Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2006-35

Masterarbeit

im Studiengang "Angewandte Informatik"

Entwicklung und Implementierung eines Domain Brokers für das Semantic Web

Tobias Knabke

in der Arbeitsgruppe für

Datenbanken & Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

1. November 2006

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 1. November 2006

Master Thesis

Development and Implementation of a Domain Broker for the Semantic Web

Tobias Knabke

November 1, 2006

Supervised by Prof. Dr. Wolfgang May
Databases and Information Systems Group
Georg-August-Universität Göttingen

Abstract

Autonomously evolving information systems are the basis of the Semantic Web. In contrast to the World Wide Web of today, the nodes in this Semantic Web reach beyond the pure viewing of web pages. As known from Web Services, they are able to answer on requests. But moreover, the reaction on events, such as database updates generated on remote nodes in the web, is possible.

This reactive behavior was implemented in the Framework for Evolution and Reactivity in the Semantic Web via Event-Condition-Action (ECA) rules. The designated action will be executed if a specific event occurs and an optional condition is fulfilled as well.

To also allow for automated reasoning, the above mentioned framework follows an ontology-based approach. An ontology, which is usually distributed over several nodes in the decentralized Semantic Web, defines a meaningful, computer-processable relation between concepts of a domain as a knowledge base. With these ontologies, a domain-dependent reaction on events is possible.

In this thesis, a Domain Brokering Service for the distribution of events with an Event Broker and the execution of actions with the help of an Action Broker is developed. Additionally, information retrieval in form of SPARQL requests is realized by a Query Broker. To support the facile integration of the Domain Broker and its components into existing information services, standard web technologies are used.

Acknowledgments

First of all, I would like to thank Prof. Dr. Wolfgang May for offering me this interesting topic as a master thesis and for the excellent personal and scientific supervision, not only of this thesis but also during my course of studies.

Likewise, I wish to thank Prof. Dr. José Júlio Alves Alferes for co-supervising this thesis.

Furthermore, I appreciate Erik Behrends, Oliver Fritzen, Franz Schenk and especially Daniel Schubert for their technological and scientific support.

I am grateful to Kristin Stamm and Franz-Josef Rolfes for their constructive comments and motivation.

Special thanks go to my parents, my sister and particularly to my girlfriend Nadine for their support and their patience at any time.

Göttingen, in the fall of 2006 Tobias Knabke

Contents

List of Figures iii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Structure of the Thesis . . . 2

2 Basics 3
- 2.1 Semantic Web . . . 3
- 2.2 Ontology . . . 3
- 2.3 XML and Related Recommendations . . . 4
- 2.4 URI . . . 6
- 2.5 RDF and RDF Schema . . . 6
- 2.6 OWL . . . 8

3 Rules 9
- 3.1 Rules and Ontologies . . . 9
- 3.2 Types . . . 10
  - 3.2.1 Deductive Rules . . . 10
  - 3.2.2 ECA Rules . . . 12

4 A General Framework for Evolution and Reactivity in the Semantic Web 13
- 4.1 General Architecture . . . 13
- 4.2 Modeling of Domain Ontologies . . . 16
- 4.3 Rules in Ontologies . . . 18
- 4.4 ECA Rules . . . 19
  - 4.4.1 ECE Rules . . . 20
  - 4.4.2 ACA Rules . . . 21
- 4.5 Rule Markup . . . 22
  - 4.5.1 Logical Derivation Rules . . . 22
  - 4.5.2 ECA Rules . . . 22
- 4.6 Communication Markup . . . 25
- 4.7 Opaque Expressions . . . 27
- 4.8 Language and Service Registries . . . 27
- 4.9 Domain Service Registries . . . 27

5 Jena Semantic Web Framework **29**
5.1 Jena RDF API . . . 29
5.2 Reasoning and Inference . . . 30
5.3 ARQ and SPARQL . . . 31
5.3.1 ARQ . . . 31
5.3.2 SPARQL . . . 32

6 Domain Brokering **39**
6.1 Event Brokering . . . 39
6.2 Action Brokering . . . 43
6.3 Query Brokering . . . 47
6.3.1 Request Format . . . 49
6.3.2 Query Decomposition . . . 50
6.3.3 Required Nodes Selection . . . 51
6.3.4 Data Integration . . . 52
6.3.5 Query Answering . . . 53
6.4 Miscellaneous . . . 54

7 Implementation **55**
7.1 Employed Technologies . . . 55
7.2 General Architecture . . . 56
7.3 Common Classes . . . 56
7.3.1 Utility classes . . . 56
7.3.2 Ontology . . . 56
7.4 Domain Broker . . . 58
7.4.1 Communication Interface of the Domain Broker . . . 58
7.4.2 Architecture of the Domain Broker . . . 61
7.4.3 Architecture of the Event Broker . . . 61
7.4.4 Architecture of the Action Broker . . . 63
7.4.5 Architecture of the Query Broker . . . 63
7.5 Domain Broker Client . . . 65

8 Conclusion **69**

Bibliography **71**

List of Figures

4.1 General Framework Architecture 14
4.2 Kinds and Components of Ontologies 17
4.3 Derivation and Mapping Rules for Events, Literals and Actions 19

6.1 Architecture: Communication with Event Brokers 40
6.2 RDF Graph of a Travel Route offered by DB 49
6.3 United Ontology 52

7.1 Class Diagram for Utility Classes 57
7.2 Class Diagram for Ontology 58
7.3 Communication Interface of the Domain Broker 59
7.4 Class Diagram for the Domain Broker 62
7.5 Class Diagram for the Event Broker 62
7.6 Class Diagram for the Action Broker 63
7.7 Class Diagram for the Query Broker 64
7.8 Managing Registrations with the Domain Broker Client 65
7.9 Sending SPARQL Queries with the Domain Broker Client 66
7.10 Viewing the Framework Log with the Domain Broker Client 67

1 Introduction

1.1 Motivation

The World Wide Web of today is undergoing an extensive change. The former *static* web evolves more and more from a medium mainly dealing with documents for people into another kind of web. In this Semantic Web that is not a new one but an extension of the current web, information is given a well-defined meaning . This enables machines to "understand" and process the data stored in the Semantic Web and thus to provide behavior in form of *portals* or *Web Services*.

Hence, the nodes inside the Semantic Web can not only be considered as static sources for the storage of web pages anymore. They are rather autonomously evolving heterogeneous information systems. For these data systems it is not sufficient to just act on their local databases. The propagation of information requested from other information systems instead is one central aspect of the Semantic Web.

As the Semantic Web is decentralized, the languages used by the (distributed) nodes are heterogeneous. Since each node usually has its own local data sources, mostly portals are used in the web of today to integrate the information that is isolated in different nodes. A traveler, for example, could use a flight portal, where information about flights (companies, dates, prices, etc.) is shown on a central web page, to choose cost optimal flights from different companies for his journey.

The example above describes an extract of the real world, which could be part of the *travel* domain. A domain consists of related issues in a specific area and usually consists of several nodes. In the *travel* domain, a flight company, e.g., offers its flights on different airports. The booking of a flight by a customer yields a debit of his bank account. This is where the *travel* domain interferes with another application domain, i.e., the *banking* domain.

Such application domains could be distributed over several nodes in the Semantic Web. Each of the nodes has its own data and thus a special behavior. The booking of the last available seat of a flight, known from a local database update, could trigger different local inference rules at the node, e.g., that seats in the fully booked airplane are not available anymore.

The information retrieval via manually programmed portals is very inflexible. To be able to describe the evolution of and the reactive behavior in the Web, the Framework for Evolution and Reactivity in the Semantic Web, described in [24] and [1], follows an ontology-based approach.

The information as a knowledge base can be distributed over several nodes. As a Semantic Web application, the framework allows for the propagation of knowledge and changes in a semantic way. To incorporate the heterogeneity of the web, independent concepts and languages, such as URI, XML, RDF, and OWL, are used for the definition of ontologies and the propagation of information.

As a first step to achieve additional information from underlying concepts, inference rules stored wrt. ontologies provide means for the conduction of automated reasoning. To bridge the gap between reactivity on the one and the heterogeneity of languages in the web on the other side, *Event-Condition-Action (ECA)* rules are used in the framework. These rules, comparable to triggers known from databases, do not only formalize the behavior of a single node in the framework, but they also enable the description of global, i.e., node-overlapping, application-wide behavior. Furthermore, ECA rules take events into account and thus support the integration of dynamic behavior of a node or an application.

To achieve high flexibility, the framework is modularly composed. For example, the ECA rules offer the usage of different languages in their components. This modularity also takes the diverse abstraction levels existing in the Semantic Web into account and separates the semantics of ECA rules from the semantics of the underlying events and actions (cf. [13]).

To manage the dynamic propagation and execution of events and actions respectively, this thesis deals with the development of a Domain Broker as a mediator between different framework components. Taking the existing infrastructure and architecture of the framework as a basis, information retrieval is additionally handled by an implemented Query Broker. To provide a suitable test environment, an exemplary information system acting on a travel ontology has been developed.

1.2 Structure of the Thesis

The thesis is structured as follows: In the next chapter, the basic terms and definitions that come along with this thesis are explained. Different types of rules and their effects wrt. ontologies are described in Chapter 3. A general outline of the framework and a review about framework specific rules as well as their rule markup are given in Chapter 4. Chapter 5 introduces the Jena Semantic Web Framework and gives an impression of SPARQL. Afterwards, Domain Brokering wrt. the Framework of Evolution and Reactivity in the Semantic Web is explained. This includes the processes related to the brokering of events and actions just as much as the handling of a query inside the Query Broker. The implementation of a the Domain Broker is described in Chapter 7. Finally, the thesis is concluded and the main topics wrt. Domain Brokering are identified to be developed in the further implementation of the framework.

2 Basics

This thesis is based on the Framework for Evolution and Reactivity in the Semantic Web that has been introduced in [24]. It follows an ontology- and resources-based approach and provides its reactive behavior by the use of *Event-Condition-Action (ECA)* rules.

In this chapter a basic view of principal concepts and languages is given. These are either used in this thesis or during the development and implementation of a prototypical Domain Broker for the Semantic Web.

2.1 Semantic Web

Today, the World Wide Web is a network of different information resources. The content, usually marked up in HTML[1], is kept in the nodes of this huge heterogeneous network. It was designed almost solely for humans to read and understand. Through the increasing size of the web combined with the availability of new technologies and applications, a strong need arises for computer programs to access information "stored" in the World Wide Web (cf. [34]).

Computers shall get a reliable way to process the semantics of web content rather than fulfill "just" routines like parsing or searching. This extension of the current web, which is not a separate one, is called Semantic Web (see [4]). It can be seen as a web of data which relies on common formats for interchange of data, not only documents (cf. [31]).

As the Framework for Evolution and Reactivity in the Semantic Web, which is a Semantic Web application as well, follows an ontology-based approach, the notion *ontology* is explained next.

2.2 Ontology

The word ontology originally deals with a theory of being or existence. Thomas R. Gruber defines an ontology as "*an explicit specification of a conceptualization*"[16].

In computer science, an ontology is a (shared) model of a domain used as knowledge representation. It consists of a taxonomy which describes a class hierarchy. In addition, an ontology does not only contain what notions are used, but also how they are named. Through these

[1] Hypertext Markup Language, for details see [19].

URIs[2] that connect notions and objects, resources can also be identified uniquely (see Section 2.4). Moreover, an ontology defines all kinds of relationships between the notions of a domain. This gives the notions a meaning which can then be used for reasoning purposes.

Additionally, ontologies might contain rules that are used for reasoning and allow for making implicit knowledge explicit. For details see [2], [34] and [15].

To be able to describe an ontology with computer-processable instruments and to express computer-understandable meaning, more technologies, for example, for knowledge representation, are needed. The concepts which are useful for this thesis are standardized by the W3C [39] and will be introduced briefly in the next sections.

2.3 XML and Related Recommendations

XML. The Extensible Markup Language (XML) is a generic, very flexible but simple text format which was derived from the SGML (for details see [14]) standard. By utilizing element tags and attributes it is used for marking up semistructured data. It allows to add arbitrary structure to a human- and machine-readable document. Therefore, XML can be used as data exchange format.

Since XML is capable of defining the rules for such tree-structured documents, it is called a *meta language*. For a concrete XML application, the structure of an XML document has to be specified, i.e., which names and values of elements and attributes are allowed to appear in which (nested) order.

This can be done in a Document Type Definition (DTD) or more restrictively by an XML Schema Definition (see below). XML can either be used to store data in a well-defined format[3] or to exchange data[4]. To give a very brief impression how XML could look like, consider the following XML snippet:

```
<root-element>
  <subelement-one attribute-one="attribute value" attribute-two="another value">
    element text
  </subelement-one>
  <subelement-two attribute="value">
    another element text
  </subelement-two>
   ...
</root-element>
```

[2] Uniform Resource Identifiers.
[3] For example, serialized in files or databases.
[4] For example, via HTTP over the web.

For more information see [12] and [42].

XML Schema. XML Schema is a schema definition language expressed in XML (1.0) syntax. It describes the structure of an XML document and provides means to constrain the content, i.e., names and values of elements and attributes. A schema is an XML document itself and supplies methods to define the structure, content, and semantics of XML documents in more detail than a DTD does.[5]

A general XML document is mapped to a special application domain through the restriction from its schema and thus the content of the XML document gets more semantics (cf. [43] and [12]).

But the possibilities XML Schema offers wrt. semantics are still not sufficient for reasoning. See [41], [5] or [43] for more details.

XSLT. The Extensible Stylesheet Language Transformations (XSLT) is one part of the XSL family [46]. It allows for the transformation of one XML source document into a result document[6].

The transformation of elements is rule-based and applied recursively. It combines declarative and functional aspects and uses XPath (see below) as addressing language. In [21], [36] and [47] more information can be found.

XPath. The XML Path Language (XPath) is a language applied to XML documents in order to access (as a small query language) and adress individual elements, attributes, or sets of them. The syntax of an XPath expression is abutted to the Unix directory notation.

For navigating from one node to another, each navigation step consists of an axis specifier, a node test, and a predicate. XPath is a basic technology for other standards, e.g., XSLT or XQuery (see below). For detailed information see [44].

XQuery. The XML Query Language (XQuery) is, as the name already implies, a query language to query XML data. It has an SQL-like syntax but also provides programming language features like variable binding and is, different from SQL, orthogonal.

XQuery uses XPath as addressing language.[7] The development of XQuery has been influenced by languages like XQL, XML-QL, OQL, XPath, and SQL. Detailed information can be found at [45].

[5] It allows, e.g., the definition of complex types.

[6] Note that the structure of the result can completely differ from the source structure and is not necessarily in XML format.

[7] Note that XPath is a subset of XQuery.

Since issues that could be described with the languages introduced above are not necessarily combined in one single document, URIs are needed to identify resources.

2.4 URI

Uniform Resource Identifiers (URIs)[8] are attached a great importance in the Semantic Web (see Section 2.5). A URI is a character string that identifies uniquely physical or abstract resources.[9] These resources can be in the scope of the Internet, e.g., web pages, but also outside of machine reachability, like real world persons. The very generic structure of a URI looks like *<scheme>:<scheme specific part>*.

There are different types of URIs such as Uniform Resource Locators (URLs), Uniform Resource Names (URN), etc. With URLs a resource is identified via its primary access mechanism, i.e., through its location (e.g., http or ftp), while URNs identify resources through their names or namespaces (urn:isbn for example), regardless of its location.[10] More information can be found in [38].

As RDF, RDF Schema, and OWL add semantics to computer-processable data by using the concepts described above, these languages will be introduced in the following sections.

2.5 RDF and RDF Schema

In literature, it is often distinguished between the *data level* and the *information* or *knowledge level*. The concepts introduced by now, mainly XML and XML Schema, operate on the data level and do not provide additional information or support reasoning (cf. [13]). Therefore, more expressive languages are needed which are described below.

RDF. The Resource Description Framework (RDF) has been developed by the W3C to encode metadata. It can be used to describe any kind of resource. These resources can be accessed and uniquely identified through a URI (cf. Section 2.4). This allows for the distribution of a resource's description over different nodes, e.g., somewhere in the web. RDF can be represented in many ways, e.g., with a graph, with triples, or in an XML markup called RDF/XML [29]. RDF mainly distinguishes between three components:

- **Resource**: Resources can be any type of data which is identified by a URI and described by RDF expressions.

[8] Originally called Universal Resource Identifier.
[9] An identical URI used in different documents expresses that the same resource is meant.
[10] Thus, a URN is a URI with the scheme urn.

- **Property**: A property defines a special aspect of a resource.
- **Statement**: A statement assigns a value to a property of a specific resource. A statement, also called triple in RDF terminology, has three components:
  - a *subject* which is a resource,
  - a *predicate* which is a property of the resource,
  - and an *object* which is the value of the property. This can, e.g., be a text literal or another resource.

To get a view what RDF basically looks like, consider the following example[11] which describes some book with any title written by some author. The first description uses the N-Triple notation[12]:

```
<http://someurl/little-rdf-book/> <http://purl.org/dc/elements/1.1/creator> “John Doe” .
<http://someurl/little-rdf-book/> <http://purl.org/dc/elements/1.1/title> “John Doe’s Little RDF Book” .
```

The example above can also be expressed in RDF/XML and would lead to the same RDF graph:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://someurl/little-rdf-book/">
  <dc:creator>John Doe</dc:creator>
  <dc:title>John Doe’s Little RDF Book</dc:title>
</rdf:RDF>
```

Thus, RDF can be used to assign responsibilities to resources via properties, for example, a domain could be allocated to a domain broker. To get more information about RDF, have a look at [7], [15], [2] and [27].

RDF Schema. The RDF Vocabulary Description Language, RDF Schema (RDFS), adds semantics to an RDF data model by defining its vocabulary. It is similar to XML and XML Schema, for it transfers a general RDF concept to a specific application domain. Written in RDF, the schema description provides possibilities to characterize groups of related resources and their relationships. Moreover, it allows for inference with the use of metadata. For more information see [12] and [28].

[11] The example uses Dublin Core [10].
[12] N-Triples are a fixed subset of the N3-Notation. See [3] resp. [25] for detailed information.

To offer more possibilities to represent machine-interpretable content and to have more facilities to express meanings, the Web Ontology Language that is based on above described concepts, such as XML, RDF , and RDF Schema, is introduced in the next section.

2.6 OWL

The Web Ontology Language (OWL) is a language to define, publish and distribute ontologies. It is technically based on the RDF syntax, but goes beyond it in its ability to add semantics to the machine-processable and understandable content.

OWL is more powerful in expressing meaning and semantics than XML, RDF, and RDFS. For example, OWL extends RDFS and adds more vocabularies to describe classes, their relationships and properties, e.g., of a domain in the Semantic Web.

Furthermore it permits reasoning over data expressed in OWL. The Web Ontology Language is derived from the DAML+OIL Web Ontology Language [8] and is divided in layers: OWL Lite, OWL DL[13], and OWL Full. Together with RDF it forms the basis of the Semantic Web. For more information see [15] and [26].

By now, only information explicitly stated in a data model or an ontology can be extracted. To be able to gain also implicitly stored knowledge, rules are needed. This will be the focus in the next chapter.

[13] Description Logic

3 Rules

In this chapter it is described how rules can be used to get infered information from an ontology. Afterwards different types of rules will be illustrated.

3.1 Rules and Ontologies

To get more information from an ontology than it explicitly contains, rules can be used. With their help, implicit knowledge can be extracted, e.g., to get inferred models. To achieve this, the rules have to be deposited inside an ontology and proofed against it.

The following example forms a basis for the samples given in the following sections. It serves as a foundation the later defined rules are applied to.

Example 3.1 *Consider the following RDF statements that could be part of a travel ontology. It represents different sections of a train schedule:*

```
<travel:Section rdf:about="#Hamburg-Bremen">
  <travel:from rdf:resource="#Hamburg"/>
  <travel:to rdf:resource="#Bremen"/>
</travel:Section>
<travel:Section rdf:about="#Bremen-Oldenburg">
  <travel:from rdf:resource="#Bremen"/>
  <travel:to rdf:resource="#Oldenburg"/>
</travel:Section>
<travel:Section rdf:about="#Oldenburg-Osnabrück">
  <travel:from rdf:resource="#Oldenburg"/>
  <travel:to rdf:resource="#Osnabrück"/>
</travel:Section>
<travel:Section rdf:about="#Osnabrück-Hannover">
  <travel:from rdf:resource="#Osnabrück"/>
  <travel:to rdf:resource="#Hannover"/>
</travel:Section>
```

3.2 Types

There are several types of rules. On the one hand there are rules that describe a *domain*. They have to be validated wrt. the domain to prove their correctness. This kind of rules specifies the relationship between objects, events and actions of the ontology.

On the other hand rules can also specify the behavior of an *application*, e.g., business rules. The change of such rules results in a different behavior of the application (cf. [1]).

In the next sections, different types of rules which are able to specify a domain and the behavior of an application will be explained.

3.2.1 Deductive Rules

General rules consist of several components. Deductive rules[14] known from logical languages like Prolog (see [37]) or Datalog (see [22]) contain a *rule head* and a *rule body*.

The knowledge base in logical languages consists of a set of facts (predicates) and rules.[15] To allow for the deduction of new facts, logical statements of the form `head :- body` are specified. If the rule body is valid, this directly implies the validity of the head. The `:-` can be read as "if". Thus, `C :- A,B` can be read as "C is true if A and B are true". The "," between A and B in the rule body could have also been written as `A & B` (cf. [37]).

Example 3.2 *To express the sections[16] mentioned in Example 3.1 in a logical programming way, the following facts are used:*

```
section('Hamburg', 'Bremen').
section('Bremen', 'Oldenburg').
section('Oldenburg', 'Osnabrück').
section('Osnabrück', 'Hannover').
```

The following rules use (free) variables[17] to express that in the simplest case X is connected with Y if X is linked by a section (train route) to Y. The second rule states a connection between X and Z if a section between X and Y exists and also Y and Z are connected. The second rule formulates the transitive closure (transitivity) of a connection.

[14] Deductive rules are also called derivation rules.

[15] Because this thesis does not mainly deal with rules, it is assumed that basic notions related to rules, such as *predicates or (free) variables*, are known to the reader. Detailed explanations as well as information about syntax can be found at [37] or [22].

[16] Note that a section from A to B is not the same as a section from B to A.

[17] Variables are written in capital letters, whereas predicate symbols, constants and function symbols begin with lower case letters.

```
connection(X,Y) :- section(X,Y).
connection(X,Z) :- section(X,Y), connection(Y,Z).
```

Now it is possible to query the database to get all connections from X to Y:

```
?- connection(X,Y).
```

This would lead to the following variable bindings in the rule body for the first rule

X → "Hamburg", Y → "Bremen" and
X → "Bremen", Y → "Oldenburg" and
X → "Oldenburg", Y → "Osnabrück" and
X → "Osnabrück", Y → "Hannover"

and

X → "Hamburg", Y → "Bremen", Z → "Oldenburg" and
X → "Hamburg", Y → "Bremen", Z → "Osnabrück" and
X → "Hamburg", Y → "Bremen", Z → "Hannover" and
X → "Bremen", Y → "Oldenburg", Z → "Osnabrück" and
X → "Bremen", Y → "Oldenburg", Z → "Hannover" and
X → "Oldenburg", Y → "Osnabrück", Z → "Hannover"

for the second rule. They are now used in the head to derive the facts

```
connection('Hamburg', 'Bremen').
connection('Hamburg', 'Oldenburg').
connection('Hamburg', 'Osnabrück').
connection('Hamburg', 'Hannover').
connection('Bremen', 'Oldenburg').
connection('Bremen', 'Osnabrück').
connection('Bremen', 'Hannover').
connection('Oldenburg', 'Osnabrück').
connection('Oldenburg', 'Hannover').
connection('Osnabrück', 'Hannover').
```

which leads to the result of the query:

X → "Hamburg", Y → "Bremen" and
X → "Hamburg", Y → "Oldenburg" and
X → "Hamburg", Y → "Osnabrück" and
X → "Hamburg", Y → "Hannover" and
X → "Bremen", Y → "Oldenburg" and

$$\mathsf{X} \rightarrow \text{“Bremen”}, \mathsf{Y} \rightarrow \text{“Osnabrück” and}$$
$$\mathsf{X} \rightarrow \text{“Bremen”}, \mathsf{Y} \rightarrow \text{“Hannover” and}$$
$$\mathsf{X} \rightarrow \text{“Oldenburg”}, \mathsf{Y} \rightarrow \text{“Osnabrück” and}$$
$$\mathsf{X} \rightarrow \text{“Oldenburg”}, \mathsf{Y} \rightarrow \text{“Hannover” and}$$
$$\mathsf{X} \rightarrow \text{“Osnabrück”}, \mathsf{Y} \rightarrow \text{“Hannover”} .$$

Until now, no impulse, i.e., an event from the “outside world” has been needed to extract new information with the help of rules. Just a *condition*, in form of the rule body, has to be fulfilled to provoke an action that is in this case the derivation of the rule head. Hence, the inferred information can be seen as *static* because it was derived from *static* facts.

Thus, by now reasoning is just possible with *static* rules which belong to the ontology. But this is not sufficient for the Semantic Web which also needs the application of *active* rules.

To integrate also *dynamic* aspects, another kind of rules must be taken into consideration which is described now.

3.2.2 ECA Rules

As mentioned earlier, the Semantic Web consists of many (autonomous) nodes. Each of them has a local state of facts, metadata, and optionally a knowledge base. Again optional, behavior can be described in a node through ECA (*Event-Condition-Action*) rules. ECA rules will fire an action provoked by an event if a certain condition is fulfilled (cf. [24]).

ECA rules show a similarity to triggers known from databases. From an external point of view they can be formulated as “when something happens and some conditions are fulfilled then something has to be done”. The semantics of ECA rules can be generalized as

ON event **AND** additional knowledge **IF** condition **DO** something.

Such *active* rules permit to control an application’s behavior. These very abstract ECA rules are called ECA-Business rules.

ECA rules in contrast to deductive rules also take events into account (cf. [1]). To integrate dynamic behavior in form of ECA rules, a framework is needed. This framework will be introduced in the next chapter.

4 A General Framework for Evolution and Reactivity in the Semantic Web

The Web of today does not only consist of browsing-oriented documents but also of nodes which, in general, provide a behavior, often summarized as *Web Services* (see [22]). Fixed sets of autonomous *information systems* are integrated through portals usually by "hard coded" services. This problem of the *data* and *semantic heterogeneity* shall be bridged by the Semantic Web. The basis of this thesis is a framework which deals with evolution and reactivity in the Semantic Web. It was introduced in [24]. After the general description of the framework and domain ontologies, framework specific rules and their corresponding markup are explained.

4.1 General Architecture

As mentioned above, the Semantic Web consists of many heterogeneous nodes. Hence, it is not surprising that the Framework for Reasoning and Evolution in the Semantic Web is employed in a distributed environment as well. It has to be taken care of a multitude of resources in form of application nodes in different domains wrt. different ontologies.

Domains and the different languages appearing in the Semantic Web are identified by their URIs (see Section 2.4). They are resources like every object that is integrated in the framework. Thus, the framework is resource-based.

As the nodes inside the Semantic Web are heterogeneous, different languages appear in the framework, too. The implementation of each language is done by a Web Service that is associated with the language (cf. [1]).

The architecture of the framework is illustrated in Figure 4.1[18]. The following example (adapted from [30]) describes the coherences depicted in this figure and is based on a use case of the travel domain.

Example 4.1 *Imagine a client C, in this case a travel agency, wants to be informed via email about events (canceled flights) from the* travel *domain in order to inform its staff to book a hotel room for the customer and to reschedule the next flight on the journey.*

[18] Note that Figure 4.1 gives an abstract overview of the architecture of the framework and does not show every aspect as, for example, the handling of queries stated against the Domain Broker.

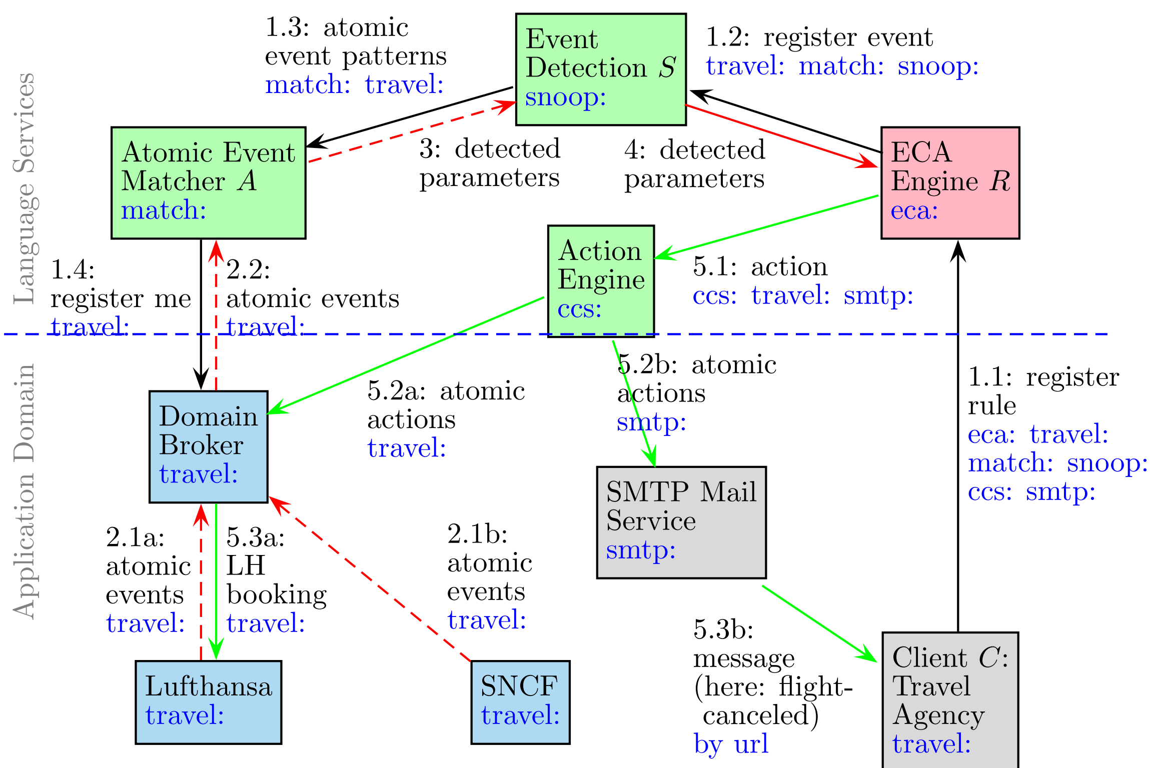

Figure 4.1: General Framework Architecture (adapted from [13])

Therefore, the agency registers a rule that contains a composite event at the ECA engine R (1.1). This composite event is composed of a flight-booked *event that covers the name and the email address of the passenger, the flight number and the flight date. A* flight-canceled *event terminates the event composition which is given in SNOOP [6].*

Because the ECA engine is not responsible for the detection of (composite) events, it registers the event component after analyzing the rule at an appropriate Event Detection Service S (1.2) that is able to detect composite events specified in SNOOP. The SNOOP service S then submits the atomic event patterns at an adequate Atomic Event Matcher (AEM) A (1.3).

*To be able to detect the occurrence of the relevant events, the requirements of the events are defined by Atomic Event Specifications (AESs). AESs are the simplest kind of event components and thus they can be seen as the leaves of an event component. An AES specifies on which events a reaction has to be taken. Hence, the AES for an event in this example contains the name of the event (*canceled-flight*) and its parameters. The formalisms to detect the relevant events are implemented by an AEM.*[19]

[19] Cf. [1].

The AEM, in turn, registers at a dedicated Event Broker of the travel *domain (1.4) since the atomic events are part of the* travel *domain. The Event Broker is managed by a Domain Broker (shown in Figure 4.1) that consists of an Event-, Action-, and Query Broker*[20].

This is just one possible solution how the Event Detection Engine becomes aware of all relevant atomic events. Before an Event Broker has been implemented during this thesis, the clients had to find out about relevant events themselves.[21] *In case an event occured, clients forwarded the events to the rule evaluation service. Afterwards, this service forwarded the event to all common event detection engines.*

The Event Broker serves as a mediator for the distribution of events of a specific domain. Atomic events are produced inside an application at nodes somewhere in the sphere of the framework. A node which produces atomic events of the travel *domain, e.g., the Lufthansa or SNCF, has to identify the responsible Event Broker*[22] *for the domain and send the events to the broker (2.1a and 2.1b respectively). The task of the Event Broker is now to forward the received event to the registered AEM A (2.2).*

A informs the Event Detection in case the event matches the registered patterns (3). If the former registered composite event is detected by the Event Detection, R will be notified (4). The ECA engine then evaluates the appropriate rules and triggers, for example, the processing of the action part of the rule registered in step 1.1.

*The basic procedure of an action processing is the same as for events (cf. [13]). Therefore, consider an extension of the rule described at the beginning of the example that in case of a flight cancellation automatically a new flight on the next day is booked (additionally to the sending of an email to the travel agency). At an appropriate action language service (here: CCS (Calculus of Communication Systems)), the action component is submitted (5.1). The Action Engine, in return, looks up the responsible Action Broker for the domain (*travel *in this case) and forwards the atomic actions to the Action Broker, managed by the superior Domain Broker (5.2a). The Action Broker disposes the execution of the action (the booking of a new flight on the next day) at the appropriate domain nodes (5.3a). If the registered rule (1.1) contains a notification request (sending of an email in this case) from Client C, the Action Engine also will send the action to a domain-independent service (5.2b), here an SMTP Mail Service, which, in turn, sends a message to the Client C (5.3b).*

As described in Example 4.1, client nodes inside the Semantic Web, e.g., a travel agency, want to be informed about special events and react on them. Events are generated by other nodes somewhere in the web. To specify the behavior of the reactions, clients register their rules

[20] The Query Broker will not explicitly be mentioned in this example.
[21] This solution is not shown in Figure 4.1.
[22] The Event Broker is supervised by a Domain Broker.

locally or remotely. Remote rules are evaluated by an evaluation service, namely an ECA engine. These engines can be interpreted as main objects for the reactive behavior in the framework.

An event can trigger, for example, the evaluation and execution of ECA rules which are handled by ECA engines. The propagation of events is done by Event Brokers that receive registrations for events of a certain domain and event type. A registration that is sent to an Event Broker basically looks as follows:

```
<register>
  <reply-to>URL where the events shall be forwarded to</reply-to>
  <domain>domain-URI</domain>
  <event-type>event type</event-type>
</register>
```

Incoming events are forwarded wrt. the obtained registrations by the Event Broker of the domain. The events as well as the actions have to be instantiated at appropriate services, e.g., at a CCS engine or at a domain node in order to be processed correctly by the relevant brokers.

To react on events, (some) nodes must be able to execute actions. Hence, Action Brokers are informed by certain nodes that an action has to be performed. For example, an airport operating company could decide to cancel a flight due to weather conditions. Then the Action Broker has to find out where[23] the action has to be executed.

Beside the propagation of events and the execution of actions, the retrieval of information is another important service of the framework. To offer a wide functionality to the clients, it must be possible to state queries against the nodes integrated in the framework. A hotel, for example, may be asked by a customer to rent a high class car for an excursion. Therefore, the hotel wants to know which nearby car rental company provides these cars and at what price they are offered. Requests like these are processed by Query Brokers. They have to decompose the request, check which concepts and nodes are relevant, integrate the collected data, state the query, and finally send the result back to the asking node.

To be able to provide the services described above, the framework follows an ontology-based approach. How the ontologies of specific domains are structured is subject of the next section.

4.2 Modeling of Domain Ontologies

A domain ontology describes all objects and their relationships of a specific domain. A *complete* domain ontology does not only contain *static* issues, but also the *dynamic* aspects of the domain.

[23] For example, at the affected flight company.

The resources in a Semantic Web environment build the *static* part of the domain ontology, while, e.g., events and actions represent *dynamic* issues. Furthermore, a domain ontology can be classified by answering the question if a domain ontology depends on an application or not.

Application domain ontologies describe all static and dynamic aspects that are related directly with the application. For example, an ontology of a travel domain characterizes resources like railway or flight companies with their associated schedules. The dynamic part could contain events like *train-delayed* or *flight-half-booked*, whereas *cancel-flight* stands for an action.

In contrast to the just presented ontologies, *application-independent domain ontologies talk about* an application (see Figure 4.2). They make a generic infrastructure available by providing services like transactions, messaging, or calendars. Application-independent domains contain also static and dynamic notions and can be combined with arbitrary application domains. An example of an application-independent domain ontology is a *calendar*. It can be seen as a class of service that defines, e.g., a *year* as a resource and can also provide *first day of year* as an event.

A complete ontology in the Semantic Web combines these different types of ontologies. Note that in a Semantic Web application often several domains interfere with each other, e.g., *travel* and *banking*.[24] Figure 4.2 illustrates the relationship between the components of an ontology and the interaction between the different types of ontologies.

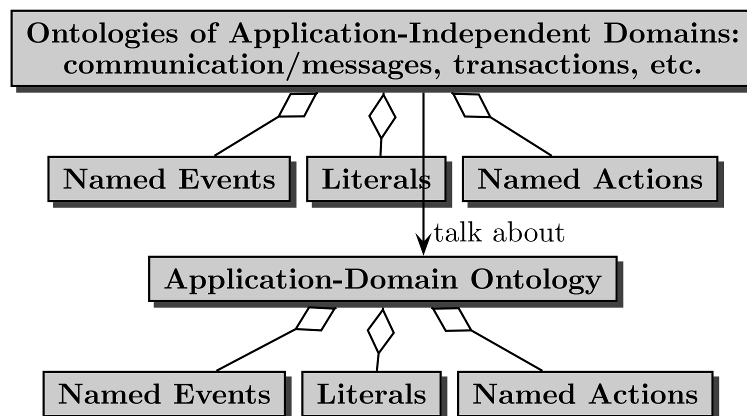

Figure 4.2: Kinds and Components of Ontologies (from [13])

As seen in Figure 4.2, a domain ontology contains literal notions, named events and named actions. Events and actions can be structured comparable to a class hierarchy. A *metadomain* contains notions that define and structure a domain ontology. This *metadomain* is associated

[24] Flight tickets are booked at a travel agency which is part of the *travel* domain, but the clearing takes places at a node belonging to the *banking* domain.

with the *world* namespace that is connected to the URL
http://www.semwebtech.org/domains/2006/world. The *metadomain* might, for example, contain the following definitions:

<world:Domain, rdf:type, owl:Class>
<world:Event, rdf:type, owl:Class>
<world:Action, rdf:type, owl:Class>

In the framework, each domain, e.g., the *travel* domain, is usually associated with an URL, where descriptions of the special domain can be found. This referenced document contains the RDF/RDFS and OWL expressions that define the domain ontology itself. The *travel* domain contains, among others, the following definitions (cf. [13]):

<travel:, rdf:type, world:Domain>
<travel:airline, rdfs:subClassOf, world:Domain-Service>
<travel:flight-event, rdfs:subClassOf, world:Event>
<travel:car-rental, rdfs:subClassOf, world:Domain-Service>
<travel:cancellation, rdf:type, world:Event>
<travel:canceled-flight, rdf:type, world:Event>
<travel:canceled-flight, rdfs:subClassOf, travel:cancellation>
<travel:canceled-flight, rdfs:subClassOf, world:Event>
<travel:cancel-flight, rdf:type, world:Action>
<travel:delay-flight, rdf:type, world:Action>
<travel:fully-booked, rdf:type, world:Event>
<travel:half-booked, rdf:type, world:Event>
<travel:booking, rdf:type, world:Event>

Besides events and actions, an ontology contains different kinds of rules which define the behavior of an application. These will be explained in the following sections.

4.3 Rules in Ontologies

In ontologies occur different types of rules, namely:

- logical derivation rules that derive property instances or concept memberships,
- *Event-Condition-Event* (ECE) rules which infer composite events from simpler ones, and
- *Action-Condition-Action* (ACA) rules that map high-level actions to simpler ones.

Some rules directly belong to an ontology while others are connected to certain services. Figure 4.3 shows types of rules that belong to an ontology.

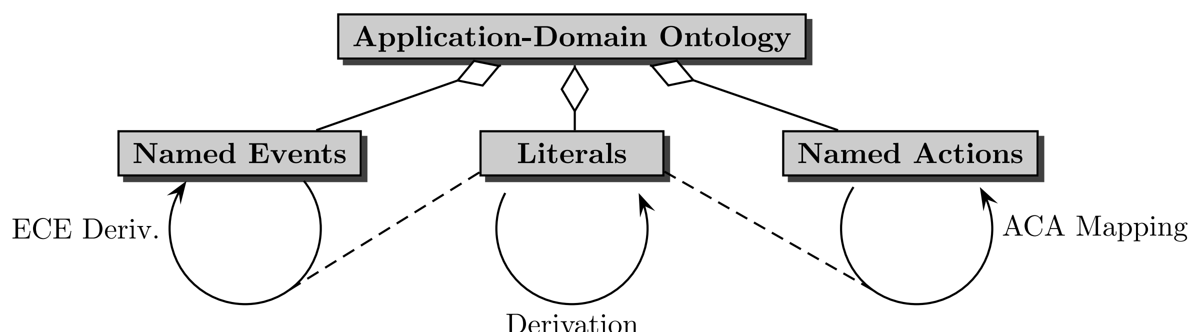

Figure 4.3: Derivation and Mapping Rules for Events, Literals and Actions (from [13])

Usually, ECE rules derive composite events from other events while ACA rules define complex (e.g., composite) actions which are built upon other actions (e.g., simpler or lower-level actions). Since ECE and ACA rules are special kinds of ECA rules, the latter are explained more detailed in the next section.

4.4 ECA Rules

The Framework for Evolution and Reactivity in the Semantic Web follows an ECA-based approach, i.e., ECA rules implement the behavior of an application and in the next step the underlying ontologies will be extended to act as a base for reasoning (cf. [1]).

Different languages may occur in the rule components to support the heterogeneity of nodes in the Semantic Web. These rule components are, in turn, handled by different nodes. Therefore, the framework provides the ability to use diverse languages in the components.

Component Languages in ECA Rules. Prolog and Datalog are rule-based languages. The expressions of their head and body are both written in the same language. This is not the case for ECA rules required in the framework (cf. [24]). As the structure is distributed over many different nodes in the Semantic Web, different languages in the components of a rule have to be taken into account to realize interoperability between the nodes. Therefore, every rule uses an event language, one or more query languages, one test language, and one or more action languages in its components (cf. [30]):

- Event Language: The language used in the event component has to support the detection of events. Atomic events are given as an XML fragment, thus, the applied language must

be able to deal with the content and the structure of such an XML document. XPath matches the demanded conditions. SNOOP can be used to handle composite events, which are built by several atomic events.

- Query Language: Query languages must provide the functionality to deal with sets of data items which can also be bound to variables. Logic and functional languages offer this functionality. XQuery provides this functional property, too.
- Test Language: Expressions of test languages must result in a logical truth value, i.e., `false` or `true`. Results of functional languages (including XPath and XQuery) can be interpreted as such.
- Action Language: In the action component it is expressed what actually has to be done. Here, arbitrary languages, like "classical" programming languages (Java,...) or algebraic languages (as process algebras), can be used. The use of Web Services via HTTP requests is also possible.

Used on different levels, such user-defined, "actual" rules act on events and are responsible for the behavior of the application. ECA rules are used internally for the detection of events of the respective level (see [1]).

Since ECE and ACA rules are special types of ECA rules, their implementation is based on ECA rules. They also use (different) languages in their components. As a characteristic, the condition component is not only used as a test, but also as query part, e.g., for the extension of variable bindings (cf. [13]). These specific ECA rules are described in detail below.

4.4.1 ECE Rules

Event-Condition-Event (ECE) rules infer an (composite) event from other events. Thus, the action consists of raising another event. For example, the event *hotel booked up on 31.12.2006* could be derived from *book room xy on 31.12.2006* if room xy was the last available room on that date.
ECE rules can be classified in the following way (cf. [1]):

- horizontal ECE rules: The high-level event is derived under certain conditions from another event on the same abstraction level. These events are logically related and a change of the events would affect the behavior of the application and invalidate it wrt. its ontology.

- upward vertical ECE rules: These rules are evoked by changes in the underlying database. Here, the events are not logically connected but related through the physical implementation of the application. An example could be the change of a flight's arrival time in a database which results in a delayed flight. The change of such rules will again invalidate the application wrt. its ontology.

4.4.2 ACA Rules

An *Action-Condition-Action* (ACA) rule maps higher-level actions to lower-level or simpler actions. These rules are needed because the direct execution of *abstract* rules is often not possible. This kind of rules is called *reduction* rules because they reduce *abstract* actions to simpler levels, e.g., in the same application domain or to its components on a lower level.
ACA rules can be divided as follows (cf. [1]):

- Horizontal reduction ACA rules are described by their components which are in terms of the application, i.e., on the same abstraction level. For example, *the transfer of money from a bank account to another* is split up into *the debit of money from one account* and *the deposition to another bank account.*

- Vertical reduction ACA rules reduce one higher level action into lower ones, for example, *the deposition of money from a bank account* is broken down into *reading the amount of money*, *adding the deposition value*, and *writing the new value.*

- Horizontal non-reduction ACA rules use the concept of *rule chaining.* An action that is not mapped to other actions and thus has to be executed as it is, is seen as an event. This yields the triggering of another action. This kind of rules is more related to ECA rules. A change of this rule would change the behavior of an application, but not invalidate it.

To clarify the partitioning in a nutshell, *transfer 500 Euro from account A to B* by *debit 500 Euro from account A* and *deposit 500 Euro to account B* is an example for the first type of ACA rules (i.e., horizontal reduction ACA rules), whereas *debit 500 Euro from account A* yields *read value from account A*, *subtract 500*, and *write the new value back* is an example for a vertical reduction ACA rule.

To operate the rules in a distributed environment, a uniform markup is needed which is explained in the next section.

4.5 Rule Markup

In order to use rules in a heterogeneous and dynamic environment, their markup has to be general and independent from certain languages. Different kinds of markups are introduced below.

4.5.1 Logical Derivation Rules

Logical derivation rules or deductive rules are used wrt. an ontology. The very generic XML based markup follows the RuleML proposal described in [17]. The basic structure is shown below:

```
<ruleml:imp annotation="rule-specific annotation">
  <!-- there may be several head elements for the derivation of new facts -->
  <ruleml:head>head specification</ruleml:head>
  <!-- there may be several body elements -->
  <ruleml:body>body specification</ruleml:body>
</ruleml:imp>
```

4.5.2 ECA Rules

An XML based markup language, called ECA-ML and described in [23], is used in the framework to define ECA rules. The basic markup has the following form (cf. [30]):

```
<eca:rule rule-specific attributes>
  rule-specific content, for example, declaration of logical variables
  <eca:event identification of the language>
    event specification, probably binding variables
  </eca:event>
  <!-- there may be several queries -->
  <eca:query identification of the language>
    query specification, using variables, binding others
  </eca:query>
  <eca:test identification of the language>
    condition specification, using variables
  </eca:test>
  <!-- there may be several actions -->
  <eca:action identification of the language>
    action specification, using variables, probably binding local variables
  </eca:action>
</eca:rule>
```

The event specification enveloped in <eca:event> also contains the atomic event specifications (AESs) in an appropriate format, e.g., in XML-QL Style Matching or based on the SNOOP algebra (cf. [1]). Comparable to events, the atomic action specifications (AASs) are enclosed in the (action specification) expressions inside the <eca:action> element.[25]

To clarify what a concrete ECA rule looks like, the rule verbally described at the beginning of Example 4.1 is shown below:

```
<eca:rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml"
          xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy"
          xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
          xmlns:mail="...">
  <eca:event>
    <snoopy:sequence>
      <travel:book-flight date="{$date}" flight="{$flight}" name="{$name}"
                          email="{$travel-agency}"/>
      <travel:canceled-flight date="{$date}" flight="{$flight}"/>
    </snoopy:sequence>
  </eca:event>
  <eca:action>
    <eca:input-variable name="travel-agency" use="$travel-agency"/>
    <eca:input-variable name="flight" use="$flight"/>
    <mail:send-message to="$travel-agency">
      we are very sorry ...  but the flight $flight had to be canceled
    </mail:send-message>
  </eca:action>
</eca:rule>
```

ECE Rules inside Ontologies. ECE rules are special types of ECA rules (cf. Section 4.3). The action part of the ECA rule here is the raise of a new (derived) event. The markup for ECE rules basically looks as follows (cf. [13]):

```
<world:definition annotation="rule-specific annotation">
  <world:defined syntax="xml">
    <!-- pattern of the event to be derived -->
    pattern of the derived event, marked up in XML
  </world:defined>
  <world:defined-as syntax="xml">
    <!-- event components -->
    event pattern, marked up in XML
    <!-- test component -->
```

[25] Since the languages for specifying atomic events or actions do not belong to the topic of this thesis, these are not discussed in detail here. For further information see [1].

```
    test pattern, marked up in XML, can contain opaque expressions
  </world:defined-as>
</world:definition>
```

ACA Rules inside Ontologies. Analogously to ECE rules, in this case the event part of an ACA rule consists of an action. This more complex, abstract action can be broken down into simpler named actions which are still abstract or into local implementations of named actions. Below, the structure of ACA rules is shown (cf. [13]):

```
<world:definition annotation="rule-specific annotation">
  <world:defined syntax="xml">
    <!-- pattern of the abstract action -->
    pattern of the abstract action to be broken down, marked up in XML
  </world:defined>
  <world:defined-as syntax="xml">
    <!-- test component -->
    test pattern, marked up in XML, can contain opaque expressions
    <!-- action components -->
    action patterns to which the abstract action is broken down, marked up in XML
  </world:defined-as>
</world:definition>
```

ECE and ACA Rules for Framework Exchange. To be able to operate ECE and ACA rules inside the framework, the rule definitions have to be converted into a framework-known format that can be handled by a dedicated engine, e.g., an ECA engine[26]. The structure for ECE rules was briefly introduced in [13] and is shown below:

```
<eca:rule>
  contents of the body of the ECE rule definition; i.e. the triggering event
  description and the condition
  <eca:action>
    <eca:raise-event>
      head of the ECE rule; i.e. the derived event specification
    </eca:raise-event>
  </eca:action>
</eca:rule>
```

ACA rules have also to be transformed into a format that framework-aware engines can deal with. Therefore, a structure was described in [13] that looks as follows:

[26] Note that the responsible engine has to be extended to be able to handle the below presented rule formats.

```
<eca:aca-rule>
  variable declarations
  <eca:define-action>
    head of the ACA rule, i.e. the abstract action specification
  </eca:define-action>
  <!-- there may be no, one or more queries -->
  <eca:query>
    query specifications, using variables, binding others
  </eca:query>
  <!-- there may be no or one test -->
  <eca:test>
    condition specifications, using variables
  </eca:test>
  <!-- there may be several actions -->
  <eca:action>
    action specification to which the abstract action is broken down
  </eca:action>
</eca:aca-rule>
```

Besides rules, also other information types have to be exchanged inside the framework. For example, the result of a SPARQL query that was stated against a Query Broker (cf. Section 6.3). Therefore, the framework-wide communication format for variable exchange is introduced in the next section.

4.6 Communication Markup

Before the framework-wide representation for variable bindings will be described, different kinds of information interchange, based on logical variables, are introduced first.

Downward Communication. In case of a rule registration, the event component is communicated to an appropriate event detection engine (cf. Example 4.1 and Section 4.5.2 for the rule instance). Moreover, if variables are already bound, they can also be contained in the message sent to the event detection service. If an event occurs that is sought-after, then queries, tests, and actions will result in certain variable bindings due to rule evaluation. These bindings have to be published to other services as well (cf. [13]).

Upward Communication. As part of upward communication, the results and variable bindings of evaluation services have to be returned. These can, for example, be sets of data items as result of queries (cf. [1]).

As also events and actions have to be interchanged as fragments, great importance is ascribed to the Domain Broker (cf. Chapter 6) .

To be able to interchange variable bindings within the framework, a unique communication format, described in [13], is used. Its basic structure is illustrated below[27]:

```
<!ELEMENT answers (answer*)>
<!ELEMENT answer (result|variable-bindings|(result,variable-bindings))>
<!ELEMENT result ANY>
<!ELEMENT variable-bindings (tuple+)>
<!ELEMENT tuple (variable+)>
<!ELEMENT variable ANY>
<!ATTLIST variable name CDATA #REQUIRED
            ref URI #IMPLIED> <!-- variable has either ref or content-->
<logic:answers>
  <logic:answer>
    <logic:result>
      any result structure
    </logic:result>
    <logic:variable-bindings>
      <logic:tuple>
        <logic:variable name="name" ref="URI"/>
        <logic:variable name="name">
          any value
        </logic:variable>
          :
      </logic:tuple>
    </logic:variable-bindings>
  </logic:answer>
  <logic:answer>
    :
  </logic:answer>
</logic:answers>
```

Even if a unique communication format is used to interact consistently among the nodes that act inside the framework, it should not be ignored that nodes in the Semantic Web use different languages. Hence, a Semantic Web application has to take the heterogeneity of its participants into account. As mentioned in Section 4.4, the Framework for Evolution and Reactivity in the Semantic Web allows for the use of different languages by using *opaque expressions*. These are described in the following section.

[27] The namespace of *logic* references to http://www.semwebtech.org/lang/2006/logic .

4.7 Opaque Expressions

As many different nodes act inside the framework, not every service is framework-aware and accordingly marked up in XML. Therefore, means have to be provided to offer the integration of these nodes.

With opaque expressions, services that are framework-unaware and/or not marked up in XML can be embedded into the framework. Opaque expressions can, e.g., occur in queries sent to a query engine managed by a Domain Broker (see Section 6.3). To use opaque languages in ECA rules, the framework-unaware language has to be embedded as follows:

```
<opaque lang="language specification">
  opaque code
</opaque>
```

To detect which services are offered inside the framework and to identify to which languages the services are connected to, Language and Service Registries are needed.

4.8 Language and Service Registries

Language and Service Registries (LSR)[28] implement the identification of appropriate services, i.e., who offers which service. They can be offered by every node inside the Semantic Web. A central LSR is not provided by the ECA framework. Thus, it is sufficient to know a *good* LSR or to know, for example, an ECA engine that, in turn, knows a *good* LSR.

In the final version, LSRs can be used by nodes in the framework to get and look up information. The request has a connection to a language and the answer from the LSR contains information how the task can be requested (cf. [13]).

But such services are not only needed globally, i.e., overlapping all domains on the framework level. To publish which nodes of a certain domain actually support which of the domain notions (described in Section 4.2), Domain Service Registries are needed.

4.9 Domain Service Registries

A Domain Service Registry (DSR) provides metadata information about the services offered by individual nodes of the domain inside the framework. Since the name implies, a DSR is a registry and, thus, not static, but can be changed through the addition of a new service.

[28] Currently a LSR is developed during a thesis.

Application services are available at a URL. The information about an application service is maintained by DSRs. A DSR declares which concepts are actually supported by a certain application service. A railway company, for example, will support *delay-train* but not *cancel-flight*.

The statements below could be part of an application service description maintained by a DSR:[29]

<LH, rdf:type, world:company>
<LH, world:uses-domain, http://www.semwebtech.org/domains/2006/travel>
<LH, world:uses-domain, http://www.semwebtech.org/domains/2006/business>
<LH, business:has-business, world:airline>
<LH, world:has-service, http://url-to-service/lh>
<LH, world:has-url, http://url-to-lh-homepage/index.html>
<LH, world:supports, travel:airport>
<LH, world:supports, travel:flight-connection>
<LH, world:supports, travel:cancel-flight>
<LH, world:supports, travel:is-delayed>

As seen above, domain ontologies are assisted by application services which, in turn, make use of domain ontologies. An application service, for example, an airline company, can use several domains. The booking of a flight ticket is assigned to the *travel* domain, while the payment of a booked ticket is included in the *banking* domain. As an application service can use different domains, it can also support several ones, e.g., *travel* and *business* in case of an airplane company.

This dynamic information about specific domains is provided by the *world* ontology that supplies concepts to state which node in the Semantic Web actually supports which domain and notion as well as which brokers are related to a domain.

In this chapter, the Framework for Evolution and Reactivity in the Semantic Web with its basic architecture and concepts has been introduced. Another kind of framework that supports the development of Semantic Web applications is described in the next chapter. It has been used during this thesis to implement the Domain Broker and its components.

[29] Because there is no DSR available as a Web Service in the framework yet, this information would be content of an ontology file inside a Domain Broker.

5 Jena Semantic Web Framework

Jena [20] is an open source framework for building Semantic Web applications with Java [35]. It arose from the HP Labs Semantic Web Programme [18]. The provided environment allows for dealing with RDF, RDFS, OWL and SPARQL (see Section 5.3.2). Moreover, it includes an (rule-based) inference engine. While offering the mentioned functionality, the Jena Framework is very convenient to model ontologies and their related issues.

The Jena Framework supports the user in many ways: It provides an RDF- and OWL-API[30], has means to read and write RDF/XML as well as N3 and N-Triples respectively. Moreover, the modeled data can either be stored in-memory or persistent. To support to query a constructed ontology, a SPARQL query engine is available.

The provided means that are relevant for this thesis are introduced now (more detailed information related to this chapter can also be found at [20]).

5.1 Jena RDF API

The Jena RDF API provides tools in form of Java classes to deal with RDF. It permits the creation of RDF models with Java by using the dedicated APIs. To keep the code readable, the use of prefixes is supported. Therefore, Jena makes methods available to write an RDF model to a file as serialized XML. Methods for reading models from a file are also designated.

But to work with RDF data, reading and writing files is not sufficient. Thus, Jena supports methods to navigate through a model in order to process the information held in a model. The framework offers the access and manipulation of the objects that are represented as sets of statements, each containing *subject*, *predicate*, and *object* of the RDF graph.

Moreover, the Jena Framework provides means to query models. But the core API of Jena supports only very restricted search primitives. As the queries are embedded in Java, they have to be formulated *imperatively.* Hence, they are not as powerful as *declarative* query languages as, for example, SQL or SPARQL (described in Section 5.3.2).

Jena enables several operations on models: union, intersection, and difference. These concepts are known from mathematical set theory and behave in the same way in Jena.

[30] Application Programming Interface

5.2 Reasoning and Inference

To support the usage of languages like OWL or RDFS, one part of the Jena Framework provides means for reasoning. Therefore, diverse reasoning engines[31] can be plugged into Jena. To work with a reasoner, it is useful to create inferred models by using the functionality provided by the Jena API. This thesis focuses on the general rule based reasoner that allows inference with the application of self defined rules. Therefore, a brief overview of the rule format that Jena claims is given now.

Rule Format. Rules in Jena basically consist of an optional rule name and a rule term which can be nested and constructed under the use of functions.

Example 5.1 *The following rule defines an "aunt" and visualizes the basic rule syntax in Jena, here in forward mode (body → head) :*

[aunt: (?m ns:mother ?x) (?s ns:sister ?m) → (?s ns:aunt ?x)]

The rule says, that if m[32] is a *mother*[33] of x and s is a *sister* of m, then s is an *aunt* of x.

Jena supports different ways to express and process rules. Besides a forward and backward mode, it is also possible to combine these rule styles.

Forward Chaining Engine. The evaluation of rules in a bottom-up style is called *forward chaining*, since it starts from facts to derive new tuples. This results in a benefit of forward chaining[34], because optimization and evaluation techniques from relational algebra may be applied (for details see [22]).

The first time an inferred model is queried by applying a reasoner configured in forward mode[35], a deduction graph is created. Rules that fire can trigger additional rules. Thus, the process holds on until the graph is stable.[36] This leads to a drawback of forward chaining: If a request is interested only in a small area of the data basis, nevertheless the whole data will be comprised in the inference and after that the relevant section will be extracted (see [22]).

[31] Two examples of predefined reasoners are the Transitive Reasoner or the Generic Rule Reasoner.
[32] Variables are denoted by a *?* at the beginning.
[33] "ns" in the rule stands for a specific namespace.
[34] To be able to process rules specified in forward ("→") mode, the applied reasoner has to be configured to run in forward mode.
[35] Note that a reasoner configured in forward mode treats all rules as if they were forward rules, even backward rules.
[36] Note that it is easily possible to create infinite loops.

Backward Chaining Engine. The execution strategy, a rule reasoner running in backward chaining mode follows, is comparable to Prolog engines. In case of a query, the logic programming engine translates the query into a goal and tries to satisfy it by matching the stored triples backward against the rule. This avoids the disadvantage mentioned wrt. forward chaining (cf. [22]).

Hybrid Rule Engine. A rule reasoner may also be configured in hybrid mode, i.e., the reasoner can handle both, forward and backward rules. This option can be used to achieve better performance.

5.3 ARQ and SPARQL

ARQ is a query engine inside the Jena Framework that supports the SPARQL query language (see [32] for details) for RDF. Before the introduction of SPAQRL, an overview of ARQ's functionalities is given.

5.3.1 ARQ

ARQ, a SPARQL processor for Jena, does not only assist in SPAQRL queries, but provides also means for multiple query languages:

- RDQL,
- SPARQL, and
- ARQ, which is the engine's own language and mainly used for experimental purposes.

In addition to supply many query languages, ARQ also features multiple query engines:

- a general purpose engine,
- remote access engines, and
- a rewriter to SQL.

Furthermore, it provides command line utilities to parse or execute queries, to run test sets, or to handle result sets. Since Jena is a Java framework, SPARQL requests can of course be embedded into Java code using the allocated API. Through this embedding it is possible to extend the SPARQL provided functionality. Moreover, it is tolerated to use customized functions in SPARQL FILTER expressions (see Section 5.3.2). These customized function library, provided by Jena, can be extended and used to map queries into application specific functions.

5.3.2 SPARQL

SPARQL is a recursive acronym and stands for SPARQL Protocol And RDF Query Language. SPARQL is both, a query language as well as a data access protocol and language for the Semantic Web. SPARQL delivers information from RDF graphs, which are a set of triples. The triples consist of a *subject*, a *predicate*, and an *object* (see Section 2.5). SPARQL provides functionalities to

- extract information represented as literals, blank nodes, and URIs,
- gather RDF subgraphs, and
- build new RDF graphs upon information achieved from the queried graphs.

Moreover, it can be used locally and remotely to access RDF data based on matching graph patterns.

To point out the syntax and functionality of SPARQL, the following RDF/XML document that could be part of a car rental domain contains two car rental companies and several cars as well as their category. Together with their category, it provides a basis of the later examples. Imagine it can be addressed by http://localhost/exampleontology/carrental.rdf.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:carrental="http://www.semwebtech.org/domains/2006/carrental#">
  <rdf:Description rdf:ID="JohnDoeCarRentalService">
    <carrental:hasCar rdf:resource="#GolfIV"/>
    <carrental:hasCar rdf:resource="#BMW330" />
    <carrental:hasCar rdf:resource="#E55AMG"/>
  </rdf:Description>
  <rdf:Description rdf:ID="RentACar">
    <carrental:hasCar rdf:resource="#GolfV"/>
    <carrental:hasCar rdf:resource="#Polo"/>
  </rdf:Description>
  <rdf:Description rdf:ID="Polo">
    <carrental:carCategory>A</carrental:carCategory>
  </rdf:Description>
  <rdf:Description rdf:ID="GolfIV">
    <carrental:carCategory>B</carrental:carCategory>
  </rdf:Description>
  <rdf:Description rdf:ID="GolfV">
    <carrental:carCategory>B</carrental:carCategory>
  </rdf:Description>
  <rdf:Description rdf:ID="BMW330">
    <carrental:carCategory>C</carrental:carCategory>
```

```
  </rdf:Description>
  <rdf:Description rdf:ID="E55AMG">
    <carrental:carCategory>F</carrental:carCategory>
  </rdf:Description>
</rdf:RDF>
```

Graph Patterns. The simplest form of a graph pattern is a triple pattern, which basically looks like an RDF triple. The difference is that a graph pattern may contain variables instead of RDF terms in any position. The combination of triple patterns mixed with value constraints yields a basic graph pattern. To fulfill a pattern, an exact match to a graph pattern is needed (cf. [32]).

Basic Syntax. The syntax resembles the *SELECT ... FROM ... WHERE ...* style of the SQL syntax. The basic structure of a SPARQL query consists of a *SELECT* and a *WHERE* part, but is usually expanded with a *FROM* part to declare where the data to be queried is found. Thus, a general SPARQL query looks like the following:

SELECT *requested variables*
FROM *specification of data sources*
WHERE
{
  dot-separated list of conditions, each formulated as triple and/or FILTER expression .
}

Variables. Variables are indicated by a *?* at the beginning. *$* can be used as an alternative to *?*. In the result of a request, each variable assignment that matches the patterns specified in the conditions and therefore fulfills the constraints is returned.

Example 5.2 *A very basic request that looks up all cars that belong to category* A *looks as follows:*

```
SELECT ?car
FROM <http://localhost/exampleontology/carrental.rdf>
WHERE
{
  ?car <http://www.semwebtech.org/domains/2006/carrental#carCategory> "A" .
}
```

The result of this query contains a Polo *as the only car available in category* A *at the two car rental companies:*

car
<#Polo>

Terms delimited by <> are IRI[37] references. The # in front of Polo aroused through the definition of a Polo as *rdf:ID* inside the RDF document, means that Polo is a locally defined and accessible node.

Prefix. To keep a SPARQL query (human) readable, the declaration of namespace prefixes, prefaced with the keyword *PREFIX*, is possible.[38] Thus, the query denoted in Example 5.2 can now be formulated as seen below:

```
PREFIX carrental: <http://www.semwebtech.org/domains/2006/carrental#>
SELECT ?car
FROM <http://localhost/exampleontology/carrental.rdf>
WHERE
{
  ?car carrental:carCategory "A" .
}
```

Value Constraints. To specify the result of a query that is received through graph pattern matching, SPARQL permits to constrain values. Value constraints take the form of boolean expressions and are stated in the query inside the *WHERE* clause, commenced by the keyword *FILTER*.

Example 5.3 *Consider a query which wants to find all "higher class" automobiles, i.e., higher than category* B*:*

```
PREFIX carrental: <http://www.semwebtech.org/domains/2006/carrental#>
SELECT ?car ?category
FROM <http://localhost/exampleontology/carrental.rdf>
WHERE
{
```

[37] Internationalized Resource Identifiers (IRIs) are generalizations of URIs and fully compatible to URLs and URIs. For detailed information see [11].

[38] It is also possible to specify a base namespace by using the keyword *BASE*.

```
  ?car carrental:carCategory ?category .
  FILTER (?category > "B") .
}
```

The above query results in the following cars:

car	category
<#E55AMG>	F
<#BMW330>	C

Optional Pattern Matching. Until now, the entire query pattern has to match to be a solution in basic graph pattern. As there are not always complete structures in RDF graphs, optional matching allows for adding information to a query where it is available, but does not reject the solution if one part in the *OPTIONAL* {*pattern*} part does not match. Note that nested optional patterns are permitted.

Matching Alternatives. Furthermore, SPARQL provides means to combine graph patterns. This makes it possible that one of several patterns may match. In case that more patterns match, all possible solutions are found. The syntax for pattern alternatives is the *UNION* keyword. A usage example looks like *pattern UNION pattern*.

Result Format Results can be considered as a result set or table analogous to the ones shown in the examples above. For each query solution there is one row that may contain empty "cells" if a variable is not bound in a certain solution.

The result sets can be accessed via an API in a supporting programming language and then be serialized into either an RDF graph or XML. An XML format, recommended by the W3C, is called the *SPARQL Query Results XML Format* and is described in [33].

Example 5.4 *This query looks for all companies offering cars in a higher class than* A *but lower than* F*:*

```
PREFIX carrental: <http://www.semwebtech.org/domains/2006/carrental#>
SELECT ?company ?car ?category
FROM <http://localhost/exampleontology/carrental.rdf>
WHERE
{
  ?company carrental:hasCar ?car .
```

```
  ?car carrental:carCategory ?category .
  FILTER (?category > "A") .
  FILTER (?category < "F") .
}
```

The result in XML format looks as follows:

```
<?xml version="1.0"?>
<sparql xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xs="http://www.w3.org/2001/XMLSchema#"
        xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="company"/>
    <variable name="car"/>
    <variable name="category"/>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="company"><uri>#RentACar</uri></binding>
      <binding name="car"><uri>#GolfV</uri></binding>
      <binding name="category"><literal>B</literal></binding>
    </result>
    <result>
      <binding name="company"><uri>#JohnDoeCarRentalService</uri></binding>
      <binding name="car"><uri>#BMW330</uri></binding>
      <binding name="category"><literal>C</literal></binding>
    </result>
    <result>
      <binding name="company"><uri>#JohnDoeCarRentalService</uri></binding>
      <binding name="car"><uri>#GolfIV</uri></binding>
      <binding name="category"><literal>B</literal></binding>
    </result>
  </results>
</sparql>
```

This SPARQL Query Results XML Format can now easily be converted, e.g., into framework-known exchange formats (cf. Section 4.6), by using XSLT stylesheets.

The description given above is just a brief overview of the functionalities of SPARQL. But SPARQL, overall, provides means like most other query languages, for example, SQL, including *ORDER BY*, *DISTINCT* and other constructs that are not mentioned here. For more detailed information about SPARQL see [32], for the XML result format see [33] and [9].

In this chapter the Jena Semantic Web Framework for building Semantic Web applications

has been introduced. Now, all theoretical and programming relevant concepts are present to describe the developed Domain Broker.

6 Domain Brokering

This chapter describes the Domain Broker which has been developed and implemented during this thesis. It is composed of an Event Broker that handles the propagation of events, an Action Broker which is responsible for the execution (in form of forwarding) of actions, and a Query Broker that handles requests exemplary shown with a SPARQL Query Broker. These specific brokers will be introduced in detail in the following sections.

6.1 Event Brokering

In this section, the purpose of Event Brokers and their integration into the framework is introduced.

Event Brokers act as mediators between several nodes of the framework. They are fed with events of their domain by event producing nodes, namely Domain Service Nodes, and forward the events to registered consumers, such as Atomic Event Matchers (AEMs). This cooperation is shown in Figure 6.1 and is described below.

Communication between CED, AEM and Event Broker. The Composite Event Detection Service (CED) (for example, a SNOOP engine) can register an Atomic Event Specification (AES) at an AEM that implements the language corresponding to the CED to enable the detection of composed events (cf. Example 4.1 and Figure 6.1 respectively). To be aware of all relevant events, the AEM, in turn, registers at an Event Broker which serves the required domain. Since an AEM can accept registrations from CEDs of different languages, an AEM itself can also register at Event Brokers of different domains to execute their service. Before the registration at an Event Broker, the AEM has to determine the domain (URI) and optionally the event type of the requested events. With the obtained URI of the domain, the relevant Event Broker can be asked at an LSR (cf. Section 4.8). Then, the AEM tells the Event Broker to forward the events of the received domain, optionally just these of a specific event type. As seen above, a registration at an Event Broker must contain the URI, because the Broker may support more than one domain and optionally an event type.

Of course, not only atomic events as part of composite events can be handled by AEMs. ECA

engines[39], for example, can register AESs at AEMs too, in order to be aware of the occurrence of atomic events. The communication between Domain Services on the one, and AEMs or CEDs on the other hand is depicted in Figure 6.1.

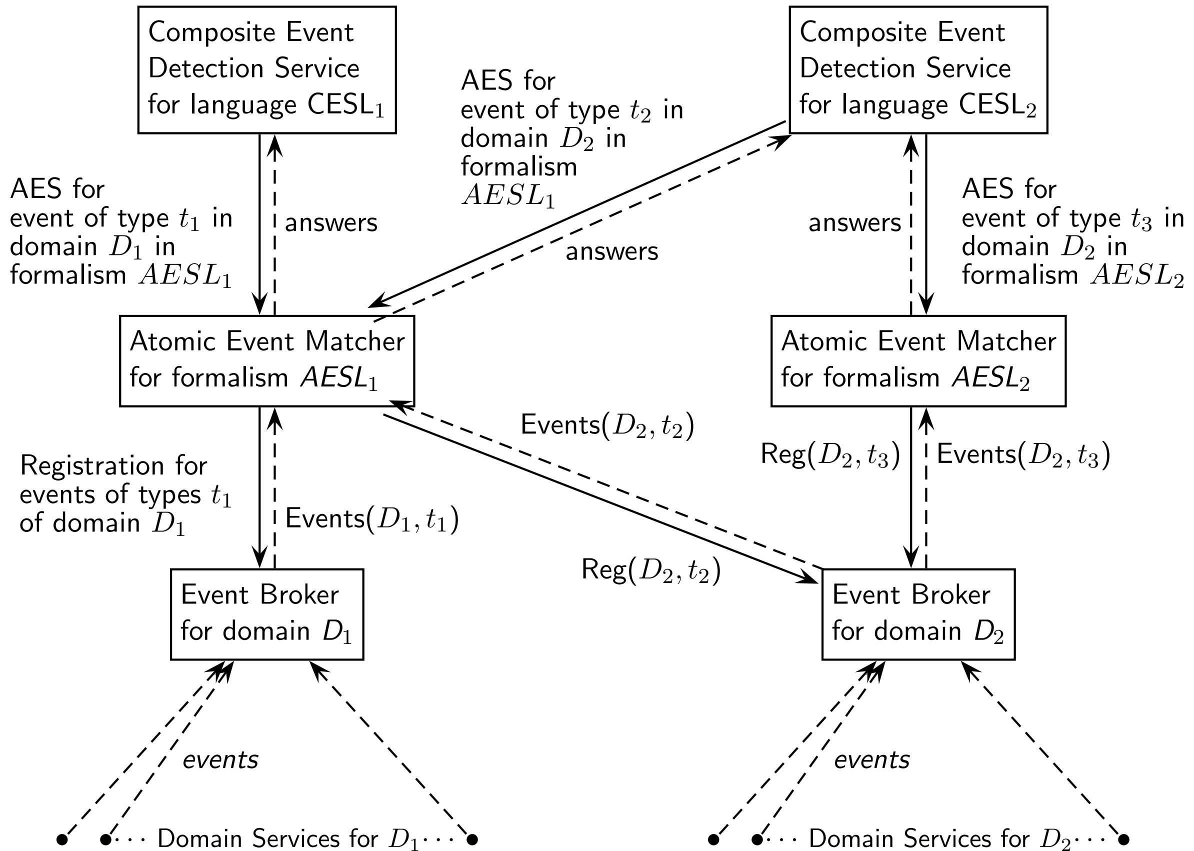

Figure 6.1: Architecture: Communication with Event Brokers (from [13])

As seen above, the nodes which are interested in events of a domain can register at the Event Broker for this domain. Therefore, they have to send registrations marked up in a special XML format to the Service-URL of the Event Broker, managed by a superior Domain Broker:

```
<register>
  <reply-to>URL where the events shall be forwarded to</reply-to>
  <domain>domain-URI</domain>
  <event-type>event type</event-type>
</register>
```

[39] ECA engines are not explicitly mentioned in Figure 6.1.

If a consumer, e.g. an Atomic Event Matcher, wants to be informed about all events of a domain, the <event-type> element can be omitted. The <reply-to> URL could optionally be communicated to the Event Broker via HTTP header.[40]

Which Event Broker is responsible for the respective domain is available at an LSR (cf. Section 4.8). These provide also the URL of the service for the registration.

In the future, there will exist services which give answers in an appropriate format, e.g., for the request of an Atomic Event Matcher which Event Broker is responsible for the *travel* domain.

The following example clarifies the process of registration and deregistration at an Event Broker.

Example 6.1 (Registration at the Event Broker) *Consider an AEM which gets the task to detect the following event type (comparable to Example 4.1):*

```
<travel:canceled-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel" />
```

It determines the fact that the root element is from the travel *domain and, moreover, specifies the event type* canceled-flight*. Thus, the AEM requests an LSR which Event Broker supports the* travel *domain. Afterwards the AEM sends a registration XML document of the following form to the Event Broker:*

```
<register>
  <reply-to>http://www.semwebtech.org/aem/2006/services/aem/receive-events</reply-to>
  <domain>http://www.semwebtech.org/domains/2006/travel</domain>
  <event-type>canceled-flight</event-type>
</register>
```

Now, the Event Broker forwards all events that match the given domain and event type to the given reply-to *URL.*

If a consumer has no interest in getting events anymore, it can certainly deregister at the Event Broker in the same way as it applied, except for replacing the root element by <deregister>:

```
<deregister>
  <reply-to>http://www.semwebtech.org/aem/2006/services/aem/receive-events</reply-to>
  <domain>http://www.semwebtech.org/domains/2006/travel</domain>
  <event-type>canceled-flight</event-type>
</deregister>
```

[40] This has not been implemented in the prototype yet.

It should be clear that the Event Broker needs book keeping to keep track of who is registered for which event. If no node is interested in an incoming event, the Event Broker can omit it.

But not only *consuming* clients, e.g., *AEMs* which want to be informed about events of a special domain, use the services of an Event Broker. In a domain exist many nodes which produce events. These Domain Services can be flight or railway companies, for example. The event *producing* nodes send their generated events not directly to the consumer (e.g., some travel agencies) but to the Event Broker of the appropriate domain where the event fits in.

Brokering of Derived Events. Inside the Framework for Evolution and Reactivity in the Semantic Web, not only atomic events occur. It is possible that the espousal of one event triggers, under certain conditions, another event. These derived events are defined through (composite) event specifications with an optional query and/or a test part.

There exist event types that are directly supported by certain domain nodes which have their own rule definitions. But also event specifications occur, for which the ontology provides global ECE derivation rules (cf. [13]).

Example 6.2 *Flight companies, for example, could be interested in events that inform them about the utilization of their machines to adjust their travel prices contemporarily. Therefore, a rule can be stored wrt. an ontology that defines the event* half-booked*:*

```
<world:definition xmlns:world="http://www.semwebtech.org/domains/2006/world"
    annotation="half-booked definition">
  <world:defined syntax="xml" xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml">
    <travel:half-booked to="$To" from="$From" date="$Date" flight="$Flight"
       xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </world:defined>
  <world:defined-as xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml"
      syntax="xml">
    <eca:event xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
      <travel:booking person="{$Person}" to="{$To}" from="{$From}" date="{$Date}"
        flight="{$Flight}"/>
    </eca:event>
    <eca:variable name="Seats">
      <eca:query>
        query the number of seats available for flight $Flight
      </eca:query>
    </eca:variable>
    <eca:variable name="BookedSeats">
      <eca:query>
        query the number of seats already booked for flight $Flight
```

```
      </eca:query>
    </eca:variable>
    <eca:test>
      <eca:input-variable name="Seats"/>
      <eca:input-variable name="BookedSeats"/>
      <eca:opaque lang="http://www.w3.org/XPath">
        <![CDATA[
          number($BookedSeats) + 1 >= number($Seats) div 2
        ]]>
      </eca:opaque>
    </eca:test>
  </world:defined-as>
</world:definition>
```

During the initialization of the Event Broker, this definition is sent in a framework-aware format (see Section 4.5.2) to the ECA engine[41] *which registers the rule. The ECA-engine, in turn, informs an AEM to be aware of the relevant events which, again, applies for* booking *events of the* travel *domain at the Event Broker. While the occurrence of a booking, the ECA engine evaluates the rule and causes the propagation of the* half-booked *event if the test part is fulfilled.*

Optional and Additional Functionality of Event Brokers. An Event Broker can also offer services usually provided by AEMs. As every service registers at an LSR, a CED searching for functionality, e.g., matching of some AES, would then get an appropriate Event Broker as its service provider.

Not every domain node is able to publish events. Therefore, an Event Broker could be used for polling issues: The Event Broker could monitor the considered resources and apply *continuous query event (CQE) rules* to get events. A use case of polling Event Brokers is the RSS-based Event Brokering. This results in raising of events which are obtained, for example, from bioinformatics services (cf. [13]).[42]

The brokering of actions is described in the next section.

6.2 Action Brokering

Action Brokering can be compared to Event Brokering. The difference, as the name already says, lies in the handling of actions. It has to be distinguished where the action request takes

[41] The ECA engine has to be adjusted in the next version to be able to handle the format for ECE rules.
[42] Note that in the developed prototype these functionalities are not implemented.

place. This can either be at a certain domain node, which will basically happen for opaque actions, e.g., for updating data. The other possibility is the use of an Action Broker to distribute actions to relevant nodes.

This chapter focuses on the latter option[43]. In principle there are two ways to implement the distribution of action requests (cf. [13]):

Action Forwarding via Broadcast. First, the requested action is broadcast to all potential nodes of the domain. The rough procedure is the same as in the main Event Broker task: The ontology inside the Action Broker is asked who supports[44] the incoming action[45] request. After the list of nodes is collected, the action is broadcast to all supporting nodes. Then, the node that receives the action, e.g., an airline company, has to decide what to do. It is naturally that only very few or just one node is really interested in executing this action. Nevertheless, this option is realized as a first step in the prototype.

Data-Dependent Action Forwarding. The second opportunity forwards the action only to the relevant nodes, but this requires more information from the ontology, i.e., *data* and not "only" *metadata*. If the ontology contains specifications, for example, which airline offers which flight, this information can be requested by the Domain Broker through a DSR. In this case, the action is just forwarded to these nodes.

Example 6.3 *Consider a railway company that detects huge technical problems at Hamburg Central Station. The responsible compartment decides to delay all arriving trains for 2 hours, expressed by a rule, e.g.:*

```
<eca:rule xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <eca:event>technical problem detected</eca:event>
  <eca:query>all $trains arriving in the next hour</eca:query>
  <eca:action>
    <travel:delay-train code="{$train}" delay="2h">
      <travel:reason>technical problem</travel:reason>
    </travel:delay-train>
  </eca:action>
</eca:rule>
```

The affected trains are caught in the <eca:query> *part, for example, in a request at a Query Broker. Then, the results are bound to the variable* $train. *One following action instance could*

[43] This option is also implemented in the Action Broker.
[44] That is *world:supports*.
[45] To be declared in the ontology as *world:Action*.

look like this:

```
<eca:action>
  <travel:delay-train code="DB678" delay="2h">
    <travel:reason>technical problem</travel:reason>
  </travel:delay-train>
<eca:action>
```

To ensure that the action shown above is indeed an action, the underlying ontology has to contain a triple of the form

<travel:delay-train, rdf:type, world:Action> .

Of course, this action only interests the *Deutsche Bahn* which operates the train *DB678*. But this information the user gets through a “sharp look” at the code of the train has to be extracted somehow from the ontology. This, in return, involves not only the use of *metadata*, but *data*. Hence, the ontology has to specify the responsible nodes through rules. This has to be done for each action:

```
(?A, has-relevant-node, ?RailwayCompany) :-
    (?A, rdf:type, travel:delay-train),
    (?A, talks-about, ?Train),
    (?Train, travel:operated-by, ?RailwayCompany).
```

To acquire the information represented by the triple (?Train, travel:operated-by, ?RailwayCompany), different realization approaches are imaginable. To get the information, either a Service Registry has to be asked which railway company runs this special train, or *all* nodes have to be asked if they operate the train, or the required piece of information must be deposited in a Domain Broker. Thus, a Domain Broker with a domain-dependent knowledge base would be useful (cf. [13]).

Usually, a domain ontology, e.g., for the *travel* domain, contains expressions which node *world:supports* a *world:Action*. Thus, the following statement could be part of the *travel* ontology:

<http://lh.com/, world:supports, travel:delay-flight>

If just atomic actions were handled by Action Brokers and every appropriate service was mentioned in the ontology as seen above, no registrations at dedicated brokers would be needed to get desired actions forwarded. But, for example, through ACA rules it is possible to add new notions to an ontology.

Brokering of Complex Actions via ACA rules. How a new notion can be contained in an ACA rule, is shown in the following example.

Example 6.4 *The booking of a* return ticket from X to Y on date A and B *will yield the booking of a* ticket from X to Y on date A *and a* ticket from Y to X on date B.

```
<world:definition annotation="return ticket definition">
  <world:defined syntax="xml">
    <travel:book-return-ticket dateA="{$dA}" dateB="{$dB}" from="{$from}" to="{$to}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </world:defined>
  <world:defined-as syntax="xml">
    <travel:book-flight dateA="{$dA}" from="{$from}" to="{$to}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
    <travel:book-flight dateB="{$dB}" from="{$to}" to="{$from}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </world:defined-as>
</world:definition>
```

Therefore, means have to be provided by an Action Broker to register or deregister for action types. During the initialization of the Action Broker, the rule which defines a return ticket is sent for registration to an appropriate service engine. The rule is converted into the following format[46]:

```
<eca:aca-rule>
  <eca:define-action>
    <travel:book-return-ticket dateA="{$dA}" dateB="{$dB}" from="{$from}" to="{$to}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </eca:define-action>
  <eca:action>
    <travel:book-flight dateA="{$dA}" from="{$from}" to="{$to}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </eca:action>
  <eca:action>
    <travel:book-flight dateB="{$dB}" from="{$to}" to="{$from}"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel"/>
  </eca:action>
</eca:aca-rule>
```

The service engine (an ECA engine or another appropriate service), in turn, registers for the action type (*book-return-ticket*) of the given domain (*travel*) at the Action Broker.

[46] The ECA engine has to be adjusted in the next version to be able to handle the format.

The registration, marked up the same way as for Event Broker registrations (see Section 6.1), is then sent to a dedicated Action Broker of the domain, e.g. to
http://www.semwebtech.org/domains/2006/domain-broker/action-broker/register-for-action.

Mapping from XML to RDF. Until now, only XML events resp. actions have been considered. If RDF URIs are used, (somehow) a mapping from XML to RDF has to take place. It has to be decided where the actual mapping is performed.

This can either happen at the Domain Broker or at the relevant domain nodes. For the first possibility, the ontology has to contain an ACA rule that specifies how the action is mapped onto RDF level. Then, the action, comparable to the above-mentioned, has to be broadcast to all nodes that support the action or is just sent to the relevant ones by using specific data.

The second option is to broadcast the action to all nodes that are mentioned in the ontology to support it. Then, the mapping is executed at the appropriate domain nodes (cf. [13]).

As mentioned above, besides an Event- and Action Broker, a Domain Broker consists also of a Query Broker which is introduced in the following section.

6.3 Query Brokering

Clients in the Semantic Web are not only interested in getting events of a certain domain. In fact, they want to get information for individual purposes. Thus, the functionality of queries is needed for the framework to fulfill the necessities of the nodes integrated in the framework.

Queries can be stated against single domain nodes, which mainly makes sense for opaque queries. In addition, Domain Brokers should allow for querying data sources of their domain distributed over several nodes in the network. Acting as mediators, they accept a query, process it, and return the answer. Therefore, Domain Brokers have the acquirements of the dedicated ontology, including RDF and OWL statements and the corresponding rules.

The so far basically drafted procedure can be adopted in the framework using, e.g., different algorithms to achieve and integrate the required information or to unite the diverse probably different data models. To get a working application, a simple approach of a Query Broker is realized first. This is done by the implementation of a Query Broker that is able to operate on SPARQL queries.

Example 6.5 *Consider a travel agency that wants to know all possible connections from Göttingen to Paris and how often the means of transport have to be changed*[47]*. Hence, the* transitive

[47] For example, from train to train or train to airplane, etc.

closure *is required. Therefore, all connections with all possible means of locomotion in the underlying ontology, e.g. train, aircraft and ship, must be considered. That the notion* Connection *is indeed a transitive closure has to be contained in the ontology. Thus, the Query Broker has to ask every node supporting* travel:Connection *for its connections, combine them and extract all connections starting in Göttingen and ending somehow, i.e., without cycles, in Paris.*

To clarify the process a query request runs through, the single steps are accompanied by an example from the *travel* domain mentioned above. Imagine a strongly simplified ontology that takes only trains and airplanes into consideration. To keep the example simple[48], our ontology consists only of five connections, operated by different companies (two railway and one flight company):

- Göttingen - Hamburg
- Hamburg - Bremen
- Hamburg - Paris
- Hamburg - München
- München - Paris

To reduce the complexity of the RDF statements and to keep the RDF graph concise, all resources (i.e., cities, trains and flights) are identified via URIs. Cities are accessed via the *mondial (mon)* protocol, trains via *train* and airplanes through *airplane* respectively. Thus, mon://country/de/city/Hamburg represents the German city named "Hamburg" in this example.

Since the nodes in the Semantic Web are distributed and ontologies are mostly divided over several resources as well, the travel ontology supported by three companies[49] in this example is partitioned, too. The following RDF statements show the individual travel services of each company, taken from different nodes:

```
<travel:Train rdf:about="train://db/ICE123/">
  <travel:from rdf:resource="mon://country/de/city/Goettingen"/>
  <travel:to rdf:resource="mon://country/de/city/Hamburg"/>
  <travel:connected-by>Train</travel:connected-by>
</travel:Train>

<travel:Train rdf:about="train://nwb/nwb456/">
```

[48] To clarify, how a general query is answered.
[49] Namely NWB, DB and LH.

```
  <travel:from rdf:resource="mon://country/de/city/Hamburg"/>
  <travel:to rdf:resource="mon://country/de/city/Bremen"/>
  <travel:connected-by>Train</travel:connected-by>
</travel:Train>

<travel:Flight rdf:about="airplane://lh/lh461/">
  <travel:from rdf:resource="mon://country/de/city/Hamburg"/>
  <travel:to rdf:resource="mon://country/fr/city/Paris"/>
  <travel:connected-by>Airplane</travel:connected-by>
</travel:Flight>
<travel:Flight rdf:about="airplane://lh/lh789/">
  <travel:from rdf:resource="mon://country/de/city/Hamburg"/>
  <travel:to rdf:resource="mon://country/de/city/Muenchen"/>
  <travel:connected-by>Airplane</travel:connected-by>
</travel:Flight>
<travel:Flight rdf:about="airplane://lh/lh444/">
  <travel:from rdf:resource="mon://country/de/city/Muenchen"/>
  <travel:to rdf:resource="mon://country/fr/city/Paris"/>
  <travel:connected-by>Airplane</travel:connected-by>
</travel:Flight>
```

Each of the above statements that belong together[50] can be visualized[51] by an RDF graph, depicted in Figure 6.2, which has not yet a connection to the other graphs.[52]

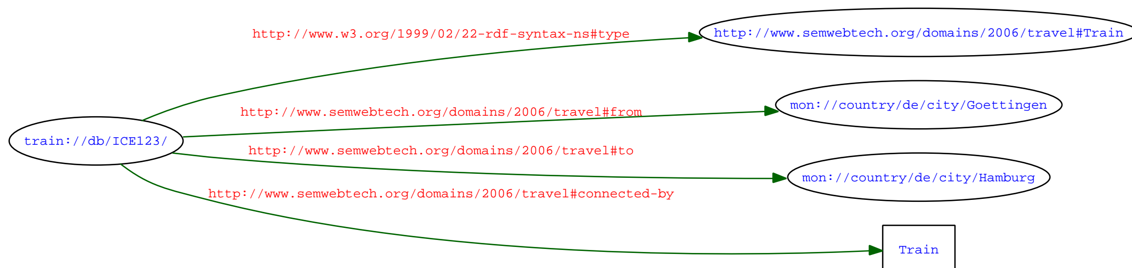

Figure 6.2: RDF Graph of a Travel Route offered by DB

6.3.1 Request Format

Requests sent to a Query Broker via the superior Domain Broker have to be marked up in XML format. To allow a Query Broker to handle the request, a query has to be of the following

[50] Here, the statements of each company, i.e., DB, NWB and LH belong together.
[51] The visualization of the graph was created with the W3C RDF Validation Service [40].
[52] Note that there is an implicit connection through identical URIs in the isolated RDF documents, but this connection has not been "established" in the Query Broker by now.

form:

```
<query>
  <opaque lang="language specification">
    query in opaque query language
  </opaque>
  <reply-to>URL where the answer is sent to</reply-to>
</query>
```

The handling of a query within the Query Broker shall be made clear as a showcase on the basis of the following request. The query asks all connections of the travel domain that start in Göttingen and end in Paris. Furthermore, it is questioned how often the means of transport have to be changed.

```
<query>
  <opaque lang="http://www.w3.org/2005/01/sparql-protocol#">
    PREFIX travel:  <http://www.semwebtech.org/domains/2006/travel#>
    SELECT ?from ?to ?changes
    WHERE {
      ?a ?b travel:Connection .
      ?a travel:from ?from .
      ?a travel:to ?to .
      ?a travel:from <mon://country/de/city/Goettingen>.
      ?a travel:to <mon://country/fr/city/Paris>.
      ?a travel:changes ?changes .
    }
  </opaque>
  <reply-to>reply-to URL of the asking node</reply-to>
</query>
```

6.3.2 Query Decomposition

To answer the query, all static notions occurring in the request, i.e., concepts and properties, have to be collected. Relevant notions are not only directly mentioned in the query, but also implicitly through

- owl:equivalentProperty,
- owl:inverseOf,
- owl:equivalentClass ,and so on,

which must also be considered. Furthermore, if the ontology contains rules of the form *head* $\leftarrow$ *body* (or *body* $\rightarrow$ *head* respectively), they also have to be taken into account. If a request

queries the head of a rule, notions in the body have to be considered. Thus, the rules have to be denoted in an appropriate format, e.g. RuleML, to be able to extract and apply the needed rules.
The concepts *Connection* and *changes* are never mentioned directly in the ontology of the *travel* domain. The decomposition of the query yields a set of relevant concepts and properties that have to be incorporated: *Connection, from, to* and *changes*. Furthermore, the ontology contains the following rules[53], which have also be taken into account:

```
[rule1: (?s1 rdf:type travel:Train) (?s1 travel:from ?a) (?s1 travel:to ?b)
  makeTemp(?s2)
→ (?s2 rdf:type travel:Connection) (?s2 travel:from ?a) (?s2 travel:to ?b) (?s2 travel:changes 0)]

[rule2: (?s1 rdf:type travel:Flight) (?s1 travel:from ?a) (?s1 travel:to ?b)
  makeTemp(?s2)
→ (?s2 rdf:type travel:Connection) (?s2 travel:from ?a) (?s2 travel:to ?b) (?s2 travel:changes 0)]

[rule3: (?s1 rdf:type travel:Flight) (?s1 travel:from ?a) (?s1 travel:to ?b)
  (?s2 rdf:type travel:Connection) (?s2 travel:from ?b) (?s2 travel:to ?c) (?s2 travel:changes ?d)
  notEqual(?s1, ?s2)
  addOne(?d, ?e)
  makeTemp(?st)
→ (?st rdf:type travel:Connection) (?st travel:from ?a) (?st travel:to ?c) (?st travel:changes ?e)]

[rule4: (?s1 rdf:type travel:Train) (?s1 travel:from ?a) (?s1 travel:to ?b)
  (?s2 rdf:type travel:Connection) (?s2 travel:from ?b) (?s2 travel:to ?c) (?s2 travel:changes ?d)
  notEqual(?s1, ?s2)
  addOne(?d, ?e)
  makeTemp(?st)
→ (?st rdf:type travel:Connection) (?st travel:from ?a) (?st travel:to ?c) (?st travel:changes ?e)]
```

6.3.3 Required Nodes Selection

After the decomposition has taken place, the ontology or better a Service Registry is asked who *world:supports* the ascertained important notions and concepts. The request results in an answer which contains all distributed single models that were mentioned above. In this example, these are the models represented by the RDF statements depicted above, namely the services offered by DB, NWB and LH.

[53] The rules are described in a format known from the Jena Framework (cf. Section 5.2).

6.3.4 Data Integration

After the relevant nodes have been discovered, the (germane parts of the) data of the affected ontologies are combined. In this example, the union of the three received single models has to be built. The resulting graph is visualized[54] in Figure 6.3.

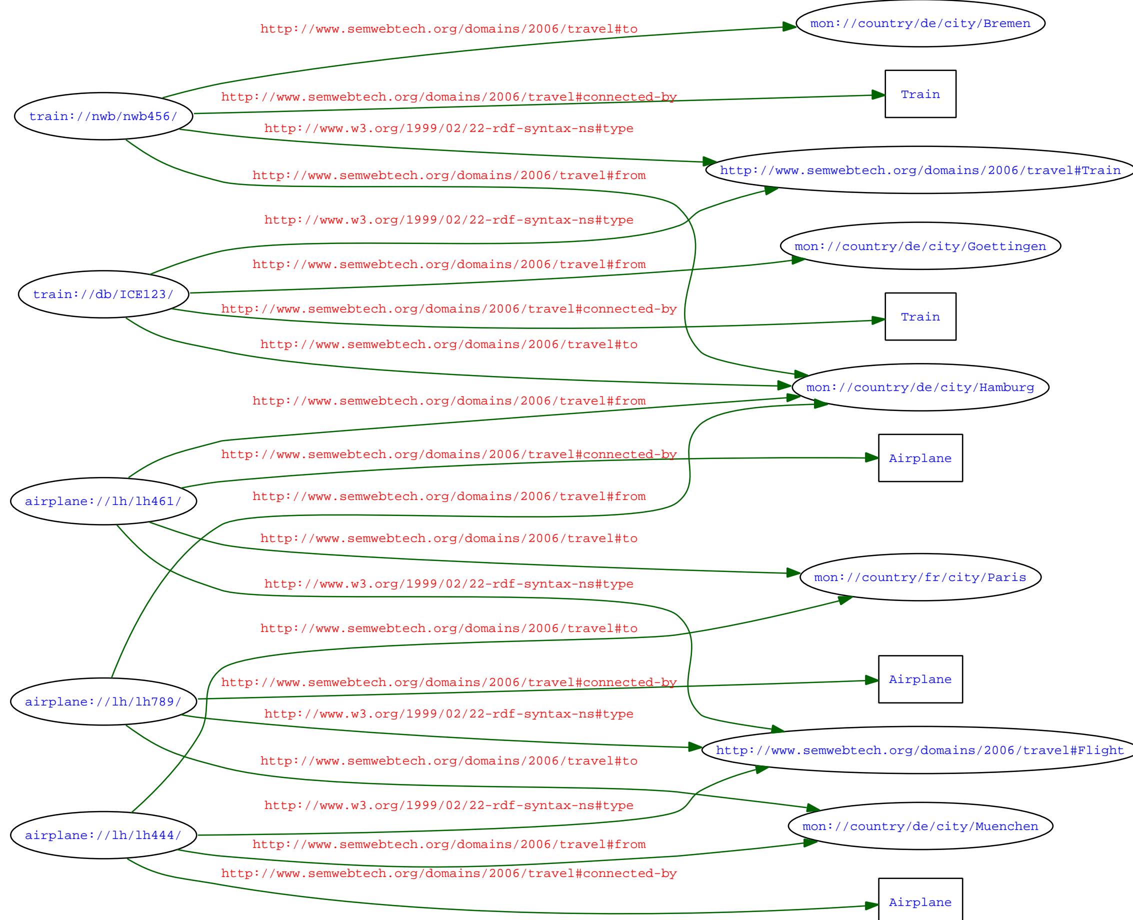

Figure 6.3: United Ontology

The integrated data model is now available at the Query Broker. Before the query could be stated against the model, an inferred model has to be built. This considers the above mentioned rules to be able to extract *travel:Connection* information.

Now, the query is stated against the local model as a knowledge base and the complete answer is returned. Another possibility would be to query the obtained relevant classes and properties (as RDF triples) and take the union of the separated answers to solve the original

[54] The visualization of the graph was created with the W3C RDF Validation Service [40].

query. Note that this would lead to a more complex approach, for example, for a request which requires a transitive closure as seen in Example 6.5.

6.3.5 Query Answering

To get the desired answer of the query, the rules have been applied to the integrated model first. The query was stated against the inferred model and the result was received. As the request was formulated as an opaque SPARQL query, the intermediate result is of the following form:

```
<?xml version="1.0"?>
<sparql
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xs="http://www.w3.org/2001/XMLSchema#"
    xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="from"/>
    <variable name="to"/>
    <variable name="changes"/>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="from">
        <uri>mon://country/de/city/Goettingen</uri>
      </binding>
      <binding name="to">
        <uri>mon://country/fr/city/Paris</uri>
      </binding>
      <binding name="changes">
        <literal datatype="http://www.w3.org/2001/XMLSchema#int">2</literal>
      </binding>
    </result>
    <result>
      <binding name="from">
        <uri>mon://country/de/city/Goettingen</uri>
      </binding>
      <binding name="to">
        <uri>mon://country/fr/city/Paris</uri>
      </binding>
      <binding name="changes">
        <literal datatype="http://www.w3.org/2001/XMLSchema#int">1</literal>
      </binding>
    </result>
```

```
  </results>
</sparql>
```

Afterwards, the response has to be transformed from the SPARQL Query Results XML Format (see [33]) to a format for basic interchange of variable bindings (cf. Section 4.6). Hence, the query used in this example results in the XML document shown below:

```
<logic:variable-bindings>
  <logic:tuple>
    <logic:variable name="from">mon://country/de/city/Goettingen</logic:variable>
    <logic:variable name="to">mon://country/fr/city/Paris</logic:variable>
    <logic:variable name="changes">2</logic:variable>
  </logic:tuple>
  <logic:tuple>
    <logic:variable name="from">mon://country/de/city/Goettingen</logic:variable>
    <logic:variable name="to">mon://country/fr/city/Paris</logic:variable>
    <logic:variable name="changes">1</logic:variable>
  </logic:tuple>
</logic:variable-bindings>
```

This XML document[55] is finally sent back to the URL that was mentioned in the <reply-to> element of the query.

6.4 Miscellaneous

Until now, each domain disposes about one Domain Broker. Since the Semantic Web is very dynamic, it is not debarred that different Domain Brokers with, for example, several Event Brokers will be established in the future. This extension would lead to a huge problem[56]: An event, generated at some node in the web, would be sent to several event brokers of the relevant domain. Then, each event broker would forward this event to the applied nodes. Hence, these nodes or some other framework service would have to decide if the event has to be handled, i.e., if the received event is actually a new one.

The issue has to be solved, how events, actions, etc. can be made unique and, thus, be distinguished by domain nodes or framework services.

As the theoretical description of a Domain Broker with its use of an Event-, Query-, and Action Broker has been given above, the next chapter deals with the prototypical implementation of the Domain Brokering Service.

[55] As here only sets of tuples are treated, the elements *answers*, *answer* and *result* can be omitted.
[56] This problem is not topic of this thesis and is, therefore, not discussed in detail.

7 Implementation

In this chapter the current implementation of the prototypical Domain Broker for the Semantic Web is introduced. First, an overview of the employed technologies is given. Afterwards, the class structure and the communication interfaces of the components will be described. Finally, the graphical Domain Broker Client will be presented.

7.1 Employed Technologies

The prototypical Domain Broker and its components are implemented in Java. The functionalities are realized as Web Services via HTTP. Some components use the Jena Framework to fulfill their tasks.

Java. Java is an object-oriented programming language. It has been invented and developed by Sun Microsystems. Software written in Java is independent from the underlying hardware as well as from operating systems. This property makes it optimally qualified for the development of a Domain Broker for the Semantic Web which shall be able to run on different systems inside of the heterogeneous Semantic Web. For more information see [35].

Jena Framework. Jena is a Java based framework for building Semantic Web applications. It provides means in form of programming environments for operating RDF, RDFS, OWL, and SPARQL concepts. For details see Chapter 5 and [20].

Web Services with HTTP. The nodes inside the Semantic Web are heterogeneous systems, implemented in diverse programming languages that run on different hardware architectures. The components of the Domain Broker, which are arbitrary framework and thus Semantic Web nodes as well, have to provide their services independent from platforms and programming languages to allow smooth communication. To accomplish these requirements, the communication inside the framework is done by XML exchange via plain HTTP methods.

In the next sections, the architecture and implementation of the Domain Broker and its components will be described in detail.

7.2 General Architecture

The Domain Broker prototype is subdivided into two different kinds of classes. On the one hand there are functionalities which are needed by many or all parts of the prototype such as the handling of XML code. On the other hand each main component[57] provides a special service that has its own needs.

This breakdown is also represented in the package structure of the implemented prototype. All classes are beneath the package org.semwebtech.broker. The commonly used classes are contained in the subpackage common. More specific classes are kept inside packages that have the name of the considered component, e.g., actionbroker for classes wrt. the Action Broker and eventbroker, querybroker, and domainbroker respectively.

7.3 Common Classes

All classes which are used by several classes inside the prototypical Domain Broker of the framework are kept inside the package common. It is substructured in util for general utility classes and the subpackage ontology for ontology depending classes.

7.3.1 Utility classes

The classes inside the packages util (see Figure 7.1)[58] represent helper classes that are useful for several other classes. A very generic class is XMLHelper, which is, among others, used for the serialization and deserialization of XML documents from and to their string representation. It is, for example, used by the Event- and Action Broker to extract the text of elements from XML documents.

The classes XMLRegisterHelper and XMLDeregisterHelper inherit from this class to provide functionalities used for the registration and deregistration at Event- and Action Brokers.

To be able to make registrations within broker components persistent, the class RegistrationsHelper is available. It uses the class RegistrationsHelperXML to serialize and deserialize the content of registration objects within files with the help of FileHelper.

7.3.2 Ontology

Classes within the package common.ontology, depicted in Figure 7.2, implement the concept and means to operate ontologies. The class Ontology forms the basis of this package. It contains

[57] Namely the Domain Broker which consists of an Event-, Action- and Query Broker.
[58] Note that the declaration of getter and setter methods for attributes of all classes are omitted within the whole chapter.

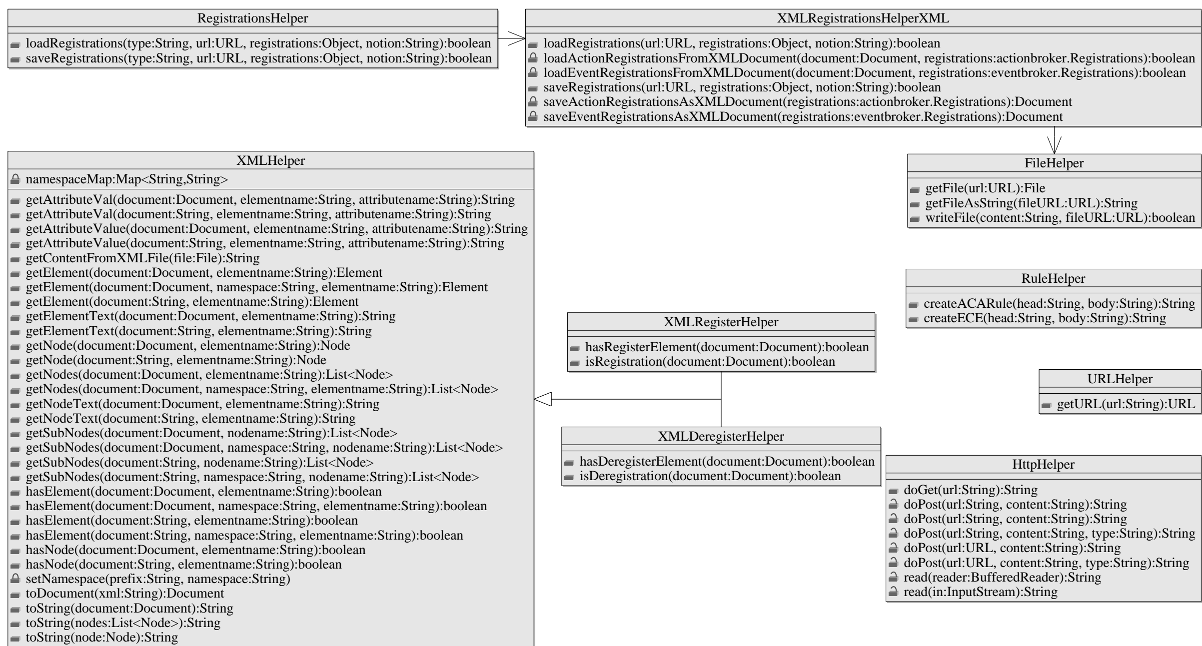

Figure 7.1: Class Diagram for Utility Classes

the N3 [3] representation of an ontology and the ontology-wide rules[59].

It offers methods to get lists of events and actions supported by the ontology. Moreover, the definition of deduction rules and ECE or ACA rules can be achieved through methods of this class. As the ontology can be provided by different nodes inside the framework, the URLs of supporting nodes can also be asked.

To be able to operate with ontology representations written in N3 format, Ontology uses the class JenaHelper to get a model out of the N3 representation.

To keep the class Ontology flexible, it uses OntologyHelper as assistance class. It supports Ontology, for example, in getting the URLs from the special ontology-related N3 representation of the nodes that support a domain.

The classes in this section are used by the classes described in the following sections.

[59] These are logical derivation rules wrt. the ontology as well as ECE and ACA rules as described in Section 3.2.1, Section 4.4.1, and Section 4.4.2 respectively.

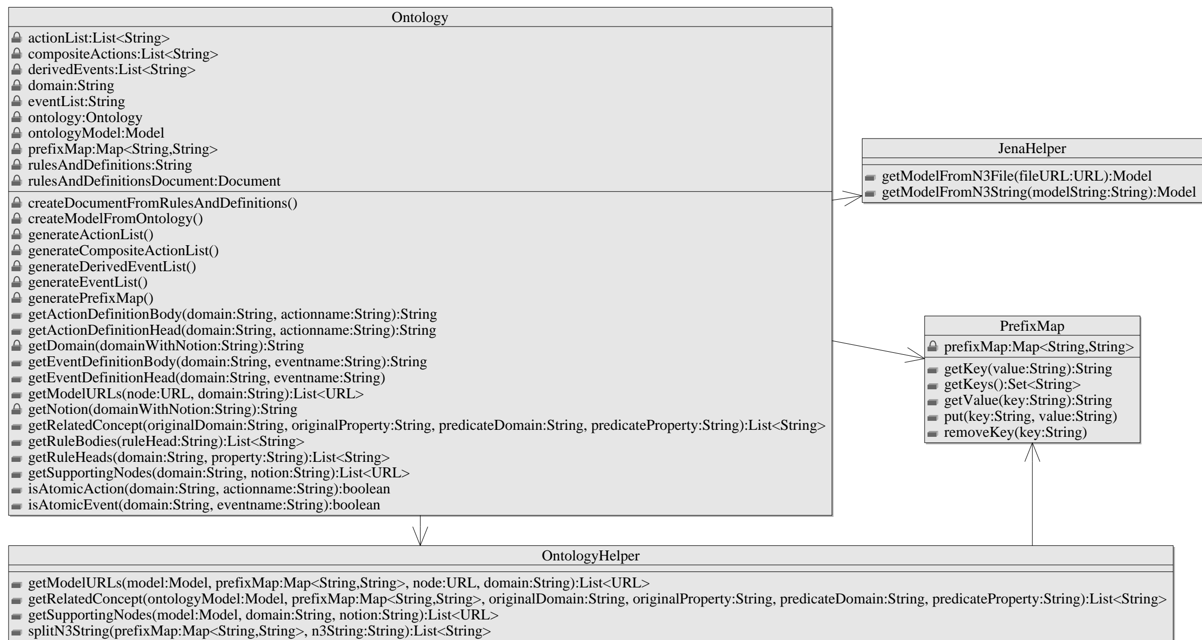

Figure 7.2: Class Diagram for Ontology

7.4 Domain Broker

In this section the communication interface and the parts which a Domain Broker consists of in the prototypical implementation that was developed during this thesis is described.

7.4.1 Communication Interface of the Domain Broker

In this section the interface via which clients can communicate with the Domain Broker is described. It is depicted in Figure 7.3.

Domain Broker Service. The services directly provided by the Domain Broker, namely register-ontology and register-rules-and-definitions are mainly used while the Domain Broker is initialized.

Event Broker Service. With view of the Event- and Action Broker services, clients can be divided into two groups:

- consuming and
- producing nodes.[60]

[60] Note that these sets are not necessarily disjoint.

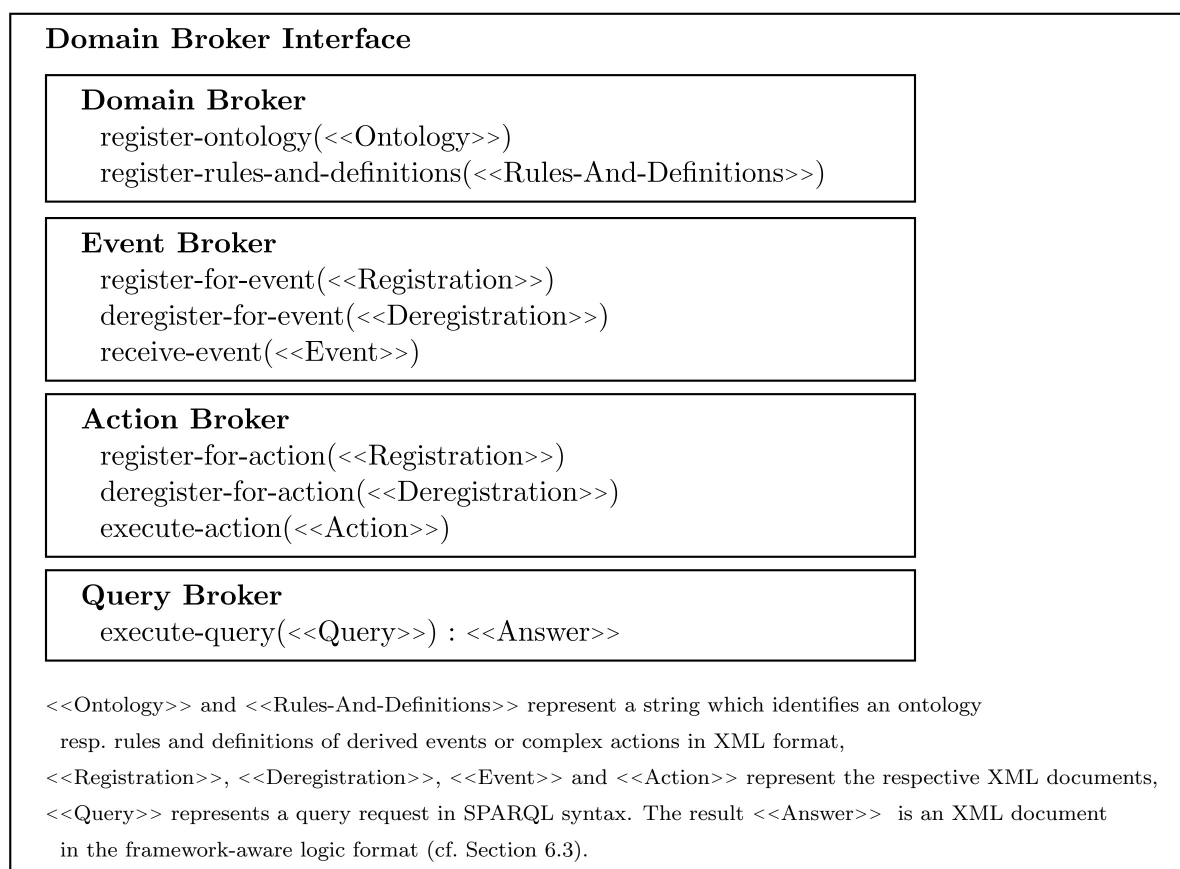

Figure 7.3: Communication Interface of the Domain Broker

From the consumers' point of view, the services register-for-event and deregister-for-event of the Event Broker are the most important ones.

Producers, on the other side, are more interested in the propagation of their events, which makes the services receive-event more interesting for them. All functionalities mentioned above deal with XML fragments.

Example 7.1 *Imagine a client, e.g., a travel agency that wants to register at an Event Broker to be informed about delayed flights. Therefore, it sends the following registration XML document to the dedicated Event Broker mentioned in an LSR:*

```
<register>
  <reply-to>http://www.travel-agency.nop/events</reply-to>
  <domain>http://www.semwebtech.org/domains/2006/travel</domain>
  <event-type>delayed-flight</event-type>
</register>
```

If an airport, as a producing node, in return, announces a delayed flight, the informed Event Broker will forward the following event to the travel agency that applied for it, i.e., http://www.travel-agency.nop/events*:*

```
<travel:delayed-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
  flight="LH123"/>
```

Action Broker Service. The procedure for the broadcasting of actions is nearly the same as for the Event Broker. Nodes that offer support, i.e., the execution of an action in this case, can be looked up in the ontology. Thus, for simple actions no registrations are needed.

In case a node is not mentioned in the ontology to *world:support* a certain action, the node can register for it through the method register-for-action of the interface. To sign off, the function deregister-for-action of the Action Broker is provided. To achieve the execution of an action, the action to be executed has to be sent to the interface method execute-action. Then, the action is forwarded to all nodes that support this action.

During the initialization of the Action Broker, the associated ontology is searched for *complex*[61] actions. The action *banking:money-transfer*, for example, is defined as an ACA-rule within the rules for the *banking* ontology. An Action Broker registers the rule definition at an appropriate engine. In this case a registration of a CCS engine at the Action Broker via the communication interface of the Domain Broker for this action could be the consecution.

Query Broker Service. The service offered by the Query Broker inside the Domain Broker can be used by different kinds of nodes in the framework to achieve information about the considered domain. The (distributed) data sources are queried with a SPARQL engine. Thus, the input of receive-query has to be a SPARQL query delivered as a string enveloped in XML markup that additionally contains a <reply-to> element. The result is sent back to the URL mentioned in the <reply-to> element.

The following example briefly shows the markup of an enveloped SPARQL query and afterwards the framework-aware result format.

Example 7.2 *Imagine a client inside the Semantic Web framework that wants to state a SPARQL query against the Domain Broker. It wraps the query in the following format (cf. Section 6.3.1):*

```
<query>
  <opaque lang="http://www.w3.org/2005/01/sparql-protocol#">
```

[61] For example, composite actions.

```
    query in SPARQL format
  </opaque>
  <reply-to>URL where the query result is expected</reply-to>
</query>
```

The result that the Query Broker achieves is returned (after transformation) in the following XML markup (cf. Section 4.6):

```
<logic:variable-bindings xmlns:logic="http://www.semwebtech.org/lang/2006/logic">
  <logic:tuple>
    <logic:variable name="variable1">variable value</logic:variable>
    <logic:variable name="variable2">variable value</logic:variable>
      ...
  </logic:tuple>
  <logic:tuple>
     ...
  </logic:tuple>
</logic:variable-bindings>
```

7.4.2 Architecture of the Domain Broker

A Domain Broker consists of an Event Broker, an Action Broker, and a Query Broker as depicted in Figure 7.4. These parts are described in detail in the following subsections. The individual broker components are implemented modularly and could also be used autonomously. The implementation according to this thesis uses a showcase Query Broker that can operate SPARQL queries. A Domain Broker takes care of a certain domain, e.g., the *travel* domain. To fulfill the requested tasks, it uses the functionalities of other diverse brokers. It also uses an ontology (see Section 7.3.2) that is forwarded to each broker to ensure that they are able to solve their tasks. The requested tasks reach the Domain Broker via the Communication Interface described in Section 7.4.1. Depending on the type of request, one of the brokers contained in the Domain Broker is taken to handle the operation.

7.4.3 Architecture of the Event Broker

The class EventBroker implements the concept of an Event Broker. It allows for the registration and deregistration for event types of a domain. Furthermore, it is possible to register and deregister rules for derived events.

To handle registrations of nodes and to be informed about events, it uses the class Registrations. To make registrations persistent and, in return, to load registrations the class Registrations provides specific methods.

Figure 7.4: Class Diagram for the Domain Broker

The main function of EventBroker is the forwarding of events. Therefore, it provides the method forwardEvent that uses the class XMLEventHelper.

The above described architecture is depicted in Figure 7.5.

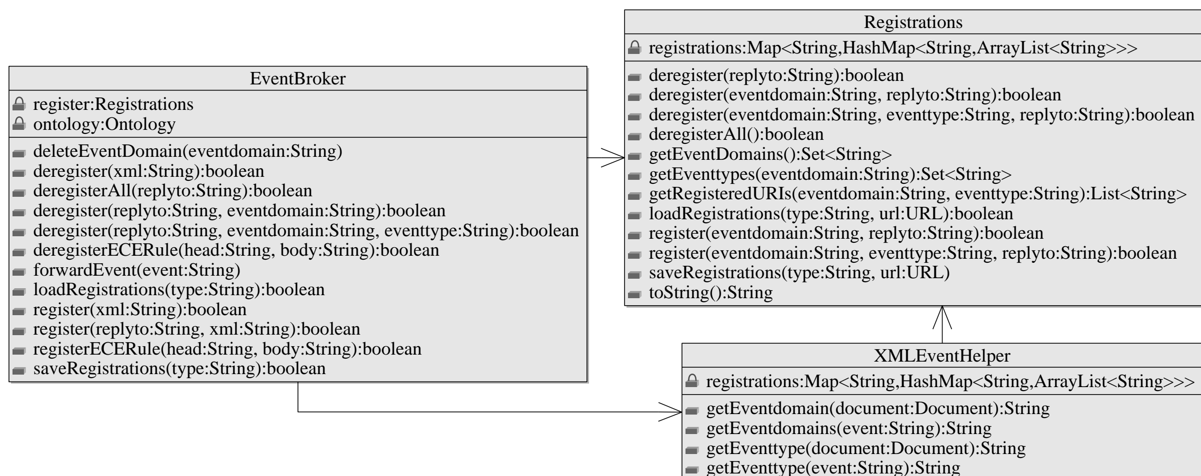

Figure 7.5: Class Diagram for the Event Broker

7.4.4 Architecture of the Action Broker

The architecture of the Action Broker is visualized in Figure 7.6. The main class ActionBroker implements the forwarding of actions that can then be executed at the appropriate nodes. To enable the handling of complex actions, ACA rules can be registered.

As already seen in the Event Broker architecture description, the Action Broker also uses a Registration class that is on the one hand capable of the registration and deregistration of nodes as well as of persistence mechanisms on the other hand.

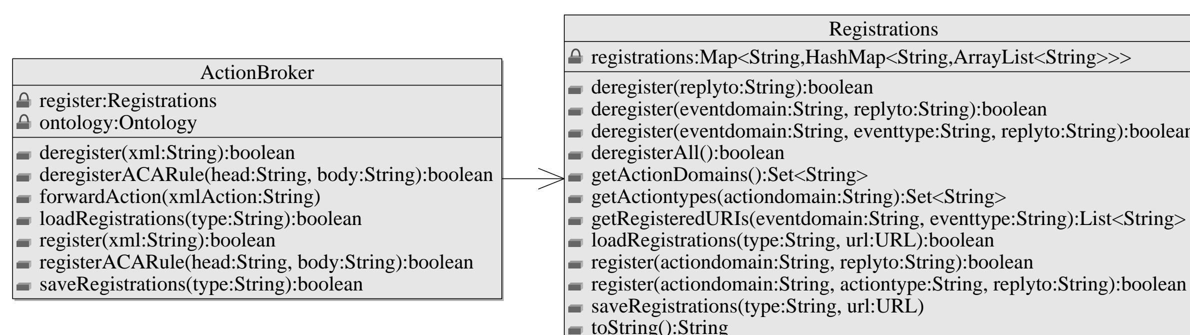

Figure 7.6: Class Diagram for the Action Broker

7.4.5 Architecture of the Query Broker

The structure of the Query Broker is more complex than the above displayed Event- or Action Broker architecture. The central part of the Query Broker in this prototypical implementation is the class QueryBrokerSPARQL inside the package broker.querybroker. To fulfill interface and modularity requirements, QueryBrokerSPARQL inherits from GenericQueryBroker and implements its abstract methods. This permits the expansion of functionality through classes that use another query language, like for example SQL.

To be able to answer a query request, behind the scene QueryBrokerSPARQL uses services of different classes which can be seen in Figure 7.7. The reasoning over the collected models and the following result format transformation is introduced next.

Reasoning over models. To answer a query, the Query Broker has to collect the different data sources that support the relevant concepts. This is managed by the method getAllRelevantConcepts of QueryBrokerSPARQL. Therefore, it uses the class SPARQLHelper that is also in the package querybroker.sparql. It supports the Query Broker to operate the SPARQL request. To retrieve the separate models from different nodes, the class ModelHelper inside the package

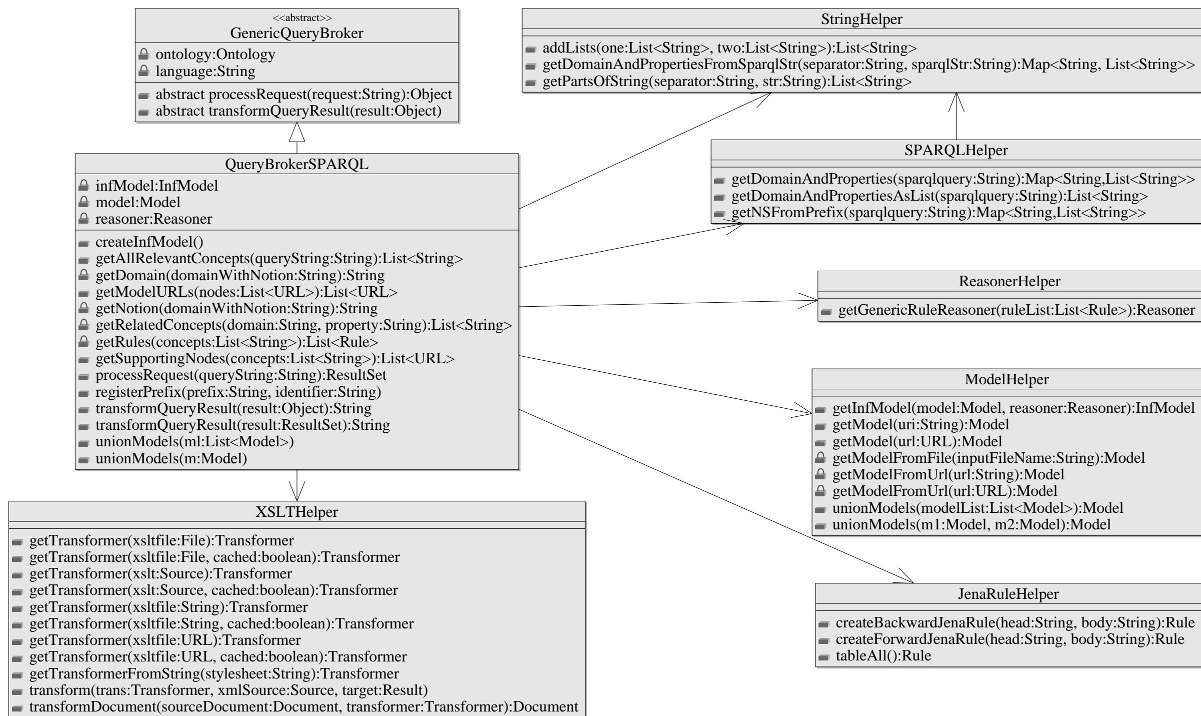

Figure 7.7: Class Diagram for the Query Broker

querybroker.util.jena is utilized. It also supports the union of different models. The class ReasonerHelper in the same package provides, as the name implies, a reasoner. To be able to create a reasoner, the Query Broker can use the class JenaRuleHelper that manages different kinds of rules (see Figure 7.7).

Transformation of result format. A SPARQL query can result in different formats. The one used in this implementation is the SPARQL Query Results XML Format [33]. Of course, the returned result format of a query has to be framework-aware for other query languages will be supported in the future. Thus, the Domain Broker needs means to transform query language specific result formats to global framework accepted formats. This functionality is provided, at least for XML based markups, by the class XSLTHelper from the package domainbroker.util.xslt (see Figure 7.7).

It serves the Domain Broker with supplying transformers and methods to transform one XML document into another.

In the next section, the graphical Domain Broker Client is introduced.

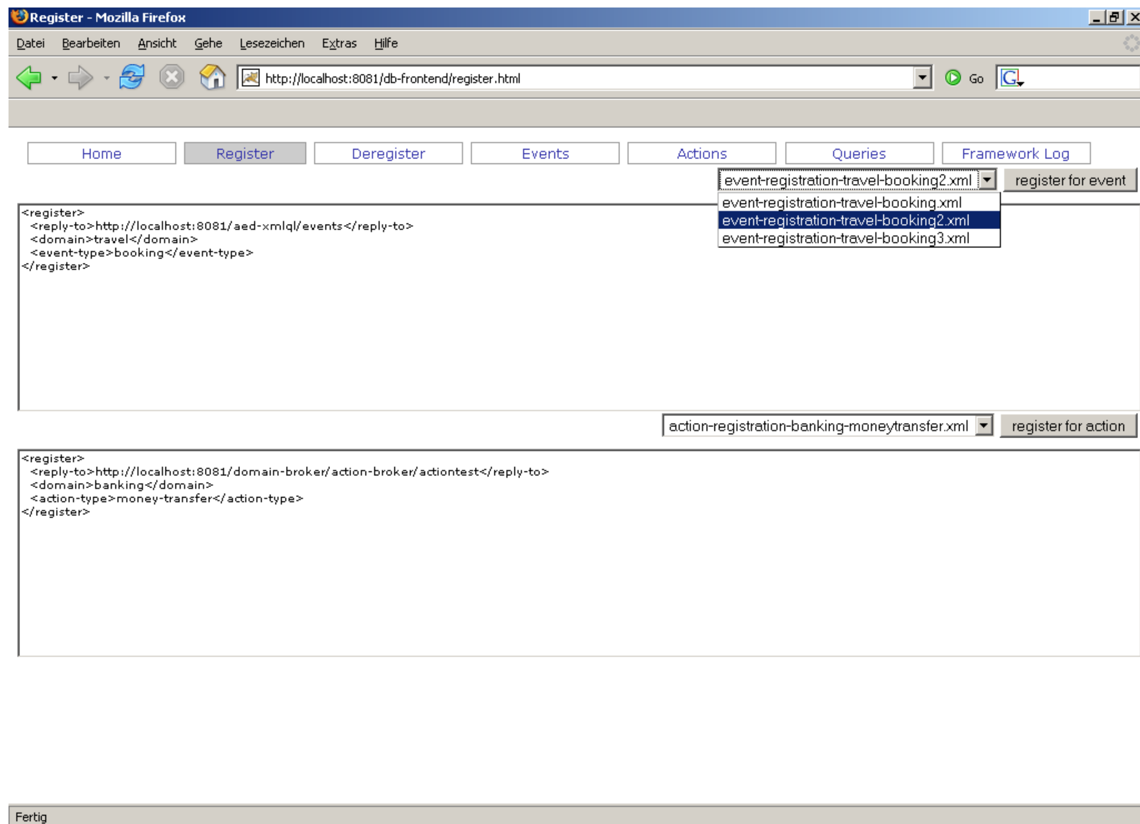

Figure 7.8: Managing Registrations with the Domain Broker Client

7.5 Domain Broker Client

The graphical Domain Broker Client is implemented as an HTML web interface. It is adapted from the graphical ECA engine client[62] to show a consistent layout wrt. the frameworks graphical clients.

The functions of the graphical Domain Broker Client can mainly be summarized by three parts:

1. It allows for the registration (link Register) and deregistration (link Deregister) of events and actions. The requests have to be marked up in the appropriate XML markup and then, the snippets are directly sent to the Domain Broker, i.e., the Event- or Action Broker (see Figure 7.8).

2. The graphical Domain Broker client provides also means to send events (link Events) or

[62] The ECA engine client was developed by Daniel Schubert, see [30].

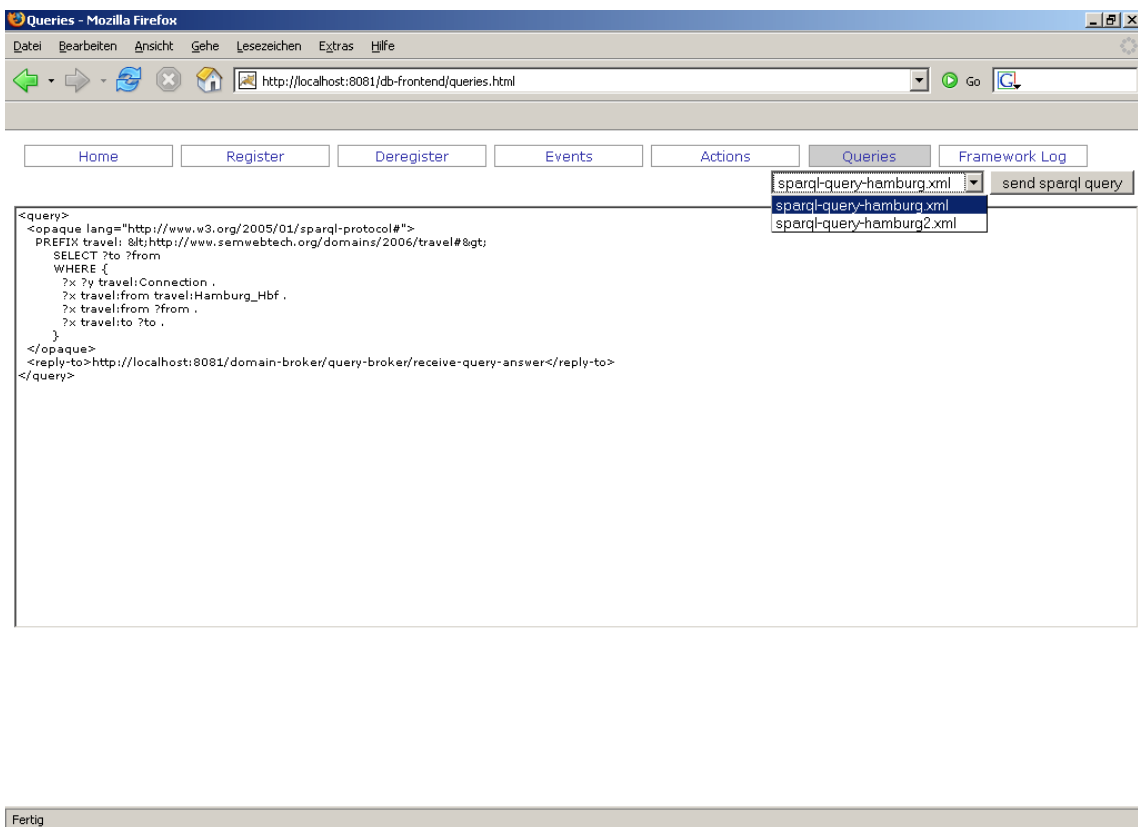

Figure 7.9: Sending SPARQL Queries with the Domain Broker Client

actions (link Actions) to the responsible broker. SPARQL queries can be stated via an interface of the graphical client, too (link Queries) (see Figure 7.9).

3. The output that is generated by the framework-wide logger is presented to the user via the link Framework Log (see Figure 7.10).

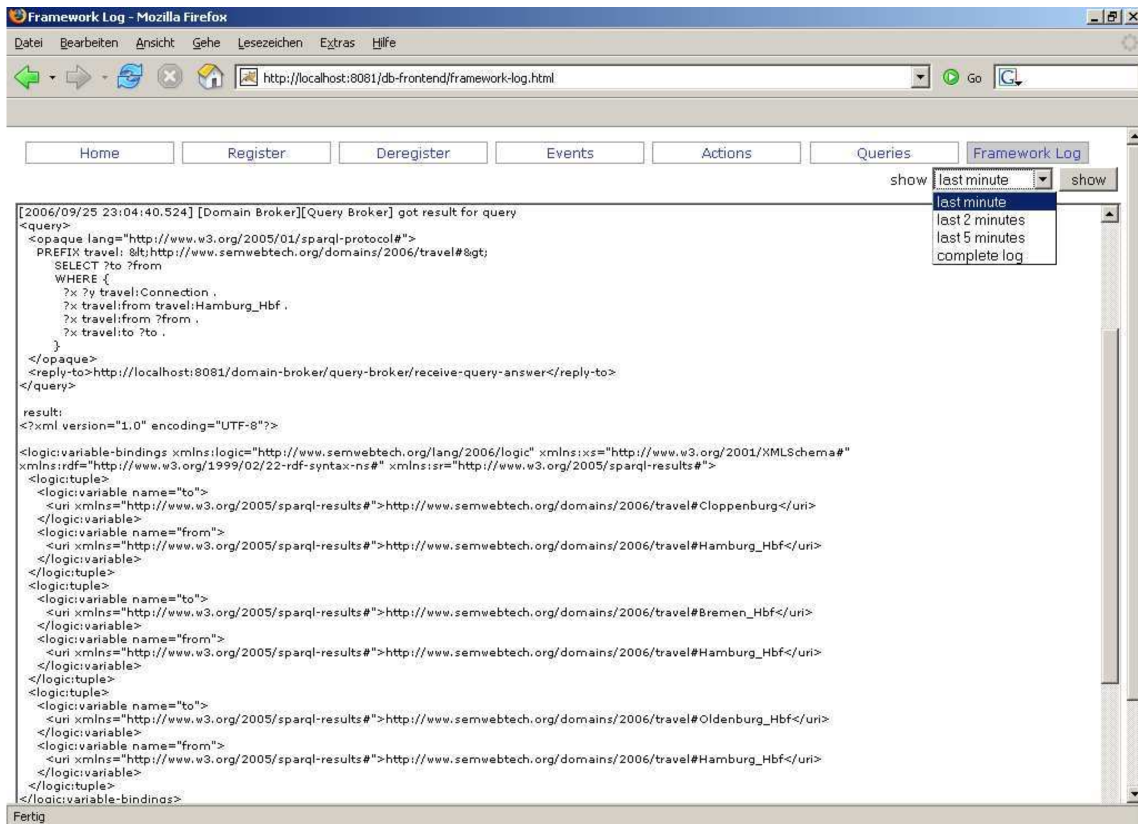

Figure 7.10: Viewing the Framework Log with the Domain Broker Client

8 Conclusion

The development of a Domain Broker builds the basis of this thesis. First, rules and their effect in form of getting implicit knowledge from ontologies have been described. To also integrate dynamic aspects and take events into account, the Framework for Evolution and Reactivity in the Semantic Web, based on ECA rules and ontologies, has been explained.

Since the nodes inside the framework have to react on events in order to execute actions on their specific domains, a Domain Broker was needed to act as a mediator between different nodes of the framework. Consisting of an Event-, Action-, and Query Broker, the Domain Broker was integrated as a module into the Framework for Evolution and Reactivity in the Semantic Web.

The Domain Broker was implemented by using standard web technologies such as HTTP-POST requests. This ensures an easy integration into information systems like the framework and transforms the components of the Domain Broker into active participants of the Semantic Web.

Although the Domain Broker uses several services of the existing framework infrastructure, still important aspects of the framework have not yet been dealt with wrt. their development as modular web services:

- A Language and Service Registry is currently under construction as subject of a thesis and will answer requests about the responsibility of nodes wrt. certain services in the near future.
- At the moment, events, actions, and queries are being simulated by test environments inside the framework components. A possible area for further development will be the establishment of producing and consuming nodes in a certain domain ontology.
- Since some nodes will not be able to send their generated events to dedicated Event Brokers, the Event Broker could be extended with polling functionality.

Bibliography

[1] José Júlio Alferes, Ricardo Amador, Erik Behrends, Mikael Berndtsson, François Bry, Gihan Dawelbait, Andreas Doms, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula Lavinia Pătrânjan, Loic Royer, Franz Schenk, and Michael Schröder. Specification of a Model, Language and Architecture for Evolution and Reactivity. Technical Report I5-D4, REWERSE EU FP6 NoE, 2005. Available at `http://www.rewerse.net`.

[2] Chris Bates. *XML in Theory and Practice*. John Wiley & Sons Ltd, 2003.

[3] Tim Berners-Lee. Notation 3. `http://www.w3.org/DesignIssues/Notation3.html`, 1998.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, May 2001.

[5] Cliff Binstock, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, and Chris Galtenberg. *The XML Schema Complete Reference*. Addison-Wesley, 2003.

[6] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617. Morgan Kaufmann Publishers Inc., 1994.

[7] Wolfgang Dalitz, Winfried Neun, and Wolfram Sperber. Semantic Annotation in Mathematics and Math-Net. In Siegfried Handschuh and Steffen Staab, editors, *Annotation for the Semantic Web*, pages 3–22. IOS Press, 2003.

[8] DAML+OIL (March 2001) Reference Description. `http://www.w3.org/TR/daml+oil-reference`, 2001.

[9] Leigh Dodds. Introducing SPARQL: Querying the Semantic Web. `http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-tutorial.html`, 2005.

[10] Dublin Core Metadata Initiative. http://dublincore.org/, 1994.

[11] Martin Duerst and Michel Suignard. RFC 3987, Internationalized Resource Identifiers (IRIs). http://www.ietf.org/rfc/rfc3987.txt, 2005.

[12] Rainer Eckstein and Silke Eckstein. *XML und Datenmodellierung. XML-Schema und RDF zur Modellierung von Daten und Metadaten einsetzen.* dpunkt.verlag, 2004.

[13] A General Framework for Evolution and Reactivity in the Semantic Web. Draft, for further information see http://www.dbis.informatik.uni-goettingen.de/rewerse.

[14] Charles F. Goldfarb. *The SGML Handbook.* Oxford University Press, 1991.

[15] Asunción Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering. With examples from the areas of Knowledge Management, e-Commerce and the Semantic Web.* Advanced Information and Knowledge Processing. Springer, 2004.

[16] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[17] David Hirtle, Harold Boley, Benjamin Grosof, Michael Kifer, Michael Sintek, Said Tabet, and Gerd Wagner. Schema Specification of RuleML 0.91. http://www.ruleml.org/0.91, 2006.

[18] HP Labs Semantic Web Research. http://www.hpl.hp.com/semweb.

[19] HyperText Markup Language (HTML) Home Page. http://www.w3.org/MarkUp, 1989.

[20] Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net.

[21] Michael Kay. *XSLT Programmer's Reference.* Wrox Press Ltd., 2000.

[22] Alfons Kemper and André Eickler. *Datenbanksysteme. Eine Einführung.* Oldenbourg, 5th edition, 2004.

[23] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active Rules in the Semantic Web: Dealing with Language Heterogeneity. In *Rule Markup Languages (RuleML)*, number 3791, pages 30–44. Springer, 2005.

[24] Wolfgang May, José Júlio Alferes, and Ricardo Amador. An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web. In *Ontologies, Databases and Semantics (ODBASE)*, number 3761, pages 1553–1570. Springer, 2005.

[25] N-Triples. `http://www.w3.org/2001/sw/RDFCore/ntriples`, 2001.

[26] OWL Web Ontology Language. `http://www.w3.org/TR/owl-features`, 2002.

[27] Resource Description Framework (RDF). `http://www.w3.org/RDF`, 2000.

[28] RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema`, 2000.

[29] RDF/XML Syntax Specification (Revised). `http://www.w3.org/TR/rdf-syntax-grammar`, 2004.

[30] Daniel Schubert. Development of a Prototypical Event-Condition-Action Engine for the Semantic Web. Bachelor Thesis, Univ. Göttingen, 2005.

[31] Semantic Web. `http://www.w3.org/2001/sw`.

[32] SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query`, 2004.

[33] SPARQL Query Results XML Format. `http://www.w3.org/TR/rdf-sparql-XMLres`, 2004.

[34] Heiner Stuckenschmidt, Frank van Harmelen, Wolf Siberski, and Steffen Staab. Peer-to-Peer and Semantic Web. In Steffen Staab and Heiner Stuckenschmidt, editors, *Semantic Web and Peer-to-Peer. Decentralized Management and Exchange of Knowledge and Information*, volume 96 of *Frontiers in Artificial Intelligence and Applications*. Springer, 2006.

[35] Sun Microsystems, Inc. The Source for Java Developers. `http://java.sun.com/`.

[36] Jenny Tennison. *XSLT and XPath On The Edge, Unlimited Edition*. M&T Books, 2001.

[37] Jeffrey D. Ullmann. *Principles of Database and Knowledge-Base Systems*, volume I: Classical Database Systems. Computer Science Press, 8th edition, 1995.

[38] Web Naming and Addressing Overview (URIs, URLs, ...). `http://www.w3.org/Addressing`, 1993.

[39] W3C – World Wide Web Consortium. `www.w3c.org`.

[40] W3C RDF Validation Service. `http://www.w3.org/RDF/Validator`.

[41] R. Allen Wyke and Andrew Watt. *XML Schema Essentials*. John Wiley & Sons, Inc., 2002.

[42] Extensible Markup Language (XML). http://www.w3.org/XML, 1998.

[43] XML Schema. http://www.w3.org/XML/Schema, 2001.

[44] XML Path Language (XPath). http://www.w3.org/TR/xpath, 1999.

[45] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, 2001.

[46] The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL, 1998.

[47] XSL Transformations (XSLT). http://www.w3.org/TR/xslt, November 1999.