.parseSuiteFromFile("file:policies.wiqa");
Policy policy =
   policysuite.get(Node.createURI("http://example.org/policy1");

// Create a graph factory and set a context variable
AcceptedGraphFactory factory = new AcceptedGraphFactory(graphset);
factory.setContextVariable("USER",
  Node.createURI("http://www.bizer.de/i"));

// Filter the graph set using the policy
AcceptedGraph acceptedGraph = factory.createAcceptedGraph(policy);

// Find information about Richard in the accepted graph
Iterator it = acceptedGraph.find(
   Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
   Node.ANY, Node.ANY);

// Get the first triple from the result
if (it.hasNext()) {
   Triple triple = (Triple) it.next();

   // Get an explanation why the triple fulfills the policy
   Explanation explanation = acceptedGraph.explain(triple);

   //Output a HTML representation of the explanation
   ExplanationToHTMLRenderer renderer =
       new ExplanationToHTMLRenderer(explanation, graphset);
   System.out.println(renderer.getExplanationAsHTML());
}
```

Figure 4.4: WIQA - Filtering and Explanation Engine usage example.

# Chapter 5

# Related Work

This chapter compares the WIQA framework with related approaches.

**Database Views.** Accepted graphs within the WIQA framework can be compared to views in the context of relational databases. WIQA's explanation capabilities relate to work within the database community on explaining data lineage and view generation. An example of an approach to explaining view generation has been developed by Cui and Wisdom in the context of the Stanford University WHIPS data warehousing system [CW00]. For a given data item in a materialized view, the authors propose a linage tracing algorithm to identify the set of source data items that produced the view item. The algorithm is applicable to aggregate-select-project-join views and can be employed by data warehouse analysis tools to provide drill-down functionality from view items to source data items. What distinguishes the WIQA framework from the work within the relational data base community is the underlying data model. By employing a variation of the RDF data model, the WIQA framework is tailored towards the integration of heterogeneous information from the Web. For instance, integrating two partial descriptions of the same object while keeping track of the provenance of different pieces of information can simply be achieved within the Named Graph data model, but is tricky using the relational data model.

**Inference Web.** A related approach to explaining information quality in the context of web-based information systems has been developed by the Inference Web project at the Stanford University Knowledge Systems Laboratory [MdS03]. The project aims at making query answers more transparent by providing explanations about information sources as well as inference processes that are used to derive query results.

The Inference Web infrastructure includes a registry containing details on information sources, reasoners, languages, and rewrite rules; a portable proof specification; and a proof and explanation browser. Inference Web and the WIQA framework assume different application scenarios. While the WIQA framework is tailored towards a simple web-based information integration scenario, Inference Web assumes an agent community consisting of actively reasoning agents that cooperatively derive query answers from shared knowledge. Therefore, Inference Web focuses on explaining distributed reasoning paths [McG96], while the WIQA framework generates explanations why subjective information quality assessment policies are satisfied.

**TRELLIS.** A further system that employs the RDF data model and provides information quality assessment functionality is the TRELLIS information analysis tool [GR02] developed at the University of Southern California. TRELLIS aims at supporting intelligence analysts in selecting quality information within a military setting. As an analyst makes a decision on which sources to dismiss and which to belief, TRELLIS captures the derivation of the decision in a semantic markup. The system then uses these annotations to derive an information quality assessment of the source based on the annotations of many individuals. Compared with the WIQA framework, TRELLIS supports only one fixed ontology for capturing quality-related meta-information and a single hard-coded information quality assessment policy, while the WIQA framework may employ arbitrary ontologies and a wide range of different assessment policies.

**Almendra and Schwabe.** An approach for translating abstract information quality requirements into concrete assessment policies is presented by Almendra and Schwabe from the Pontificia Universidade Catolica do Rio de Janeiro in [AS05]. Their work builds directly on the work presented in this thesis and employs the Named Graph data model, the Semantic Web Publishing Vocabulary and the TriQL.P policy language, an earlier version of the WIQA-PL policy language. In addition to our work, where each policy must explicitly specify all the conditions that triples must fulfill to be accepted, they propose an ontology for expressing information quality requirements (TrustPoint) and an algorithm that automatically derives information quality assessment policies from these requirements by combining TriQL.P query fragments. Given adequate tool support, their translation mechanism provides a valuable extension to our work as it reduces the technical knowledge required from a policy author.

# Bibliography

[AS05] Vinicius Almendra and Daniel Schwabe. Real-world Trust Policies. In *Proceedings of the Semantic Web Policy Workshop*, 2005.

[BC06] Christian Bizer and Richard Cyganiak. NG4J - Named Graphs API for Jena. http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/, 2006. Retrieved 09/25/2006.

[BCW05] Christian Bizer, Richard Cyganiak, and Rowland Watkins. NG4J - Named Graphs API for Jena. In *2nd European Semantic Web Conference - Posters and Demonstrations*, 2005.

[BHL06] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML 1.0 (Second Edition). http://www.w3.org/TR/REC-xml-names/, 2006. Retrieved 09/25/2006.

[Bur06] Johann Burkard. Generate UUIDs (or GUIDs) in Java. http://johannburkard.de/software/uuid/, 2006. Retrieved 09/25/2006.

[Car03] Jeremy Carroll. Signing RDF Graphs. In *Proceedings of the 2nd International Semantic Web Conference*, pages 369–384, 2003.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0 - W3C Recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116, 1999. Retrieved 09/25/2006.

[CDD$^+$04] Jeremy Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Aandy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th World Wide Web Conference*, pages 74–83, 2004.

[CLW02] Ritu Chadha, George Lapiotis, and Steven Wright. Special Issue on Policy-based Networking. *IEEE Network*, 16(2):8–56, 2002.

[CS04] Jeremy Carroll and Patrick Stickler. TriX: RDF Triples in XML. In *Proceedings of the 13th International World Wide Web Conference: Alternate Track Papers and Posters*, pages 412–413, 2004.

[CW00] Yingwei Cui and Jennifer Widom. Practical Lineage Tracing in Data Warehouses. In *Proceedings of the 16th International Conference on Data Engineering*, pages 367–378, 2000.

[FIP95a] FIPS PUB 180-1. Secure Hash Standard. National Institute of Standards and Technology, 1995.

[FIP95b] FIPS PUB 186-2. Digital Signature Standard (DSS). National Institute of Standards and Technology, 1995.

[Fre91] Free Software Foundation Inc. GNU General Public License - Version 2. http://www.gnu.org/licenses/gpl.txt, 1991. Retrieved 09/25/2006.

[FWS05] Babak Firozabadi, William Winsborough, and Akhil Sahai, editors. *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, 2005.

[GM06] Jennifer Golbeck and Aaron Mannes. Using Trust and Provenance for Content Filtering on the Semantic Web. In *Proceedings of the Models of Trust for the Web Workshop*, 2006.

[GR02] Yolanda Gil and Varun Ratnakar. Trusting Information Sources One Citizen at a Time. In *Proceedings of the 1st International Semantic Web Concerence*, pages 162–174, 2002.

[HPFS02] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. http://www.ietf.org/rfc/rfc3280.txt, 2002. Retrieved 09/25/2006.

[ISO96] ISO/IEC 14977:1996. Information technology - Syntactic Metalanguage - Extended BNF. International Organization for Standardization, 1996.

[Jen05] Jennifer Golbeck. *Computing and Applying Trust in Web-based Social Networks*. PhD thesis, University of Maryland, 2005.

[KS98] B. Kaliski and J. Staddon. RFC2437 : PKCS1: RSA Cryptography Specifications Version 2.0. http://www.ietf.org/rfc/rfc2437.txt, 1998. Retrieved 09/25/2006.

[LMS05] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. http://tools.ietf.org/html/4122, 2005. Retrieved 09/25/2006.

[Mar02] Massimo Marchiori. The Platform for Privacy Preferences - W3C Recommendation. http://www.w3.org/TR/P3P/, 2002. Retrieved 09/25/2006.

[McG96] Deborah McGuinness. *Explaining Reasoning in Description Logics*. PhD thesis, Rutgers - The State University Of New Jersey, 1996.

[MdS03] Deborah McGuinness and Paulo da Silva. Infrastructure for Web Explanations. In *Proceedings of the 2nd International Semantic Web Conference*, pages 113–129, 2003.

[Mos05] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, 2005. Retrieved 09/25/2006.

[Pie04] Piero Bonatti et al. Rule-based Policy Specication: State of the Art and Future Work. http://rewerse.net/deliverables/i2-d1.pdf, 2004. Retrieved 09/25/2006.

[PS05] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050217/, 2005. Retrieved 09/25/2006.

[Reg99] Regents of the University of California. The BSD License. http://www.opensource.org/licenses/bsd-license.php, 1999. Retrieved 09/25/2006.

[Riv92] Ronald Rivest. RFC 1321: The MD5 Message-Digest Algorithm. http://tools.ietf.org/html/rfc1321, 1992. Retrieved 09/25/2006.

[Sea06] Andy Seaborne. ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/, 2006. Retrieved 09/25/2006.