Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Übersetzung von Aktionen auf semantischer Ebene in einem RDF Web Service

Thomas Westphal

am Lehrstuhl für

Datenbanken & Informationssysteme

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

30. März 2007

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 30. März 2007

Bachelor Thesis

Translation of semantic level actions in an RDF Web Service

Thomas Westphal

30. March 2007

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

Abstract

The basis of the Semantic Web is provided by autonomously evolving information systems. Today's internet mostly consists of web pages that can't be processed easily in a computer-understandable way. These information systems can be queried as known from Web Services and additionally use domain ontologies that enable automatic information integration. MARS Modular Active Rules in the Semantic Web is based on this infrastructure and adds the capability to specify and execute active rules using the Event-Condition-Action (ECA) paradigm. Reactive behavior is created by an event-driven communication using semantically high level events that are raised within a network of independent services. High level events are derived from data model changes using ECE rules. If an optional condition is satisfied, semantically equally high level actions are distributed and executed. In this thesis, an ACA mapper is developed that enables the definition of rules that map semantic level actions to a sequence of data model update commands. Such a mapping is needed to execute domain level actions in an RDF Web Service.

Contents

List of Figures

Chapter 1

Introduction

The infrastructure provided by the Semantic Web differs severely from today's World Wide Web where the content is mostly given by documents that are intended for human readers to use. The information technologies developed for the Semantic Web enable the creation of autonomously evolving heterogeneous information systems that hold computer-understandable data.

These systems are not only static isolated data sources but may be extended to show reactive behavior driven by the communication of semantically high level events. Usually, these events are raised within an information system according to particular update operations. Active rules serve to trigger actions if certain events occur and when optional additional conditions are satisfied.

Based on the MARS Framework that is presented in [1], this thesis deals with the development of an ACA mapper for an RDF Web Service. Such a module is necessary for the execution of semantic level actions at the Web Service. It enables the definition of ACA rules that describe how a high level domain action can be executed locally using data model operations.

The next chapter gives an overview of the MARS Framework including the description of important aspects of the Semantic Web in general. Following this, the RDF Web Service is introduced that is the foundation of the ACA mapper. Additionally, a RDF update language is explained that was developed with the Web Service and has been extended by an XML markup for this thesis. Chapter 4 deals with the description of the ACA mapper and explains how high level actions are processed and additionally describes the syntax of ACA rules. Finally, Chapter 5 describes the implementation of the prototype by giving an outline of the employed technologies and by describing the core classes followed by the installation instructions of the extended RDF Web Service.

Chapter 2

MARS: Modular Active Rules in the Semantic Web

2.1 Semantic Web

The goal of the Semantic Web is to provide a new infrastructure in the internet where different services can communicate in a computer-understandable manner. Today's internet usually presents itself as a source of HTML documents that represent a markup of information for human readers to process. The actual information contained in these documents is usually stored within databases. Such an infrastructure is not suited for automated machine-understandable integration of different data sources. That is because direct integration of different databases requires a lot of effort because they usually do not share a common schema. They have to be mapped manually by a wrapper that joins the different schemas and makes them accessible through a common interface. For an HTML page to be interpreted in a machine based manner also special wrappers are needed to provide a way to enrich the given data with semantics.

The Semantic Web is an approach to build a semantical level upon today's internet. Basically, that is achieved by using a data model that forces users to define common ontologies. These are used to describe the concepts and relations of a certain domain on a semantical level. Such a procedure is not new because other data models may also be described on a meta level using some kind of schema description. The difference is the importance to do so. In the Semantic Web, however, defining a common schema is a core task and not just an auxiliary description.

To achieve this the Semantic Web uses W3C [20] recommendations that will be described in short in the following.

RDF and RDF Schema The Resource Description Framework (RDF) [14] is the foundation of the Semantic Web. It provides the possibility to describe concepts in a computer-processable manner. That is done by defining subject-predicate-object triples where two resources can be associated. Resources and predicates are identified by URIs[1]. An RDF specification can be seen as graph with labeled nodes and labeled edges. Furthermore, the RDF Schema (RDFS) [16] provides meta data specification for RDF. There are certain predefined concepts to describe classes and properties. Also, the hierarchy of classes and properties may be expressed.

[1]Unified Resource Identifier

RDF graphs can be defined using several formats. The most common ones are the N3 syntax [11] and RDF/XML [15]. The former is a direct approach that simply uses triple definitions to describe RDF content. N3 is easy to read for human readers and therefore it is the preferred syntax in this thesis. RDF/XML provides an XML language for the description of RDF data. Because XML is a common exchange format in many contexts and especially within the internet, RDF/XML is most commonly used.

OWL The Web Ontology Language (OWL) [12] extends the meta concepts given by RDF Schema and provides the possibility to describe resources on an even higher semantic level. Using OWL, it is possible to specify concepts in more detail. For example, a property of an concept can be defined as inverse to another property or as transitive. Therefore, even the transitive closure can be expressed. To do so, OWL needs some kind of reasoning which is provided by Description Logic. The latter is a decidable subset of First Order Logic.

Ontology An ontology in computer science is a set of notions, concepts, concept hierarchy and relations known about a given domain. It describes the whole vocabulary and meta data of the specific domain.

An ontology in the Semantic Web differs from those of classical data models in its complexity. By defining an Entity-Relationship model it is possible to describe a domain ontology in the relational model. But such an ontology consists only of static notions expressed by relations or entity types with their attributes and relationships. With UML it is possible to define ontologies that do not only describe static notions. Dynamic issues of a domain may also be described by declaring actions. A complete domain ontology in the Semantic Web does not only have to describe the static issues of that domain like predicates or literals but also all dynamic issues including actions *and* events. For example a banking domain will consist of concepts like bank, money or credit-transfer or define actions like withdraw-money and events like overdraft.

2.2 MARS Framework

The MARS-Framework (Modular Active Rules in the Semantic Web) [1] provides an infrastructure for implementing reactive behavior within the Semantic Web. Such behavior is achieved by the specification and execution of active rules. These rules follow the Event-Condition-Action (ECA) paradigm and therefore consist of three components. An event that determines when the rule shall be triggered, a condition (often some kind of query collecting data) and an action which is executed when the condition is satisfied (possibly using gathered information). The infrastructure that is suggested by this framework intends the distribution of events throughout a network of participating service nodes, followed by the evaluation of registered rules and final distribution of the resolving actions as consequences of these events. The *ECA Rules* are intended to be executed on a high global level. Rules can be defined using arbitrary languages for the event, condition and action component because there is an abstraction level provided by Language Services that implement the different language processors. These Language Services are accessible via a Languages and Services Registry (LSR) that is addressed by the rule processor. Therefore any event or action algebra may be used to define a rule for which there is an according Language Service that implements this language.

The bottom of the MARS infrastructure is provided by Domain Services that implement the behavior of certain domain ontologies. Therefore they have to emit domain events and have to be able to process domain actions.

The concepts of the MARS Framework are described in an ontology and as such it is a resource within the Semantic Web itself. The ontology contains MARS meta concepts for describing and implementing domain services. Such concepts include the languages needed for defining rules or queries and methods for defining actions and events that build the main structure for the reactive behavior.

ECA Rules These rules are the core of the MARS Framework. They relate events that may occur within the network with actions that are to be executed upon these events. On the detection of an event an optional condition part is evaluated which may then lead to the execution of the action part.

Example 1 (ECA Rule) *The following XML document represents an* ECA Rule *in the* eca-ml *markup. It has an* eca:Event *componment using* snoopy: *as event algebra, a condition component provided by the* eca:Query *element with an opaque SPARQL query and finally an action part using* ccs: *as language.*

This particular rule assures that when any flight is canceled on which a customer of the agency was booked the agency is informed by email. Additionally to the email a hotel room for the passenger should automatically be booked at the concerning airport.

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml#">
  <eca:Event xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy#">
    <snoopy:Sequence>
      <travel:delayed-flight flight="{$flight}" date="{$date}"/>
      <travel:canceled-flight flight="{$flight}" date="{$date}"/>
    </snoopy:Sequence>
  </eca:Event>
  <eca:Query>
    <eca:opaque language="sparql" domain="travel">
      ... some query according to the task selecting the
          $name of the passenger and a corresponding $hotel-uri ...
    </eca:opaque>
  </eca:Query>
  <eca:Action xmlns:ccs="http://www.semwebtech.org/eca/2006/ccs#">
    <ccs:Sequence>
      <ccs:Action>
        <travel:reserve-room hotel="$hotel-uri" name="$name"/>
      </ccs:Action>
      <ccs:Action>
        <smtp:send-mail to= "myAgency@some-provider.com">
          "flight $flight caceled ... customer $name was bokked at $hotel-uri ...
        </smtp:send-mail>
      </ccs:Action>
    </ccs:Sequence>
  </eca:Action>
</eca:Rule>
```

For more details about *ECA Rules* and their markup see [1].

ECA Engine, Event Detection Service, Action Engine Some service within the framework has to be responsible for the evaluation of *ECA Rules*. Such a service is provided by the *ECA Engine*. A prototype has been implemented in the bachelor thesis by Daniel Schubert [17]. The service supports the registration of *ECA Rules*. These are executed when a certain atomic or composite event occurs. The *ECA Engine* itself is not responsible for the detection of events or for the execution of actions but delegates this tasks to specialized services. These are the *Event Detection Service* and the *Action Engine*, respectively. There are several of these services, each for a particular event or action language.

The *Event Detection Service* is able to detect not only atomic but also composite events (for example a sequence of events as in Example 1). It uses *Atomic Event Matchers* to gather atomic event occurrences within a certain application domain.

Event and Action Language Services There are several Language Services for event action and query handling. These services are responsible for different languages (event, action algebras etc.) but each class of these services provides a common set of tasks. That means that for example every *Composite Event Detection Engine* supports a task for the registration of an event pattern or that each *Action Engine* supports the execution of an action. Just like with the heterogeneity of possible languages for an *ECA Rule*, every provider of a Language Service has the liberty to implement these tasks freely. There is a registry provided by the LSR where all Language Services are listed including an description of how they have to be addressed. Therefore, this registry can be asked which particular service processes for example a certain event algebra.

Domain Service Domain Web Services implement one or more domain ontologies. That means they use concepts and behavior that is defined in these domain descriptions. For example, a travel agency may use a travel ontology containing concepts like flight-connection, hotel, or book-flight along with additional notions defined within a buisness ontology etc. A Domain Service has to implement an appropriate communication interface for being compatible with the MARS Framework. That includes receiving tasks as actions from a domain language and raising events respectively. Additionally, a domain service has to provide some information about the kind of services it supports.

Domain Broker Obviously, events and actions have to be distributed between different services within the framework. Domain Brokering Services provide this task. For each application domain there is at least one domain broker where Domain Services may be registered. The domain brokering may be divided in three separate services. An *Event Broker* receives and forwards events raised within registered Domain Services. A *Query Broker* provides an interface for SPARQL requests within the domain. Additionally, there is an *Action Broker* that forwards actions within a downward communication to Domain Services that are marked to support the certain action command. That means that an *Action Engine* that is used to execute a certain complex action by the *ECA Engine* forwards atomic domain ontology actions to an Action Broker of that domain.

At this point the broker iterates over the registered Domain Services and either broadcasts the action or chooses relevant services by some kind of reasoning using the underlying domain ontology.

Example 2 *The communication between the framework services is quite complex. There are multiple abstraction levels, so it is advisable to introduce a small step by step example adapted from [1]. As a case study, Figure 2.2.1 below shows a travel agency (*Client C*) that registers an ECA rule at an ECA Engine (point 1.1). The particular rule was already described in Example 1.*

The ECA Engine *recognizes the request to register a new rule and extracts the event component. Corresponding to the used event algebra (here* snoop:*) an Event Detection Service is contacted that processes the composite event definition (point 1.2). The* ECA Engine *is registered to be informed about any occurrences of the complex event.* Atomic Event Matchers *are involved internally (1.3). At this point the event is splitted into its atomic components from the* travel: *language. Therefore, a* Domain Broker *or more explicitly an* Event Broker *of the* travel: *application domain is contacted and asked to register the* Atomic Event Matcher *to be informed of occurrences of relevant events raised within the domain (1.4). If so (2.1, 2.2), these events or their variable bindings respectively are communicated upwards (3). If the whole event as defined by the event algebra is detected, the ECA Engine is informed (4). Now the whole action part is sent to an* Action Engine *that supports the used action algebra (here* ccs:*) (5.1). Atomic events are sent to appropriate brokers and services (5.2) where they are forwarded or executed.*

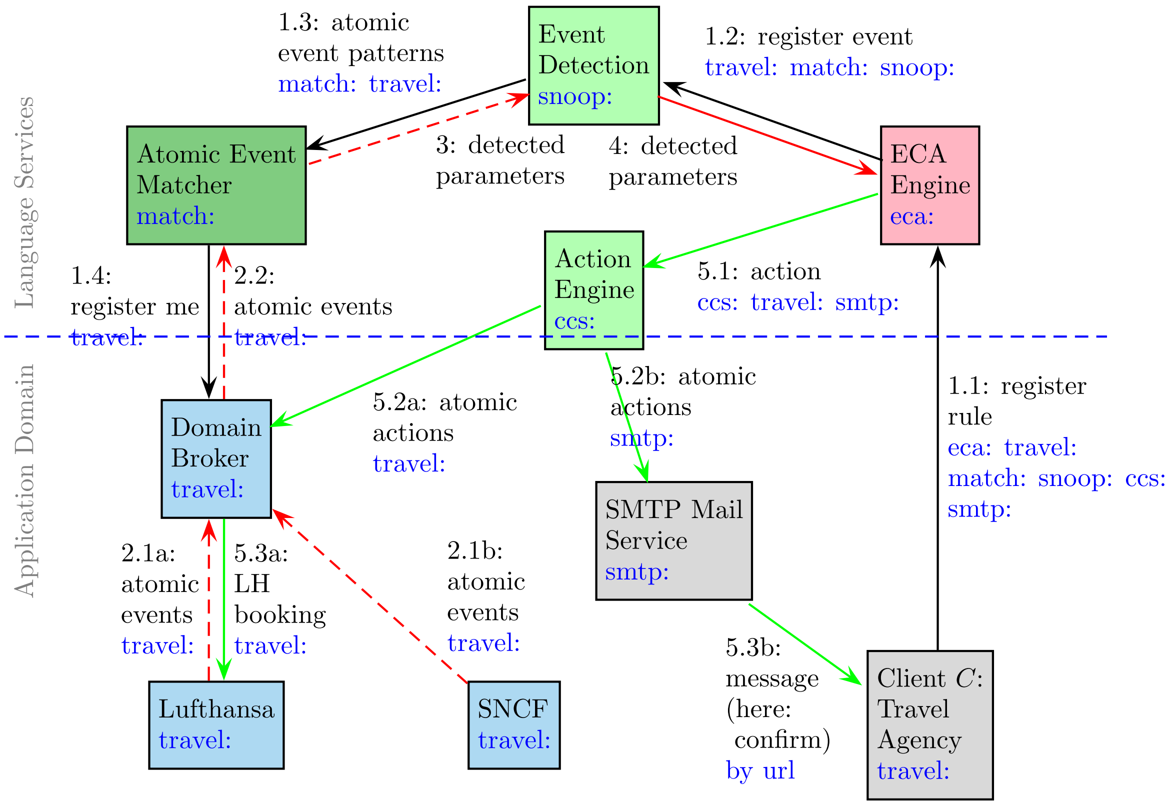

Language Services
Application Domain
Event Detection snoop:
1.3: atomic event patterns match: travel:
1.2: register event travel: match: snoop:
Atomic Event Matcher match:
3: detected parameters
4: detected parameters
ECA Engine eca:
Action Engine ccs:
5.1: action ccs: travel: smtp:
1.4: register me travel:
2.2: atomic events travel:
Domain Broker travel:
5.2a: atomic actions travel:
5.2b: atomic actions smtp:
1.1: register rule eca: travel: match: snoop: ccs: smtp:
SMTP Mail Service smtp:
2.1a: atomic events travel:
5.3a: LH booking travel:
2.1b: atomic events travel:
5.3b: message (here: confirm) by url
Lufthansa travel:
SNCF travel:
Client C: Travel Agency travel:

Figure 2.2.1: Communication: Event Processing, Action Forwarding (taken from [1])

As seen above, the communication interfaces within the MARS framework are dynamic. Additionally to the Language and Service Registry, there has to be a corresponding registry for Domain Brokers and Domain Services. When an Event Detection Service or an Atomic Event Matcher wants to register a certain event at an Event Broker there has to be a way to identify the corresponding Domain Brokers responsible for that domain. For actions distribution there is the same problem. Even if a Domain Broker for a domain is known, it is insufficient to broadcast an action request to all Domain Services of that domain for not all will support that specific action. Therefore there has to be a Domain Service Registry (DSR) that provides more detailed information about the Domain Services.

2.3 Application Domain Nodes

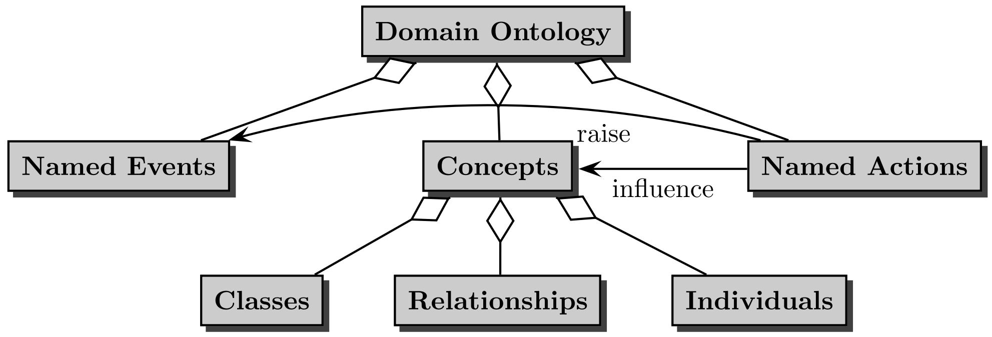


Figure 2.3.1: Structure and Interference in Ontologies (taken from [1])

Application domain nodes represent Domain Service implementations in an application domain like airlines, car rentals, universities or other. These nodes are similar to ordinary Web Services and hold local data in some kind of database or knowledge base that may be queried and manipulated through external interfaces. The difference to ordinary Web Services is that they are registered at a Domain Broker of the domain they support and of course they are framework-aware.

An application in the Semantic Web will usually use several domain ontologies though it will "live" in one in particular. A travel agency for example will use a *travel* application domain but probably it will also need certain features from a *banking* ontology. Generally ontologies of several domains interfere. But there are also interferences between the components of every single ontology as shown in Figure 2.3.1. The static issues or concepts may be divided into classes, relationships and individuals. All of these may be influenced by actions that may further raise events.
The framework defines special concepts for actions and events of a domain ontology. These are:

```
[ mars:Event rdf:type owl:Class ]
[ mars:Action rdf:type owl:Class ]
```

For being recognized by framework-aware nodes it is necessary that actions and events that are defined in an ontology can be identified to belong to these classes by reasoning. It is recommended that a domain ontology describes a hierarchy of actions and events that inherit in some way from mars:Event and mars:Action.

Example 3 (Application Domain Ontology) *Usually, application domain ontologies are defined by OWL documents. Consider an ontology regarding a university with its staff, students lectures etc. The following example uses the N3 syntax [11] . It contains class definitions as well as predicates and dynamic issues represented by actions and events. Note that the ontology describes* uni:Action *and* uni:Event *concepts which are subClassOf* mars:Action *and* mars:Event *respectively. The inheritance from the* mars *concepts is important to achieve the intended behavior and framework awareness. Alternatively defining* uni:hired-professor *directly as instance of* mars:Event *would be possible. But that would suppress the fact that it is not only an event but its also belonging to the university ontology.*

The university application domain ontology will be the foundation for other examples in this thesis. The URI of this ontology is http://localhost/test.owl#*. It has a Domain Broker that can be reached at* http://localhost/service/uni-domain *and there is one application domain service that implements the domain reachable at the URL* http://localhost/service/uni-service.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix mars: <http://www.semwebtech.org/2006/mars#> .
@prefix uni: <http://localhost/test.owl#> .
@prefix my: <http://localhost/service/> .

uni: a mars:Domain ;

uni:Action a mars:Class ;
      rdfs:subClassOf mars:Action ;
      mars:belongs-to-domain uni: .
uni:Event a mars:Class ;
      rdfs:subClassOf mars:Event ;
      mars:belongs-to-domain uni: .

uni:register-student a mars:Class ;
      rdfs:subClassOf uni:Action ;
      mars:belongs-to-domain uni: .
uni:professor-hired a mars:Class ;
      rdfs:subClassOf uni:Event ;
      mars:belongs-to-domain uni: .
           ...

uni:Lecture a mars:Class ;
       mars:belongs-to-domain uni: .
uni:Professor a mars:Class ;
       mars:belongs-to-domain uni: .

uni:in-charge-of-lecture a mars:Property ;
       mars:belongs-to-domain uni: ;
       rdfs:domain uni:Professor ;
```

```
        rdfs:range uni:Lecture ;
        owl:inverseOf uni:lecture-has-professor .
               ...
```

Additionally to application domain ontologies, there are also application-independent ones. The latter describe an application or provide concepts and behavior that will be needed in arbitrary applications. This includes for example services like messaging, transactions, calendars, generic data manipulation etc.

Within the MARS Framework a Domain Service has to provide two special tasks. These are described in the Service Ontology (see [1]). The relevant part is (given in RDF/XML):

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl"
     xmlns=http://www.semwebtech.org/2006/mars#"
     xml:base=http://www.semwebtech.org/2006/mars">
     <rdf:Description rdf:about=#DomainService">
          <meta-provides-task rdf:resource="/domain-node#receive-query"/>
          <meta-provides-task rdf:resource="/domain-node#receive-action"/>
          <meta-provides-task rdf:resource="/domain-node#give-service-description"/><!– opt –>
     </rdf:Description>
</rdf:RDF>
```

Therefore a Domain Service has to provide an interface to receive queries and application level actions. Note that these resource URIs only represent the names of the tasks that have to be supported by a node. How they are implemented and addressed exactly is to be specified for each node separately. Furthermore, there has to be the possibility to request a detailed Service Description. Such a description is itself a small ontology given in RDF/XML. It describes a service concerning the application domains it uses and the high level actions it supports. The Service Description of a domain node supporting the university ontology described earlier could look like the following RDF/XML example.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:mars="http://www.semwebtech.org/2006/mars#"
     xmlns:uni="http://localhost/test.owl#" >
     <mars:DomainService rdf:about=http://localhost/test/uni-service">
          <mars:uses-domain rdf:resource=http://localhost/test.owl#"/>
          <mars:supports rdf:resource="http://localhost/test.owl#register-student"/>
          ...
     </mars:DomainService>
</rdf:RDF>
```

Additionally to the Service Description, a Domain Service also has to describe its tasks and the way they can be addressed respectively. When a new Domain Service is initialized and is inserted into the network provided by the framework, it has to register at a Domain Broker for each domain it supports. Such a registration includes the communication of the Service Description of the new domain node. Possibly, the Domain Broker will request the Service Description to identify the actions the node supports. The information gathered about an application domain node is forwarded and stored within the DSR.

Application domain nodes are the leaves in the Semantic Web. They show reactive behavior by supporting certain actions and events of one or more application ontologies. Actions are either received by a Domain Broker or any other node. Usually actions are conceps ?X in an ontology such that { ?X rdfs:subClassOf mars:Action } holds. These actions are communicated in an XML markup possibly with variable bindings and represent a small RDF/OWL graph. Their execution normally leads to a change in the local data state like booking a flight or registering a student for a particular course.

Example 4 (Application Domain Actions) *Assume an ECA rule for an airport or travel ontology that reacts upon bad weather situations. The actions that are triggered by such a rule may lead to the delay or canceling of flights. Such a rule may lead to the broadcasting of a series of application domain actions like the following one:*

```
<travel:cancel-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
    flight="LH1234" date="20060523">
    <travel:reason>bad weather</travel:reason>
</travel:cancel-flight>
```

For this example it is assumed that the RDF statement

[travel:cancel-flight rdfs:subClassOf mars:Action]

holds in the concerned ontology.

These nodes also support application ontology events by being able to emit events upon changes of their local data (possibly caused by the execution of actions). Such a *push communication* for news is a great advantage of the MARS Framework. Nodes that support this reactive behavior do not have to be monitored to identify changes. For example an event may be raised when a flight is half booked or a professor is newly employed at a certain university. Events are just like actions communicated in a XML format. Analogously to actions events are concepts ?X such that { ?X rdfs:subClassOf mars:Event } holds. They are forwarded to the corresponding domain broker and may be responsible for the triggering of ECA rules which again leads to the raising of actions.

Example 5 (Application Domain Events) *Events are data fragments that are available in XML markup. The events*

```
<travel:canceled-flight flight="LH123" date="20060523"/>
    <travel:reason>bad weather</travel:reason>
</travel:canceled-flight>

<travel:delayed-flight flight="LH1234" minutes="30"/>
```

are possible events from the traveling domain and mean that the flight "LH1234" is canceled for the given reason or that its departure has been delayed for 30 minutes.

For being able to execute application domain actions or emitting events it is necessary that a node supports the registration of rules that define how a certain high level action can be processed locally or when a certain application domain event shall be raised. In detail there are three kinds of rules that have to be considered:

- low level **local ECA rules (triggers)** that ensure local integrity maintenance in a knowledge base,
- **ECE rules** that raise application level events upon changes in the local knowledge base triggered by RDF modification events,
- **ACA mappings** which map higher level application domain actions to actions on a lower semantic level. The processing of semantically high actions from the ontology level involve the execution of several data model level actions within the domain node. These are the only kind of actions the node can process directly.

So when actions are received by a node, it checks if there are registered *ACA mappings* that map this probably high level action to a sequence of instructions it can process locally. Such instructions may include updating the local data, directly raising derived events or other. Sometimes it is necessary for certain updates in the local knowledge base that triggers care for the model integrity (see Section 3.4).

Registered *ECE rules* may be triggered when the local model of a node has changed. These rules react upon RDF/OWL data model level events such as insertion, deletion or modification of a statement. That means that while mapping received actions to a semantically lower level, an application domain node ascends low level data model events to the application ontology level. The latter will be forwarded by the corresponding event broker and will probably lead to the detection of other events and to the execution of actions through the ECA Engines.

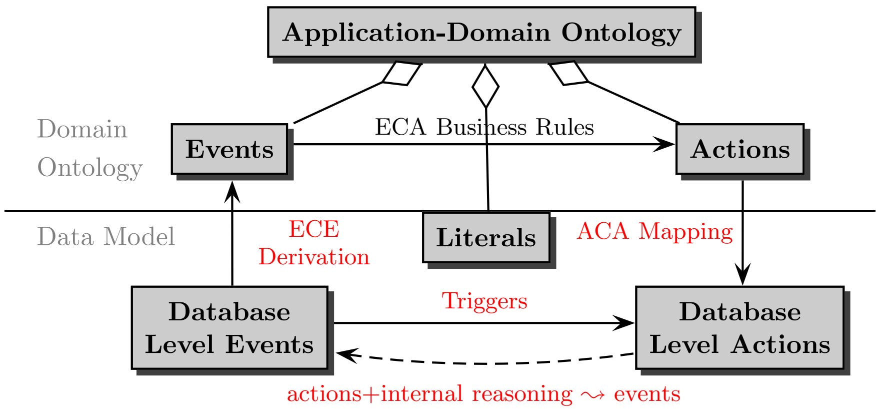


Figure 2.3.2: Interference of Events, Actions, and Literals (taken from [1])

Summary Application domain nodes support the following behavior:

- receiving queries; answering them,
- receiving action requests, executing them,
- raising events and
- administrative stuff: registering *ACA mappings*, *ECE rules* etc.

2.4 ACA Mappings at the Application Domain Node

So one of the main tasks of an application domain node is to execute high level actions defined in the corresponding ontology. From the view of the ontology, the actions sent to a domain service are atomic. Nevertheless for an application that is supposed to update its local data these actions can't be executed directly. They have to be mapped to the local data model level. As mentioned above, a node uses ACA mappings to provide this task. How that happens is explained in Section 4.3 and shall be briefly introduced here.

Assume the request to execute an action is received by a domain node. This action shall represent a simple modification of the local knowledge base. More specifically it represents the registration of students for a certain lecture. The message could look like this:

```
<uni:register-students xmlns:uni="http://localhost/test.owl#">
    <uni:lecture>http://localhost/myuni/lectures/WS20062007/semweb</uni:lecture>
    <uni:student>http://localhost/myuni/students/2019987</uni:student>
    <uni:student>http://localhost/myuni/students/2029382</uni:student>
    <uni:student>http://localhost/myuni/students/1023457</uni:student>
    <uni:student>http://localhost/myuni/students/2014473</uni:student>
</uni:register-students>
```

To deal with such an action, an application node needs to know some kind of INSTEAD-trigger that translates the semantically high action to simple data model updates. Additionally, it is possible that some actions should only be executed if a certain condition is fulfilled. In the given example it could be advised to check if the lecture is known to the local knowledge base. Furthermore the mechanism providing the mapping functionality has to be implemented by a procedural language because it has to iterate over the list of students. Additionally, it could be necessary to generate a new resource URI and insert several statements about that resource into the model.

Seeing that the input language is XML and the task of mapping such an input to a sequence of data model updates can be seen as a transformation, the idea to use XQuery [22] or XSLT [23] suggests itself. These are well known W3C recommendations and integrate nicely in the general web service scenario.

The above action <uni:register-students> could easily be mapped by an XQuery instruction:

```
declare namespace uni = "http://localhost/test.owl#";
let $lecture_uri := //uni:register-students/uni:lecture/text()
return
  <rdfu:condition ask="&lt;{$lecture_uri}&gt; rdf:type uni:Lecture">
```

```
{
   for $student in //uni:register-students[./uni:lecture]/uni:student
   return
     <rdfu:insert>
       <rdf:subject rdf:resource="{$lecture_uri}" />
       <rdf:predicate rdf:resource="http://localhost/test.owl#has-student" />
       <rdf:object rdf:resource="{$student/text()}" />
     </rdfu:insert>
}
</rdfu:condition>
```

Syntax and semantic of ACA mappings like these will be discussed in Section 4.3. The result of an ACA mapping is expected to be an executable sequence of actions on a lower semantical level that are natively implemented within the domain node. The XML element rdfu:condition is such an action from the extended RDF update language (Section 3.2). It expects an ask parameter containing an ASK SPARQL query. The contents of the condition element will be executed if the condition evaluates to true using the local knowledge base as underlying model. Naturally rdfu:insert is also a native node action. It just inserts a given RDF statement into the local model. Note that several XQuery modules will be automatically imported on execution of a XQuery ACA mapping as will be described in Section 4.3.1

Chapter 3

RDF Web Service with Update and Trigger Functionality

This thesis deals with the adaption of an RDF Web Service to an application domain node in the MARS Framework [1]. The Web Service was created by Elke von Lienen as a Diploma thesis [19]. It uses the Jena API [6] as foundation for an RDF/OWL knowledge base and provides an infrastructure for active rules with triggers. Jena is an open framework for the Semantic Web written in Java. It provides tools for dealing with RDF/OWL graphs like storing the model data in a relational database. This Web Service uses PostgreSQL for the storage. Furthermore, it uses the separate DL reasoner Pellet [13] for reasoning via the DIG interface [3]. A detailed description of the Web Service can be found in [10].

The triggers that are implemented in the Web Service react on insert, update or delete transactions within the knowledge base. They may be defined to either fire before or after execution of the requested update. Therefore they may be used for maintaining the consistency of the knowledge base, committing complex update transactions or for the creation of events. These events are communicated to a registered Domain Broker [8]. The triggers are described in more detail in Section 3.4.

On top of that, the Web Service offers the possibility to query the local model using SPARQL [18]. So it provides most of the conditions necessary to be used as application domain node within the MARS Framework. But so far it has no interface for receiving complex actions from the Domain Broker. The only update possibility is given by a low level update interface described in Section 3.1. The actions which are sent by the Domain Broker are defined in the specific domain ontology and are therefore on a higher semantic level than simple RDF updates.

The following sections describe the service in more detail. Note that minor changes were made to syntax and functionality compared with the prototype first implemented. Everything described in this chapter presents the current state of development.

3.1 Interface for Manipulation of the OWL Model

The Web Service provides the possibility to manipulate the OWL model. For this task an *RDF update* language has been developed. *RDF update* provides the structure for update requests which can be executed by the Web Service. Each single of these update messages begins with

a command string mostly followed by an RDF triple specifying the statement to be updated. The triples should be given with absolute URIs but namespace prefixes are possible for standard namespaces as rdf, owl and rdfs or namespaces explicitly defined within the model.
There are six different update operations for the manipulation of RDF statements:

insert (rdf:subject URI, rdf:predicate URI, rdf:object URI or literal) This update command inserts the given [rdf:subject rdf:predicate rdf:object] triple into the model.

assert (rdf:subject URI, rdf:predicate URI, rdf:object URI or literal) Assert ensures that after execution of the command the model holds the given RDF statement. If the latter already holds because it is explicitly given as a fact or can be inferred by the reasoner, nothing happens.

delete (rdf:subject URI, rdf:predicate URI, rdf:object URI or literal) Deletes the given RDF statement if it is explicitly contained as a fact in the model. That means if the given statement exists as materialized fact within the model it is removed. Note that delete does not ensure that the statement will not hold after the operation. It may still be able to derive the given statement by reasoning.

delete resource (resource URI) The argument of this *RDF update* command may be any resource URI. It leads to the deletion of any statement containing this resource either in subject, predicate or object position. Just like delete the delete-resource operation does not influence statements derived by reasoning.

retract (rdf:subject URI, rdf:predicate URI, rdf:object URI) Like delete, this *RDF update* command removes a statement from the model. The difference is that using this update operation it is guaranteed that the specified statement cannot be inferred by other statements in the model. That means after this operation was executed successfully it is ensured that the given statement does not hold within the model.

update (rdf:subject URI, rdf:predicate URI, rdf:object URI or literal)
set subject|predicate|object = new_value With this operation it is possible to update the specified RDF statement by assigning a new *rdf:subject* URI, *rdf:predicate* URI or *rdf:object* value within the statement.

rename [property of class] (old rdf:predicate URI, [rdfs:Class URI,] new rdf:predicate URI)
By sending a rename request to the web service it is possible to rename a *rdf:predicate* URI within the model. There are two possibilities to use this update option. Every occurrence of the specified *rdf:predicate* URI is renamed. If a class is given, only those URIs are renamed which are a property of an instance of the specified *rdfs:Class*.

Example 6 *By sending an HTTP message with the content*

```
insert ( http://some.class.uri, rdf:type, owl:Class)
```

to the web service, the statement

```
[ <http://some.class.uri>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.w3.org/2002/07/owl#Class> ]
```

will be inserted into the model. If the model contains the statement:

```
[ <http://some.instance.uri>
          <http://some.namespace#property> "current value" ]
```

and the object value should be changed to "new object value" *the* RDF update *message would look like this:*

update (http://some.instance.uri, http://some.namespace.uri#property, “current value“)
set object = “new object value“

3.2 RDF Update XML Markup

The RDF update language defines data model update instructions that are important not only for a generic RDF Web Service but also for an application domain service within the MARS Framework. These are the low level actions that are used by ACA rules at Domain Services to map higher domain level actions to the data model level. To be used by ACA mappings, it is helpful if there is an XML markup for these operations. The markup described in this section follows a straightforward attempt. It should be noted that this XML language uses a namespace with the base http://www.semwebtech.org/lang/2006/ as it is intended for languages within the MARS Framework. Additionally to the operations introduced in the previous sections, there is a new one called rdfu:condition that can be used to evaluate a given SPARQL ASK query. If it holds, update commands defined within the rdfu:condition element are executed. Otherwise nothing happens.
The update commands rdfu:insert, rdfu:delete, rdfu:assert and rdfu:retract are very similar in their plain text syntax and therefore also in their XML markup. As an example, an insert command is given below. The only difference to the other commands is the name of the root element. Of course it has to be the URI of the update command that should be executed.

<**rdfu:insert** xmlns:rdfu=“http://www.semwebtech.org/lang/2006/rdfupdate#”
  xmlns:rdf=“http://www.w3.org/1999/02/22-rdf-syntax-ns#” >
  <**rdf:subject** rdf:resource=“subject URI” />
  <**rdf:predicate** rdf:resource=“predicate URI” />
  [<**rdf:object** rdf:resource=“object URI”/> | <**rdf:object**>object literal<**/rdf:object**>]
<**/rdfu:insert**>

The URIs that specify the RDF resources have to be absolute. RDF literals are also allowed as rdf:object. In the prototype implementation string literals can be specified by a quoted char sequence.

Example 7 *Assume the following statement with a typed literal should get inserted into a knowledge base using the RDF update language.*

```
[ http://my/persons#peter http://my/meta#name
             "Peter"^^<http://www.w3.org/2001/XMLSchema#string> ]
```

The corresponding RDF update command would be:

```
<rdfu:insert xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <rdf:subject rdf:resource="http://my/persons#peter" />
    <rdf:predicate rdf:resource="http://my/meta#name" />
    <rdf:object>"Peter"</rdf:object>
</rdfu:insert>
```

Note that the URI and literals used have to be PCDATA contents. That means that characters like '<' and '>' have to be translated to the corresponding XML entities.

rdfu:update is marked up as follows:

```
<rdfu:update xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <rdf:subject rdf:resource="subject URI" />
    <rdf:predicate rdf:resource="predicate URI" />
    [<rdf:object rdf:resource="object URI"/> | <rdf:object>object literal</rdf:object>]
    <rdfu:set>
      [<[rdf:subject|rdf:predicate| rdf:object] rdf:resource="new value"/>] |
       <rdf:object>object literal</rdf:object>]
    </rdfu:set>
</rdfu:update>
```

Thus this command basically extends the commands mentioned above by a rdfu:set element that contains one child element that is either rdf:subject, rdf:predicate or rdf:object. The new value will be assigned to the corresponding component of the statement.

The element rdfu:rename is the markup for both rename and rename property of class that are described in Section 3.1. The command rename property of class may be expressed by including a rdfs:Class element with the class URI as text content as child of rdfu:rename.

```
<rdfu:rename xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    rdf:resource="resource URI" />
    [<rdfs:Class rdf:about="optional class URI"/>]
    <rdfu:new-value rdf:about="object URI or literal"/>
</rdfu:rename>
```

rdfu:delete-resource is the last remaining original RDF update command. It is simply marked-up as follows:

```
<rdfu:delete-resource xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    rdf:resource="resource URI" />
```

A new RDF update command has to be introduced to enable a conditional execution of application domain actions. That means that possibly actions depend upon a certain model state at the domain node and should only be executed if a condition concerning the model holds. The new command is rdfu:condition. RDF update commands that are placed within a rdfu:condition element are not immediately executed. First a condition that is expressed by a SPARQL ASK query is evaluated against the local knowledge base. Only if it holds, the inner commands are executed. The query is given as string contents of a attribute of the rdfu:condition element named rdfu:ask. Ony a ASK query is accepted. For convenience reasons, the prototype accepts ASK queries that only consist of the *condition pattern* of the query. A syntactically correct ASK query would look like

$$ASK\ \{\ \ll condition\ pattern \gg\ \}.$$

Of course that is accepted but it would also be accepted without the keyword ASK and the curly brackets. Note that a SPARQL query requires for an absolute resource URI, that means without prefix, to be embedded within angle brackets. But these have to be replaced by their corresponding XML entities as said above.

3.3 SPARQL Queries Evaluated against the OWL Model

The RDF Web Service wouldn't be useful if it was not possible to query the model held by it. So SPARQL queries may be sent to the Web Service as plain text content of an HTML request. These are evaluated against the local model. Only SELECT and ASK queries are accepted and answered by a variable binding in the usual XML markup of the MARS framework. Note that for an ASK query a variable named ask is bound to the answer. The answers are returned according to the following DTD:

```
<!ELEMENT variable-bindings (tuple+)>
<!ELEMENT tuple (variable+)>
<!ELEMENT variable ANY>
<!ATTLIST variable name CDATA #REQUIRED
          ref URI #IMPLIED> <!– variable has either ref or content–>
```

In contrast to plain XML where namespaces are not relevant, in the Semantic Web they play an important role. So it shouldn't be withheld that the elements of the variable bindings belong to the namespace http://www.semwebtech.org/lang/2006/logic#.

Example 8 *SPARQL queries are sent as plain text contents of an HTTP request to the web service. For example:*

```
prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
prefix rdfs: http://www.w3.org/2000/01/rdf-schema#
select ?sub where { ?sub rdf:type rdfs:Class }
```

The answer of such a query could look like this:

```
<logvars:variable-bindings xmlns:logvars="http://www.semwebtech.org/lang/2006/logic#">
  <logvars:tuple>
    <logvars:variable name="sub" ref="http://some.namespace#uri1"/>
  </logvars:tuple>
  <logvars:tuple>
```

```
    <logvars:variable name="sub" ref="http://some.namespace#uri2"/>
   </logvars:tuple>
    ...
  </logvars:variable-bindings>
```

3.4 Triggers

The main advantage of the RDF Web Service is to provide an infrastructure for triggers to react upon model modifications. With these triggers it is not only possible to raise application domain events on special model changes but they are essentially necessary for a semantically correct model change itself. That is because the model the web service holds does not only contain materialized facts but also derived ones. Dealing with information within the semantic web is a complex matter. When an application uses the relational data model, the application logic is practically outside the database and referential integrity should be sufficient for maintaining a consistent state. On the contrary, an RDF/OWL model holds not only facts but the model itself is enriched by logic and therefore application semantics which leads to the necessary of higher level modification capabilities. Rules have to be defined explicitly to ensure the consistency of the model and to react accordingly to semantic constraints.

Triggers may be registered to the web service by sending an HTTP message with a content that corresponds to the following syntax:

```
CREATE TRIGGER name
[ LET variable := value ]*
ON
  [
     [ [ INSERTION|DELETION|MODIFICATION ] OF property OF INSTANCE [ OF class ]? ]
   | [ [ CREATION|DELETION|MODIFICATION ] OF INSTANCE OF class ]
   | [ [ RETRACT|INSERT|UPDATE ] OF property OF INSTANCE [ OF class ]? ]
   | [ NEW CLASS ]
   | [ NEW PROPERTY [ OF class ]? ]
   | [ NEW PROPERTY OF INSTANCE [ OF class ]? ]
   | [ NEW STATEMENT ABOUT INSTANCE [ OF class]? ]
  ]
[ WHEN SPARQL-Anfrage ]?
DO
BEGIN
  [
     [ RDF update action ]
   | [ SEND(url, message) ]
   | [ RAISE EVENT( event in XML markup with variables ) ] ;
  ]*
END;
```

Now the semantics of triggers used in the RDF Web Service will be described. There are two different kinds of triggers used here. Firstly, there are direct triggers that react immediately upon the request of an update and secondly, there are indirect triggers that react upon a chance that was preformed within the model.

Immediate execution of direct triggers means that they are fired before the update itself is performed. These triggers are intended to support the requested update in being performed accordingly. That means the triggers ensure that an update will not violate the consistency of the model in perspective of the OWL model theory. Because these triggers do not react on actual changes of the model but only on the request of an modification the actions to be triggered are restricted to RDF update actions. Such triggers are defined as follows:

ON [INSERT|UPDATE|RETRACT] OF *property* OF INSTANCE [OF *class*]? Direct triggers react before an insert, update or retract operation of an instance (optionally of the specified class) is executed.

Example 9 *Consider a model with the property* hasPresident *that is defined as functional because it describes that a company has a unique president. Assume the model contains two different entities* x *and* y *and the statement* [c hasPresident x]. *The RDF update operation* insert (c, hasPresident, y) *would immediately cause an inconsistency since hasPresident is required to be functional. At this point a direct trigger may be used to delete the previous entry of* hasPresident *before the new one is inserted. Such a trigger could look like this:*

```
ON INSERT OF hasPresident
WHEN SELECT ?c ?x WHERE { ?c <hasPresident> ?x .
        FILTER( ?c = NEW.subject ) .
        FILTER( ?x != NEW.object ) }
DO
BEGIN
    retract (?c, hasPresident, ?x);
END;
```

Example 10 *Another example shows a different reason why these direct triggers are necessary. Consider a property* hasHusband *and the statements* [hasHusband owl:inverseOf hasWife], [Alice hasHusband Bob]*and* [Dan hasWife Carol]. *Although not contained as materialized facts both* [Bob hasWife Alice] *and* [Dan hasHusband Carol] *are derived by the reasoner and are therefore evaluated as true within the model.*
Assume now an RDF database that contains both [Emmy hasHusband Frank] *and* [Frank hasWife Emmy] *as facts. Deleting one of them has no effect since the reasoner will derive it from the other one. Thus a trigger*

```
ON RETRACT OF hasHusband
    % by an raising of this trigger the following variables are bound
    % OLD.subject := wife URI
    % OLD.property := hasHusband
    % OLD.object := husband URI
DO
BEGIN
    delete ( OLD.object, hasWife, OLD.subject );
END;
```

(and vice versa) is necessary to perform the deletion as intended. Only by removing both statements from the model any change becomes visible to the outside. Here it is essential that the trigger is fired upon the request of an update instead of a model change, because neither delete (Emmy, hasHusband, Frank) *nor* delete (Frank, hasWife, Emmy) *will change the model on their own.*

As said above, indirect triggers are raised after an actual model change. They are intended to implement the complex application logic and reactive behavior of the RDF web service. Therefore, in addition to RDF update actions, such triggers may also raise application domain events. Indirect triggers are defined as follows:

ON [INSERTION|MODIFICATION|DELETION] OF *property*
OF INSTANCE [OF *class*]? Such a trigger is raised if a property by the given name is inserted, updated or deleted from a resource (optionally of the specified class).

ON NEW PROPERTY OF INSTANCE [OF *class*]? is raised when any new property is inserted to a resource (optionally: to a specified class). In difference to the latter one, such a trigger reacts on any property change of an resource and not only to the change of a property with a certain name.

ON NEW STATEMENT ABOUT INSTANCE [OF *class*]? This trigger condition extends the latter one to any statement change of an instance. For example a change of the object of an property.

ON [CREATION|MODIFICATION|DELETION] OF INSTANCE OF *class* is raised when a resource of a given type is created, modified or deleted.

ON NEW CLASS is raised if a new class is introduced to the model.

ON NEW PROPERTY [OF *class*]? is raised when a new property is introduced to the model. This property has not to be assigned to any instance for such a trigger to be fired.

Example 11 *Assume that an RDF application node of a university is supposed to be able to list all publications that have been published by its members. That means all publications should be listed that were published at a time when the author was employed by that university. Note that that knowledge is independent from the current employer of the author. Publications are intended to be assigned to the employer university at the time of publication. Consider a model where a query* [my_university produced publication] *would lead to the intended information and statements like* [hans_peter works_for my_university], [hans_peter published publication_0815] *and* [works_for rdfs:domain Person] *are contained.*

The following trigger is an example how the knowledge could be collected:

```
ON INSERTION OF published OF INSTANCE OF Person
    % by an raising of this trigger the following variables are bound
    % NEW.subject := author URI
    % NEW.property := published
    % NEW.object := publication URI
WHEN SELECT ?U WHERE { NEW.subject <works_for>?U . ?U rdfs:type <university>}
DO
    insert ($U, produced, NEW.object) ;
END;
```

The condition part of a trigger is optional. But when used, it always has to start with the keyword WHEN followed by a valid SPARQL SELECT or ASK query. Internally nothing else happens than the evaluation of the given query using the Jena Framework. Variables that were bound within a SELECT query will be assigned to the variables in the action part respectively. If no variables are bound, or an ASK query results in false, the trigger will not fire.

As already shown in the examples, there are variables that are automatically bound to the resource URIs of the RDF statement that fired the trigger. NEW and OLD variables are bound that have the components subject, property and object. These components hold the corresponding statement URIs. When using a trigger that reacts on the creation of a new class, only a component class is available. On execution of a trigger that reacts on insertion of a new property to the meta data, the components of the NEW variable are property and domain.

The binding of the variables NEW and OLD is intuitive. NEW is bound to a statement after its creation or update and OLD to a statement that has been deleted or has been updated.

It is possible to bind variables at the beginning of a trigger definition. Such variables have to start with $. They are available in the condition and the action part. All RDF update requests can be raised by a trigger. For indirect triggers, also the sending of an HTTP message to a given URL is possible. Application domain events can also be raised by sending a specific form of message to a registered domain broker as described in Section 4.1. In the prototype, it is necessary to set the URL for the concerned Domain Broker in a configuration file. For more detail on how to configure the web service see Section 5.7. The event to be sent has to be in a valid XML markup. It may contain variables from the LET part or the condition part as well as the variables NEW and OLD as described above. A sequence of action may be defined separated by semicolons.

Chapter 4

Adaption of the RDF Web Service to an Application Domain Node

The central theme of this thesis is the adaption of the RDF web service described in the previous chapter to an actual application domain node within the MARS Framework. The prototype of the Web Service implemented by Elke von Lienen provides a good foundation. The triggers of the RDF Web Service can be used to define rules for performing low level updates and derivation of events upon model state changes. So they provide a mechanism for upward communication. To turn the service into a part of the MARS Framework, it was necessary to add a downward communication interface. That is the capability to map abstract application domain actions to low level *RDF update* actions like they are described in Section 3.1. Such high level actions are forwarded from the Domain Broker where the node is registered after they were raised by an ECA rule within the ECA Engine or sent to the broker directly.

The handling of these high level actions is implemented in a wrapper around the core Web Service as shown in Figure 4.0.1. That wrapper is addressed by a different URL to separate the functionalities.

Apart from this new interface, some minor changes had to be performed. Mostly they can be summarized as debugging of existing functionalities and minor changes in syntax and usage. For example, the syntax of the trigger definitions has changed. Note that all changes that were made in the web service are orthogonal to the functionalities that were described in Chapter 3. That means that everything said about that service also holds for the application domain node developed for this thesis.

4.1 Events raised within the Domain Node

An application domain node knows mainly three ways to communicate with the rest of the framework. It supports a SPARQL query interface where other nodes like domain brokers may post queries. The answer is a variable binding XML message as described in Section 3.3. Although it is possible and necessary for other nodes to query an application domain node in such a pull information manner, the intended communication within the framework is event-driven. Instead of asking the node whether something has changed from outside it is the node itself that informs the concerning domain about local updates that were performed.

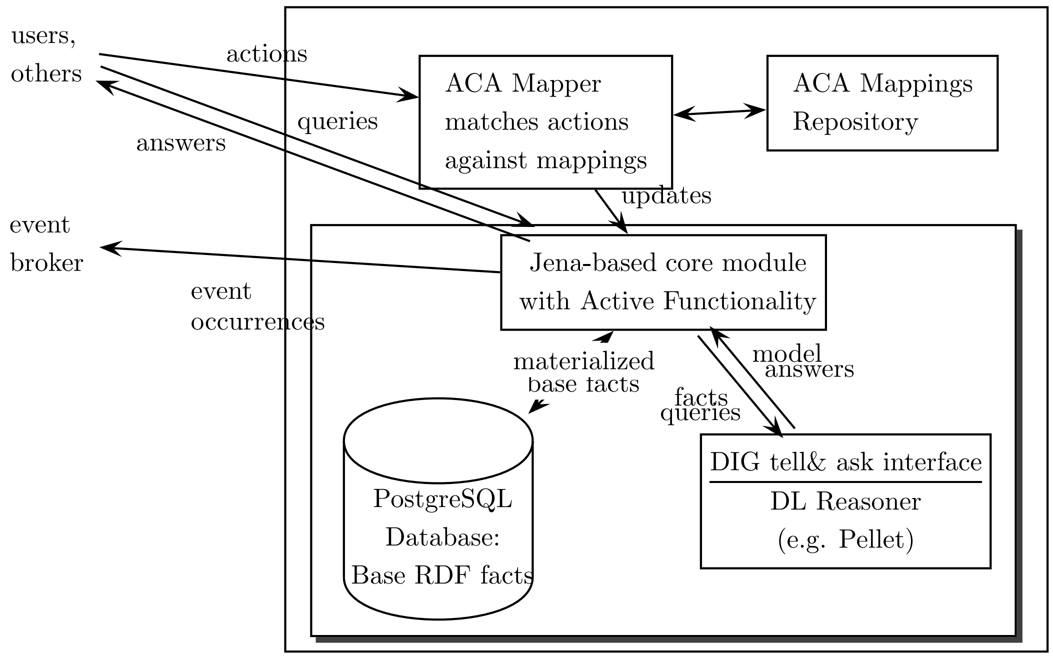


Figure 4.0.1: Architecture of the Domain Node (taken from [1])

The information that is communicated within the Semantic Web usually is on a semantically higher level than simple update operations. That is why the domain node needs an mechanism to define ECE Rules that map simple data model updates to the application domain level. Such an infrastructure is realized by the triggers of the Active Web Service. As described earlier in Section 3.4, there exists a certain trigger action especially for this task. The latter is: raise event (<<XML content>>), where the content is intended to be an application domain event in XML markup. Only indirect triggers that react upon model changes in the node may raise events. Note that the events have to be defined within the Application Domain Ontology to be recognized as such. To define an event named *uni:professor-hired* statements like

```
[ uni:professor-hired rdfs:subClassOf uni:Event ]
[ uni:Event rdfs:subClassOf mars:Event ]
```

have to be known to the ontology as described in Section 2.3.

Example 12 *Consider an application domain node in the university ontology. Assume that every time a new professor is hired in one of the participating universities, an event* uni:professor-hired *should be raised. The employment of a person may be stored using the predicate* uni:employed*. So an event should be raised when a statement* [my-university-uri uni:employed some-person-uri] *is inserted into the local model. A trigger for that task could look like that:*

```
ON INSERTION OF uni:employed OF INSTANCE OF uni:University
WHEN ASK { <NEW.object> <rdfs:type> <uni:Professor>}
DO
BEGIN
RAISE EVENT (
        <uni:professor-hired xmlns:uni="http://localhost/test.owl#"
           professor="NEW.object" university="NEW.subject"/>
);
END;
```

So the emitting of high level events is fully covered by the trigger functionality implemented within the Active Web Service - at least for events that should be raised upon model state changes. But the second major task of an application domain node, the executing of high level actions was not supported by the Web Service described in the previous chapter. The implementation of such an interface is the main topic of this thesis.

4.2 Action Interface of the Application Node

An application domain node needs an interface for the execution of high level application domain actions. As described earlier, these actions are small RDF/OWL graphs that are transmitted between nodes participating in a Semantic Web network. Usually actions are the outcome of the evaluating of an ECA rule. That means the behavior that is triggered upon certain events that occurred within the network. Actions are distributed by Action Brokers that are a component of the Domain Brokers.

The actions communicated within the Semantic Web are on the application domain level, just like the events. So analogously to ECE Rules that map data model events to the higher level there have to be ACA mappings that map high level actions to data model updates. The following sections describe a proposal of how such an ACA mapping can be implemented by a Domain Service.

4.2.1 Syntax and Semantics of Actions

Within the MARS Framework, every domain action has to be defined as a class that extends the mars:Action concept. That is because an actual action that is distributed within the Semantic Web represents an instance of the corresponding action class. Take for example the possible university domain ontology action uni:add-publication that may be used to distribute the insertion of a new publication within a network of participating university application nodes. Such an action has to be declared in the University Domain Ontology with statements like:

```
[ uni:add-publication rds:subClassOf uni:Action ]
[ uni:Action rdfs:subClassOf mars:Action ]
```

An actual instance of that action may look like the following XML fragment.

```
<uni:add-publication xmlns:uni="http://localhost/test.owl#">
   <uni:published-by rdf:resource="university-URI" />
   <uni:title>some title</uni:title>
   <uni:abstract> ... </uni:abstract>
   <uni:author rdf:resource="first-author-URI" />
   <uni:author rdf:resource="second-author-URI" />
</uni:add-publication>
```

Such an XML fragment can be interpreted as RDF/XML definition and therefore be validated with the W3C-RDF-Validator [21] when it is nested within an rdf:RDF root element. Doing so will show that indeed an anonymous resource of type http://localhost/test.owl#add-publication is described with the following N3 statements:

```
[ genid:A333958 rdf:ype http://localhost/test.owl#add-publication ]
[ genid:A333958 http://localhost/test.owl#published-by http://base/university-URI ]
[ genid:A333958 http://localhost/test.owl#title "some title" ]
[ genid:A333958 http://localhost/test.owl#abstract "..." ]
[ genid:A333958 http://localhost/test.owl#author http://base/first-author-URI ]
[ genid:A333958 http://localhost/test.owl#author http://base/second-author-URI ]
```

Figure 4.2.1 shows that resource as graph.

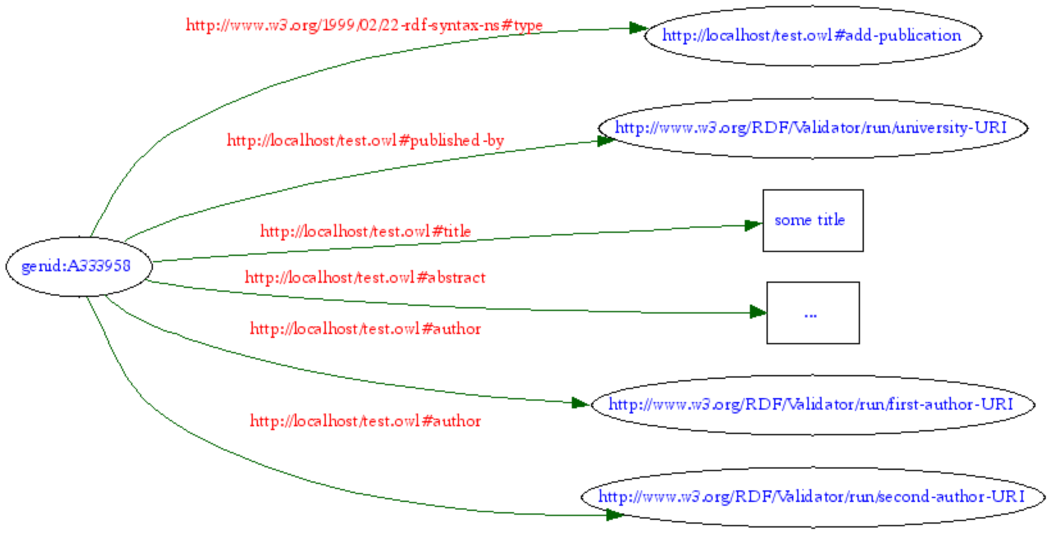

http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://localhost/test.owl#add-publication
http://localhost/test.owl#published-by
http://www.w3.org/RDF/Validator/run/university-URI
http://localhost/test.owl#title
some title
genid:A333958
http://localhost/test.owl#abstract
...
http://localhost/test.owl#author
http://www.w3.org/RDF/Validator/run/first-author-URI
http://localhost/test.owl#author
http://www.w3.org/RDF/Validator/run/second-author-URI

Figure 4.2.1: RDF Graph of the Action Instance

A domain node that receives such an action is requested to check if it is interested in publications that are published by that university and if so, add this particular one to its model. Therefore an ACA mapping has to provide a possibility to query the local model to decide whether a certain action instance is relevant.

4.2.2 Actions with Variable Bindings

Actions are not necessarily sent fully instantiated but as action patterns with variable bindings instead. That is because internally within the ECA Engine with its Event Detection Services, event occurrences are also communicated in form of variable bindings. Action definitions of ECA Rules declare input variables that use variables from the event pattern and the optional query part of a rule. When an action is actually triggered, it is evaluated by the Action Engine where the variable bindings are usually evaluated so that the action patterns are mapped to instances of the action that are forwarded to the Domain Brokers and finally to Domain Services to be executed.

Therefore domain nodes have to be able to bind variables to action patterns. Such an action pattern is of the following form:

```
<travel:delay-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
    flight="{$flight}" date="{$date}">

<logvars:variable-bindings xmlns:logvars="http://www.semwebtech.org/lang/2006/logic#">
    <logvars:tuple>
        <logvars:variable name="flight">LH1234</logvars:variable>
        <logvars:variable name="date">20060523</logvars:variable>
    </logvars:tuple>
    <logvars:tuple>
        <logvars:variable name="flight">GR5432</logvars:variable>
        <logvars:variable name="date">20060523</logvars:variable>
    </logvars:tuple>
</logvars:variable-bindings>
```

Variables that are used in the pattern have to start with an $ and may be engulfed within curly brackets.

4.3 ACA Mapping Wrapper

It is the ACA wrapper for application domain actions that extends the Reactive Web Service to an application domain node. Such a wrapper has to provide several tasks. Of course its main task is to map domain actions to the data model level. Additionally it has to provide administrative functionalities for ACA mapping registration. On top of that the ACA interface supports a simple action algebra by allowing the execution of action sequences and the binding of variables to action patterns.

But what exactly is such a mapping? As it is implemented in the prototype, an ACA mapping is the transformation of a received XML action message to an XML document that is a markup for instructions that can natively be executed at a domain node. These instructions are mostly XML mark-upped RDF update commands that were introduced in Section 3.1. Therefore the output of a mapping is interpreted as an action itself. Note that the MARS Framework only requires the node to provide the action translation according to the given Application Ontology. How that is done is not part of the framework but lies in the responsibility of the provider of the node implementation. The usage of an XML markup for the RDF update language suggests itself because XML is easy to parse and with the Document Object Model (DOM) there is a well documented API [4].

What has not been covered is how exactly the transformation process is done. Because the prototype uses an XML language for actions on the data model level, the task of a mapping is to transform an XML document into another XML document. That is a common task in the W3C XML world and therefore already well covered. The prototype both accepts ACA mappings that use either an XQuery expression [22] or an XSLT style-sheet [23]. These W3C Recommendations both implement the needed functionality. XQuery is a clause based query language for XML that supports the generation of an XML document as a result. XSLT enables to define style sheets that transform an input XML document into a requested format using a tree walk formalism.

Because the result of a mapping is interpreted as an action it is possible to define an ACA mapping that raises other application domain actions within the specific domain node instead of data level updates.

As the term Action-Condition-Action already suggests, ACA mappings have to provide the possibility to formulate conditions that restrict the execution of an action at a certain domain node. It has to be able to place conditions on the action message that was received and on the state of the local knowledge base. The first requirement is implicitly fulfilled because both XQuery and XSLT support placing very expressive conditions on the XML input they process. But neither one of them support querying an RDF knowledge base. Therefore, the RDF update language was extended by the possibility to make the execution of update commands conditional on a certain model data state.

The general semantics of the RDF update language has already been described in Section 3.2. It has been developed to enable the modification of an RDF Web Service. Therefore, it provides a method for the ACA wrapper to formulate a data model update. Because the updates requested by an application level action are probably complex and include the change or insertion of multiple RDF statements, the processing of sequences of these commands has to be possible.

Now the example ACA mapping from Section 2.4 is revisited. To refresh the memory, this example maps the application level action uni:register-students to the data model level. To do so for every student that is listed in the action message, an RDF statement is inserted into the model that associates the student to the lecture. But before doing so the local knowledge base is queried to check if the given lecture is already known. Because if it is not, the action is of no interest for the node. Using XQuery as translation language, such a mapping could look like the following:

```
declare namespace uni = "http://localhost/test.owl#";
let $lecture_uri := //uni:register-students/uni:lecture/text()
return
<rdfu:condition xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  rdfu:ask="&lt;{$lecture_uri}&gt; rdf:type uni:Lecture">
  {
    for $student in //uni:register-students[./uni:lecture]/uni:student
    return
    <rdfu:insert>
      <rdf:subject rdf:resource="{$lecture_uri}" />
      <rdf:predicate rdf:resource="http://localhost/test.owl#has-student" />
      <rdf:object rdf:resource="{$student/text()}" />
    </rdfu:insert>
  }
</rdfu:condition>
```

This example uses an rdfu:condition element to check if the local knowledge base knows a resource of type uni:Lecture with the URI given in the high level action message. That is done by the query:

ASK { <$lecture_uri> rdf:type uni:Lecture }.

If it is found to be true, the rdfu:insert commands within the rdfu:condition element are executed.

Now after the general architecture of the wrapper has been described the translation part of ACA mappings should be described, in more detail. That will be done in the following two sections.

4.3.1 ACA Mappings using XQuery

Both XQuery and XSLT are equally suitable to transform XML documents. Although XSLT is especially developed for this task, the prototype prefers XQuery to formulate ACA mappings. That is mainly because XQuery is declarative and more straightforward. Additionally, it offers to import function modules. That feature is used within the prototype. Before a mapping using XQuery is executed, several predefined XQuery modules are added to the mapping and default namespaces are declared. There are prefix namespace bindings that are usually used in every mapping. So the prefixes rdf, rdfs, owl and rdfu are respectively bound to their namespaces. That means that these prefixes do not have to be bound explicitly.

The generation of the XML representation of rdfu commands is provided by functions similar to:

- **rdfu:stringLiteral($obj)**
- **rdfu:intLiteral($obj)**
- **rdfu:doubleLiteral($obj)**
- **rdfu:booleanLiteral($obj)**
- **rdfu:insert($sub, $pre, $obj)**
- **rdfu:delete($sub, $pre, $obj)**
- **rdfu:retract($sub, $pre, $obj)**
- **rdfu:assert($sub, $pre, $obj)**
- **rdfu:delete-resource($sub)**
- **rdfu:update-subject($sub, $pre, $obj, $new)**
- **rdfu:update-predicate($sub, $pre, $obj, $new)**
- **rdfu:update-object($sub, $pre, $obj, $new)**
- **rdfu:rename($old, $new)**
- **rdfu:rename-property-of-class($old, $new, $class)**

The insert update statement in the example from above can be written as follows:

```
rdfu:insert ( $lecture_uri, http://localhost/test.owl#has-student,
              $student/text() )
```

The first four functions in the list above format a given object literal value as typed RDF literal. For example a call of rdfu:intLiteral(10) will return

```
10^^<http://www.w3.org/2001/XMLSchema#int>.
```

Additionally to the simpler syntax, the RDF update functions have the advantage that they implicitly implement a ACA condition. That is because the functions only return an XML RDF update element if the value of the given $obj or $new parameter exists. If the corresponding value is empty, the update command is not executed because nothing has to be updated. Using this feature, it is possible to define ACA mappings that do not have to check explicitly for the existence of certain text or attribute nodes in the received high level action.

Example 13 *XQuery mappings can be very simple containing only XPath expressions. Take for example a ACA mapping definition as follows:*

```
declare namespace uni = "http://localhost/test.owl#";
(
   rdfu:insert (uni:add-student/@uri,
                "http://localhost/test.owl#name",
                rdfu:stringLiteral(uni:add-student/uni:name/text())),
   rdfu:insert (uni:add-student/@uri,
                "http://localhost/test.owl#city-of-birth",
                uni:add-student/uni:city-of-birth/text()),
   rdfu:insert (uni:add-student/@uri,
                "http://localhost/test.owl#date-of-birth",
                uni:add-student/uni:date-of-birth/text()),
   rdfu:insert (uni:add-student/@uri,
                "http://localhost/test.owl#mobile-phone",
                uni:add-student/uni:mobile/text())
)
```

If a domain node with such an ACA mapping receives the action message with all properties that are mentioned in the mapping like:

```
<uni:add-student xmlns:uni="http://localhost/test.owl#"
   uri="stud-uri">
  <uni:name>Peter Mueller</uni:name>
  <uni:city-of-birth>Berlin</uni:city-of-birth>
  <uni:date-of-birth>23.05.1984</uni:date-of-birth>
  <uni:mobile>017712345678</uni:mobile>
</uni:add-student>
```

then for all these properties there are rdfu:insert *returned by the mapping. But if an action message misses one ore more of these properties like:*

```
<uni:add-student xmlns:uni="http://localhost/test.owl#"
   uri="stud-uri">
  <uni:name>Peter Mueller</uni:name>
</uni:add-student>
```

then the same mapping returns only rdfu:insert *elements for the properties that are present. In this example only*

```
<rdfu:insert xmlns:rdfu="..." xmlns:rdf="...">
   <rdf:subject rdf:resource="stud-uri" />
   <rdf:predicate rdf:resource="http://localhost/test.owl#name" />
   <rdf:object>Peter Mueller</rdf:object>
</rdfu:insert>
```

is returned by the mapping.

The ACA module definitions can be found in Appendix C. It is intended that there are function module definitions for application domains. Every Domain Ontology should describe how the URIs for the concepts defined in these ontologies are created. The prototype expects such definitions to be made within Domain Ontology XQuery modules. Therefore, the prototype supports the submission of new function modules as described later. These modules should contain functions for the creation of URIs and other Domain specific operations. Note that these modules have to be imported explicitly in an ACA mapping

4.3.2 ACA Mappings using XSLT

The prototype also supports the definition of ACA mappings as XSLT style-sheet. This method is much less comfortable within the prototype because there are no simplifications implemented. For example the XSLT version of the short XQuery mapping.

```
declare namespace uni = "http://localhost/test.owl#";
for $stud in uni:add-student[./uri]
let $stud_uri := concat("http://localhost/test.owl#", $stud/uri/text())
let $name := $stud/uni:name/text()
return
(
   rdfu:insert ($stud_uri, rdf:ns("type"), "http://localhost/test.owl#Student"),
   rdfu:insert ($stud_uri, rdf:ns("type"), "owl:Thing"),
   rdfu:insert ($stud_uri, "http://localhost/test.owl#name", $name)
)
```

would look like the following:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
  xmlns:uni="http://localhost/test.owl#">
  <xsl:template match="uni:add-student[./uri]">
    <xsl:variable name="stud_uri"
      select="concat('http://localhost/test.owl#', uri)" />
    <xsl:variable name="name" select="uni:name" />
    <rdfu:insert xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
      <rdf:subject><xsl:value-of select="$stud_uri" /></rdf:subject>
      <rdf:predicate>rdf:type</rdf:predicate>
      <rdf:object>http://localhost/test.owl#Student</rdf:object>
    </rdfu:insert
    <rdfu:insert xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#">
      <rdf:subject><xsl:value-of select="$stud_uri" /></rdf:subject>
      <rdf:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:predicate>
      <rdf:object>http://www.w3.org/2002/07/owl#Thing</rdf:object>
    </rdfu:insert>
    <xsl:if test="not(empty($name))">
      <rdfu:insert xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#">
        <rdf:subject><xsl:value-of select="$stud_uri" /></rdf:subject>
        <rdf:predicate>http://localhost/test.owl#name</rdf:predicate>
        <rdf:object><xsl:value-of select="$name" /></rdf:object>
      </rdfu:insert>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

4.4 Administration of the ACA Wrapper

An important aspect of a description of the ACA wrapper is of course its administration. There has to be a way to register ACA mappings and XQuery modules that contain domain specific functions. The prototype provides certain node update actions for this task. Note that these actions are not intended to be used by anyone other than the provider of the particular application domain node. The node update actions supported by the prototype are implementation dependent. That means they are not part of the MARS Framework but simply implement needed functionalities of the particular domain node implementation.

There are different categories of node update actions. They all are syntactically equal to application domain actions and RDF update actions because the ACA wrapper already provides the possibility to process this kind of actions. Firstly, there are those that add or delete node components. They are as follows:

- **applnode:register-aca-mapping** - Registraction of an ACA mapping
- **applnode:delete-aca-mapping** - Removal of an ACA mapping
- **applnode:register-aca-function-library** - Registration of an XQuery module
- **applnode:delete-aca-function-library** - Removal of such a module
- **applnode:register-trigger** - Registration of an RDF data model trigger as described in 3.4
- **applnode:delete-trigger** - Removal of such a trigger
- **applnode:read-rdf** - Update of the knowledge base by loading given RDF statements

Most of these XML mark-upped actions expect the XML elements to have plain text content. Only applnode:register-aca-mapping and applnode:read-rdf also support content that is given in XML. For applnode:read-rdf that is an ontology in RDF/XML. An ACA mapping may be given as

XSLT style-sheet and therefore as XML child element of applnode:register-aca-mapping. Instead of defining the corresponding contents directly, it is possible for each of these actions to specify an attribute named applnode:url that contains the URL[1] from where the actual content will be loaded. The actions will now be described in detail:

An applnode:register-aca-mapping action needs several parameters. Every mapping has a unique name. An Element named applnode:name may be used to specify that name. Otherwise a default name is used. The application domain action that this particular mapping is intended for has to be specified. For that task there are the two elements applnode:action-namespace and applnode:action-name. This approach is chosen because it avoids the necessity of string parsing to separate namespace and local part of the action URI. Another necessary parameter is the language used for the mapping definition. The element applnode:language is expected to have either "xquery" or "xslt" as content. Optionally, an owner resource for the mapping may be specified. It is stored within the database. As mentioned above, an applnode:register-aca-mapping action either needs an element applnode:url with the location of the mapping content or the mapping has to be given as plain text or XML content. So such an action has the following form:

```
<applnode:register-aca-mapping
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#">
  <applnode:name> ... </applnode:name>
  <applnode:action-namespace> ... </applnode:action-namespace>
  <applnode:action-name> ... </applnode:action-name>
  <applnode:language> ... </applnode:language>
  [<applnode:url> ... </applnode:url>]
  [<applnode:owner-resource> ... </applnode:owner-resource>]
  ... ACA mapping given in XQuery or XSLT ...
</applnode:register-aca-mapping>
```

The other node update actions are simpler. An XQuery module may be registered by embedding the module definition in an applnode:register-aca-function-library element. Additionally, a unique applnode:name has to be given. That parameter is also the only parameter of applnode:delete-aca-mapping, applnode:delete-aca-function-library and applnode:delete-trigger. The action applnode:register-trigger takes only a trigger definition as plain text content without any other parameters. Finally, applnode:read-rdf can be used to directly change the knowledge base by sending an ontology description that is added to the local model of the node. The language that ontology is described in has to be specified by the parameter applnode:input-language . The accepted input formats are "RDF/XML", "RDF/XML-ABBREV", "N-TRIPLE", "N3-PLAIN", "N3-PP", "N3-TRIPLE" and "N3". The RDF data has to be given as contents of the applnode:read-rdf element or of course via a URL as described above.

On top of the node update actions already mentioned there are some more that are not responsible for changes within the node but are nevertheless needed. These are:

- **applnode:actionSequence** - Implements a sequence of actions to be executed in the ACA wrapper. All child elements of that XML node are interpreted as action and executed in

[1] Unified Resource Locator

document order.

- **applnode:nodeSequence** - Similar to applnode:nodeSequence with the difference that child elements are not necessarily interpreted as plain action.
- **applnode:dump** - Returns a dump of the current domain node. There are several modes to choose from.
- **applnode:describe-service** - Implements the Domain Service task:
  http://www.semwebtech.org/2006/mars/domain-node#give-service-description
- **applnode:query** - Enables the domain node to receive a SPARQL query. Therefore it is an implementation of:
  http://www.semwebtech.org/2006/mars/domain-node#receive-query
- **applnode:raise-event** - Implements mars:raise-event
- **applnode:raise-directed-event** - Implements mars:raise-directed-event

The prototype implementation of the ACA wrapper ensures that the result of an ACA mapping is a valid XML document and therefore can be parsed as such. This is done by placing it into an applnode:actionSequence element. That is done implicitly because it resembles the fact that it is an action sequence that the mapping should produce. Any sequence of elements send to the action interface is in principle interpreted as sequence of actions. If the parsing of the message received by the wrapper fails, the message is wrapped into a applnode:nodeSequence element and parsed again. The semantics of applnode:nodeSequence is almost the same as the meaning of applnode:actionSequence except that the contents of an applnode:nodeSequence element may be an action pattern with variable bindings. That is checked by searching for a logvars:variable-bindings sub element. If such exists, the whole message is interpreted as application domain action and the variables are bound. Any element of the message apart from the logvars:variable-bindings is treated as action pattern.

Using applnode:dump it is possible to get a dump file. Practically, the prototype returns an applnode:actionSequence containing node update actions. These represent the action instances that are needed to clone the particular node concerning its ACA mappings, ACA XQuery modules, triggers and knowledge base content. The applnode:dump action uses two parameters. These are applnode:mode and applnode:target-url. Using applnode:mode it is possible to specify what should get dumped. As contents it accepts “rdf”, “aca-mapping” “aca-function-library”, “trigger” and “all”. A combination of these keywords separated by whitespace is possible. By the parameter applnode:target-url a URL may be optionally given where the dump file is sent to via HTTP.

That is also the case for applnode:describe-service where it is also possible to give an additional target using applnode:target-url.

Apart from the RDF update actions, an ACA mapping may use mars:raise-event or mars:raise-directed-event as reaction upon an application domain action. These two actions lead to the raising of events defined as XML contents of the elements. For mars:raise-event that means that its child elements are sent to the Event Broker that is registered with the domain node. mars:raise-directed-event needs the parameter target-url where the content is sent to, respectively. It does not matter if these two actions are used with mars or applnode as namespace because the two different signatures of these actions are treated equally.

Indeed, the namespaces of all node update actions may be ignored. If any namespace is used, it has to be applnode or mars for mars:raise-event respectively. But if no namespace is used at all, the actions are identified correctly by their local names. That means for example that the prototype reacts equally when receiving one of the actions <applnode:describe-service /> or <describe-service /> and returns the Service Description of the node as RDF/XML.

Just like for the RDF update actions there is a default XQuery module that simplifies the usage of node update actions. That module is listed under 5.5. Naturally it is also automatically imported and therefore binds the applnode prefix. Finally there is a basic ACA module that is imported in every XQuery mapping. So far that module has only one function. It has the signature aca:getProperty($node as element(), $name as xs:string). Using this function, it is possible to get a property of the given name from an element node. That property may either be given as attribute or as sub-element. So when using this function to get the content of action properties it does not matter if these are given as attribute or element. This may be used in ACA mappings to achieve a higher compatibility for differently mark-upped application domain actions. Another convenience functionality is provided by the functions owl:ns($name as xs:string), rdfs:ns($name as xs:string) and rdf:ns($name as xs:string). These may be used to phrase an absolute URI in the corresponding namespace with the value of the $name argument as local part.

4.5 Task Description of the Domain Node Prototype

As already mentioned in Section 2.3, a domain node has to implement the following tasks:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl"
     xmlns=http://www.semwebtech.org/2006/mars#"
     xml:base=http://www.semwebtech.org/2006/mars">
     <rdf:Description rdf:about="#DomainService">
         <meta-provides-task rdf:resource="/domain-node#receive-query"/>
         <meta-provides-task rdf:resource="/domain-node#receive-action"/>
         <meta-provides-task rdf:resource="/domain-node#give-service-description"/><!– opt –>
     </rdf:Description>
</rdf:RDF>
```

The prototype provides these tasks according to the following task description:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns="http://www.semwebtech.org/2006/dsr#"
   xml:base="www.semwebtech.org/mars"
   xmlns:mars="www.semwebtech.org/mars#">
   <mars:DomainService  rdf:about="http://localhost:8080/semweb/uni-service"
     <mars:uses-domain rdf:resource="http://localhost/test.owl#"/>
     <has-task-description>
       <TaskDescription>
         <describes-task rdf:resource=" /domain-node#receive-query" />
         <provided-at rdf:resource=" http://localhost:8080/semweb/rdfserver/services" />
```

```
          <input>element query</input>
          <variables>no</variables>
          <communication-mode>synchronous</communication-mode>
        </TaskDescription>
      </has-task-description>
      <has-task-description>
        <TaskDescription>
          <describes-task rdf:resource=" /domain-node#receive-action" />
          <provided-at rdf:resource=" http://localhost:8080/semweb/rdfserver/actions" />
          <input>item *</input>
          <variables>*</variables>
        </TaskDescription>
      </has-task-description>
      <has-task-description>
        <TaskDescription>
          <describes-task rdf:resource=" /domain-node#give-service-description" />
          <provided-at rdf:resource=" http://localhost:8080/semweb/rdfserver/services" />
          <input>element describe-service</input>
          <variables>no</variables>
          <communication-mode>synchronous</communication-mode>
        </TaskDescription>
      </has-task-description>
    </mars:DomainService >
</rdf:RDF>
```

Chapter 5

Implementation

This chapter deals with the description of the ACA wrapper implementation. After a short overview of technologies used, the interaction of important components will be described. This is followed by the communication interface of the action wrapper and a short description of the extended Web Service Client.

5.1 Technologies

The ACA wrapper is implemented in Java [5]. Specifically, the Web Service uses the Servlet technology. The wrapper is integrated by adding a new Servlet to the Web Service that extends its functionality. The Servlet structure will be described later in Section 5.3.

The actual translation of XML high level actions to model updates is provided by XQuery or XSLT style-sheets. These technologies are implemented in the SAXON package that can be found at [7]. The wrapper uses SAXON 8.8 which supports XPath 2.0, XSLT 2.0, and XQuery 1.0. XQuery is accessed via the SAXON implementation of XQJ (XQuery API for Java)[1].

5.2 Architecture

Along with the introduction of the ability to process application domain actions, the package structure of the Web Service developed in [19] was renewed.
Now there are the following packages :

- `org.semwebtech.applnode` - Base package with ModelServer class
- `org.semwebtech.applnode.aca` - ACA wrapper main package
- `org.semwebtech.applnode.aca.database` - Database sub package for ACA transformation
- `org.semwebtech.applnode.servlet` - Servlet classes
- `org.semwebtech.applnode.trigger` - Trigger package
- `org.semwebtech.applnode.util` - Utility classes used in several packages
- `org.semwebtech.applnode.xml` - XML utilities (including XQuery and XSLT processors)

[1]currently published as "Early Draft Review 2" (JSR 225) at http://jcp.org/en/jsr/detail?id=225.

5.2.1 Core Architecture of the ACA Wrapper

The core class of the ACA wrapper is `org.semwebtech.applnode.aca.Action`. It is one of the few public classes of the `aca` package and provides all interfaces needed to execute an application domain action. Note that the action wrapper uses the DOM API [4]. An XML action that should be executed using the wrapper has to be parsed into a DOM document.

The gateway for action execution is the static method

```
Action.executeActionNode(ActionTracer, ModelServer, Element).
```

The element argument is expected to be a parsed XML element representing an action instance. The class `org.semwebtech.applnode.ModelServer` is the core of the RDF Web Service. It encapsulates the knowledge base and provides the update mechanisms including the trigger functionality. An instance of this class has to be given to the wrapper, because data model updates that resolve from the application domain action execution are directly executed. It would be possible to communicate with the `ModelServer` over HTTP but the prototype directly calls the methods responsible for the model update rather than sending update messages. The first argument of the `executeActionNode` method is an instance of an utility class that logs the action execution. When executing an application domain action, usually several update instructions are triggered within the `ModelServer`. Each of these is logged by the `org.semwebtech.applnode.aca.ActionTracer` instance. Of course that is not necessary for the execution of high level actions because Actions produce no result but only side effects in the form of model changes or the sending of asynchronous messages if events are raised by a mapping. The logging is needed for debugging capabilities. The action interface of the extended Web Service optionally returns a result in the form of an XML document containing the logged actions.

For the actual execution of actions that are given to the `executeActionNode` method, the class `org.semwebtech.applnode.aca.ActionExecuter` is used internally. This class provides a set of `org.semwebtech.applnode.aca.ActionImplementation` objects, each of which is responsible for the execution of an action with a certain action URI. These action URIs are for example the RDF update actions like http://www.semwebtech.org/lang/2006/rdfupdate#insert or the domain node actions like http://www.semwebtech.org/2006/application-node#actionSequence etc. For each of these actions, there is an implementation class that extends `ActionImplementation`. Basically, the `ActionExecuter` iterates over the known action implementing objects, checking if their URI matches the URI of the action it is about to execute. Every `ActionImplementation` has to implement the method `execute(ActionTracer, ModelServer, Element)` that finally is responsible for the execution of that particular type of action. At this point, the element node is evaluated. It should be mentioned that the URI of an action instance is identified by the namespace and local name of the element argument given to `executeActionNode`.

Note that application domain actions may have arbitrary URIs. That is why there is a special `ActionImplementation` class named

```
org.semwebtech.applnode.aca.AbstractNodeActionImpl.ApplicationDomainAction
```

that is not responsible for a static URI but any URI that has been registered with an ACA mapping. Such registration may be checked with the method

```
MappedDomainActionRegistry.isMappedApplicationAction(String ns, String name).
```

On execution of an application domain action, the element representing the action is given to an `org.semwebtech.applnode.aca.AcaMappingProcessor` instance as shown in Figure 5.2.1. There, all ACA mappings that are registered for the current action URI are used to transform the action. The method `Iterator<AcaMapping> iterator(String ns, String name)` from the class `org.semwebtech.applnode.aca.database.AcaMappingRegistry` is used to iterate over all relevant ACA mappings. The actual tranformation is done by the utility classes `XQuery` and `Xslt` from the package `org.semwebtech.applnode.xml`. The result of `transformAction(Element)` of `AcaMappingProcessor` is again a DOM element. It is ensured that it is an actionSequence element containing all RDF update actions etc. that are intended for the execution of the particular Domain Action. At this point, the method

```
ActionExecuter.executeNode(ActionTracer, ModelServer, Element)
```

is simply called again with the original `ActionTracer` and `ModelServer` arguments and the new actionSequence element as third argument. At his run of that method, the `ActionImplementation` objects of the RDF update actions contained in the http://www.semwebtech.org/2006/application-node#actionSequence are called. They directly perform the requested model changes using the methods provided by the `ModelServer` argument.

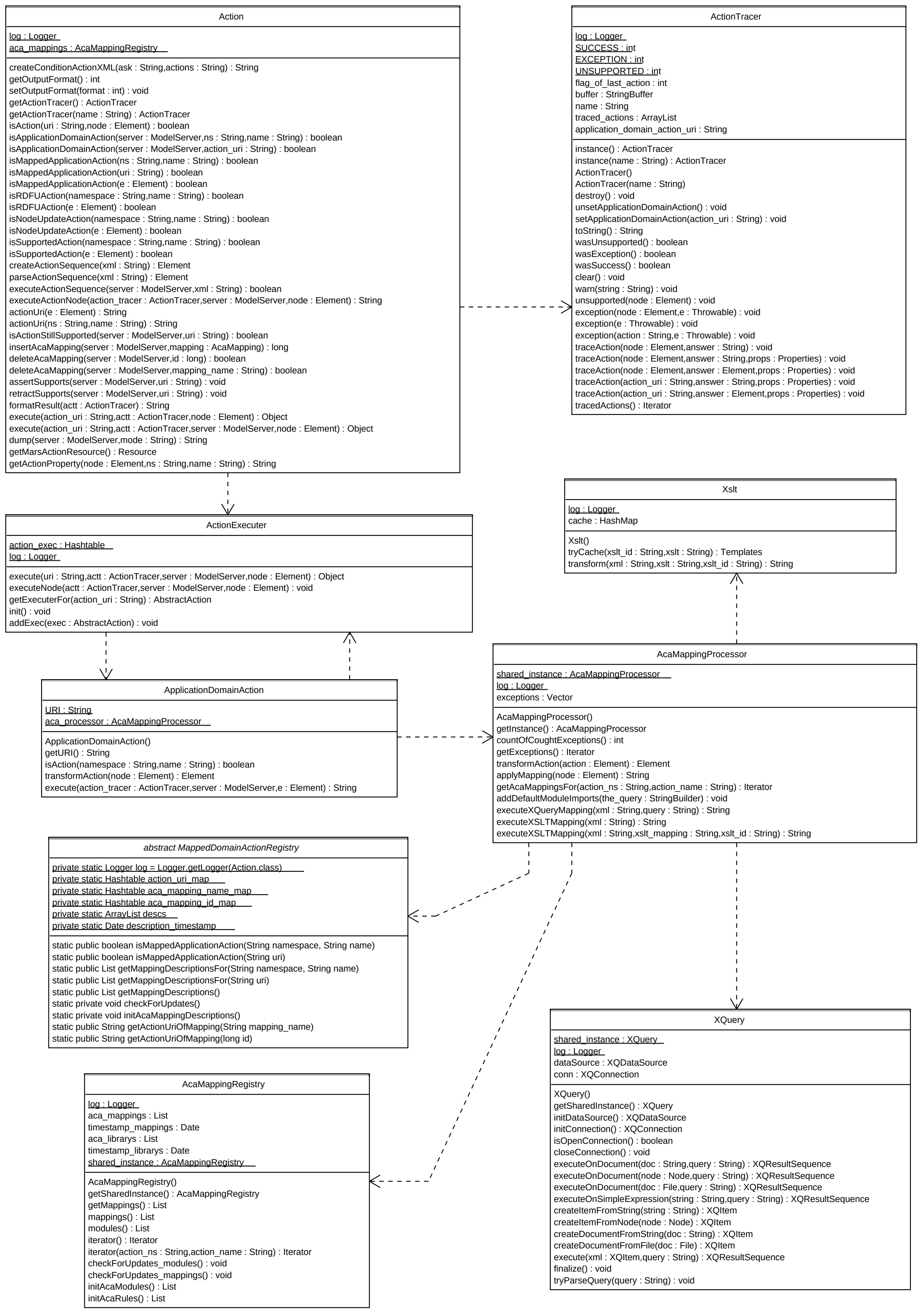

Action
log : Logger
aca_mappings : AcaMappingRegistry
createConditionActionXML(ask : String,actions : String) : String
getOutputFormat() : int
setOutputFormat(format : int) : void
getActionTracer() : ActionTracer
getActionTracer(name : String) : ActionTracer
isAction(uri : String,node : Element) : boolean
isApplicationDomainAction(server : ModelServer,ns : String,name : String) : boolean
isApplicationDomainAction(server : ModelServer,action_uri : String) : boolean
isMappedApplicationAction(ns : String,name : String) : boolean
isMappedApplicationAction(uri : String) : boolean
isMappedApplicationAction(e : Element) : boolean
isRDFUAction(namespace : String,name : String) : boolean
isRDFUAction(e : Element) : boolean
isNodeUpdateAction(namespace : String,name : String) : boolean
isNodeUpdateAction(e : Element) : boolean
isSupportedAction(namespace : String,name : String) : boolean
isSupportedAction(e : Element) : boolean
createActionSequence(xml : String) : Element
parseActionSequence(xml : String) : Element
executeActionSequence(server : ModelServer,xml : String) : boolean
executeActionNode(action_tracer : ActionTracer,server : ModelServer,node : Element) : String
actionUri(e : Element) : String
actionUri(ns : String,name : String) : String
isActionStillSupported(server : ModelServer,uri : String) : boolean
insertAcaMapping(server : ModelServer,mapping : AcaMapping) : long
deleteAcaMapping(server : ModelServer,id : long) : boolean
deleteAcaMapping(server : ModelServer,mapping_name : String) : boolean
assertSupports(server : ModelServer,uri : String) : void
retractSupports(server : ModelServer,uri : String) : void
formatResult(actt : ActionTracer) : String
execute(action_uri : String,actt : ActionTracer,node : Element) : Object
execute(action_uri : String,actt : ActionTracer,server : ModelServer,node : Element) : Object
dump(server : ModelServer,mode : String) : String
getMarsActionResource() : Resource
getActionProperty(node : Element,ns : String,name : String) : String
ActionTracer
log : Logger
SUCCESS : int
EXCEPTION : int
UNSUPPORTED : int
flag_of_last_action : int
buffer : StringBuffer
name : String
traced_actions : ArrayList
application_domain_action_uri : String
instance() : ActionTracer
instance(name : String) : ActionTracer
ActionTracer()
ActionTracer(name : String)
destroy() : void
unsetApplicationDomainAction() : void
setApplicationDomainAction(action_uri : String) : void
toString() : String
wasUnsupported() : boolean
wasException() : boolean
wasSuccess() : boolean
clear() : void
warn(string : String) : void
unsupported(node : Element) : void
exception(node : Element,e : Throwable) : void
exception(e : Throwable) : void
exception(action : String,e : Throwable) : void
traceAction(node : Element,answer : String) : void
traceAction(node : Element,answer : String,props : Properties) : void
traceAction(node : Element,answer : Element,props : Properties) : void
traceAction(action_uri : String,answer : String,props : Properties) : void
traceAction(action_uri : String,answer : Element,props : Properties) : void
tracedActions() : Iterator
Xslt
log : Logger
cache : HashMap
Xslt()
tryCache(xslt_id : String,xslt : String) : Templates
transform(xml : String,xslt : String,xslt_id : String) : String
ActionExecuter
action_exec : Hashtable
log : Logger
execute(uri : String,actt : ActionTracer,server : ModelServer,node : Element) : Object
executeNode(actt : ActionTracer,server : ModelServer,node : Element) : void
getExecuterFor(action_uri : String) : AbstractAction
init() : void
addExec(exec : AbstractAction) : void
AcaMappingProcessor
shared_instance : AcaMappingProcessor
log : Logger
exceptions : Vector
AcaMappingProcessor()
getInstance() : AcaMappingProcessor
countOfCoughtExceptions() : int
getExceptions() : Iterator
transformAction(action : Element) : Element
applyMapping(node : Element) : String
getAcaMappingsFor(action_ns : String,action_name : String) : Iterator
addDefaultModuleImports(the_query : StringBuilder) : void
executeXQueryMapping(xml : String,query : String) : String
executeXSLTMapping(xml : String) : String
executeXSLTMapping(xml : String,xslt_mapping : String,xslt_id : String) : String
ApplicationDomainAction
URI : String
aca_processor : AcaMappingProcessor
ApplicationDomainAction()
getURI() : String
isAction(namespace : String,name : String) : boolean
transformAction(node : Element) : Element
execute(action_tracer : ActionTracer,server : ModelServer,e : Element) : String
abstract MappedDomainActionRegistry
private static Logger log = Logger.getLogger(Action.class)
private static Hashtable action_uri_map
private static Hashtable aca_mapping_name_map
private static Hashtable aca_mapping_id_map
private static ArrayList descs
private static Date description_timestamp
static public boolean isMappedApplicationAction(String namespace, String name)
static public boolean isMappedApplicationAction(String uri)
static public List getMappingDescriptionsFor(String namespace, String name)
static public List getMappingDescriptionsFor(String uri)
static public List getMappingDescriptions()
static private void checkForUpdates()
static private void initAcaMappingDescriptions()
static public String getActionUriOfMapping(String mapping_name)
static public String getActionUriOfMapping(long id)
XQuery
shared_instance : XQuery
log : Logger
dataSource : XQDataSource
conn : XQConnection
XQuery()
getSharedInstance() : XQuery
initDataSource() : XQDataSource
initConnection() : XQConnection
isOpenConnection() : boolean
closeConnection() : void
executeOnDocument(doc : String,query : String) : XQResultSequence
executeOnDocument(node : Node,query : String) : XQResultSequence
executeOnDocument(doc : File,query : String) : XQResultSequence
executeOnSimpleExpression(string : String,query : String) : XQResultSequence
createItemFromString(string : String) : XQItem
createItemFromNode(node : Node) : XQItem
createDocumentFromString(doc : String) : XQItem
createDocumentFromFile(doc : File) : XQItem
execute(xml : XQItem,query : String) : XQResultSequence
finalize() : void
tryParseQuery(query : String) : void
AcaMappingRegistry
log : Logger
aca_mappings : List
timestamp_mappings : Date
aca_librarys : List
timestamp_librarys : Date
shared_instance : AcaMappingRegistry
AcaMappingRegistry()
getSharedInstance() : AcaMappingRegistry
getMappings() : List
mappings() : List
modules() : List
iterator() : Iterator
iterator(action_ns : String,action_name : String) : Iterator
checkForUpdates_modules() : void
checkForUpdates_mappings() : void
initAcaModules() : List
initAcaRules() : List

Figure 5.2.1: ACA Core Architecture

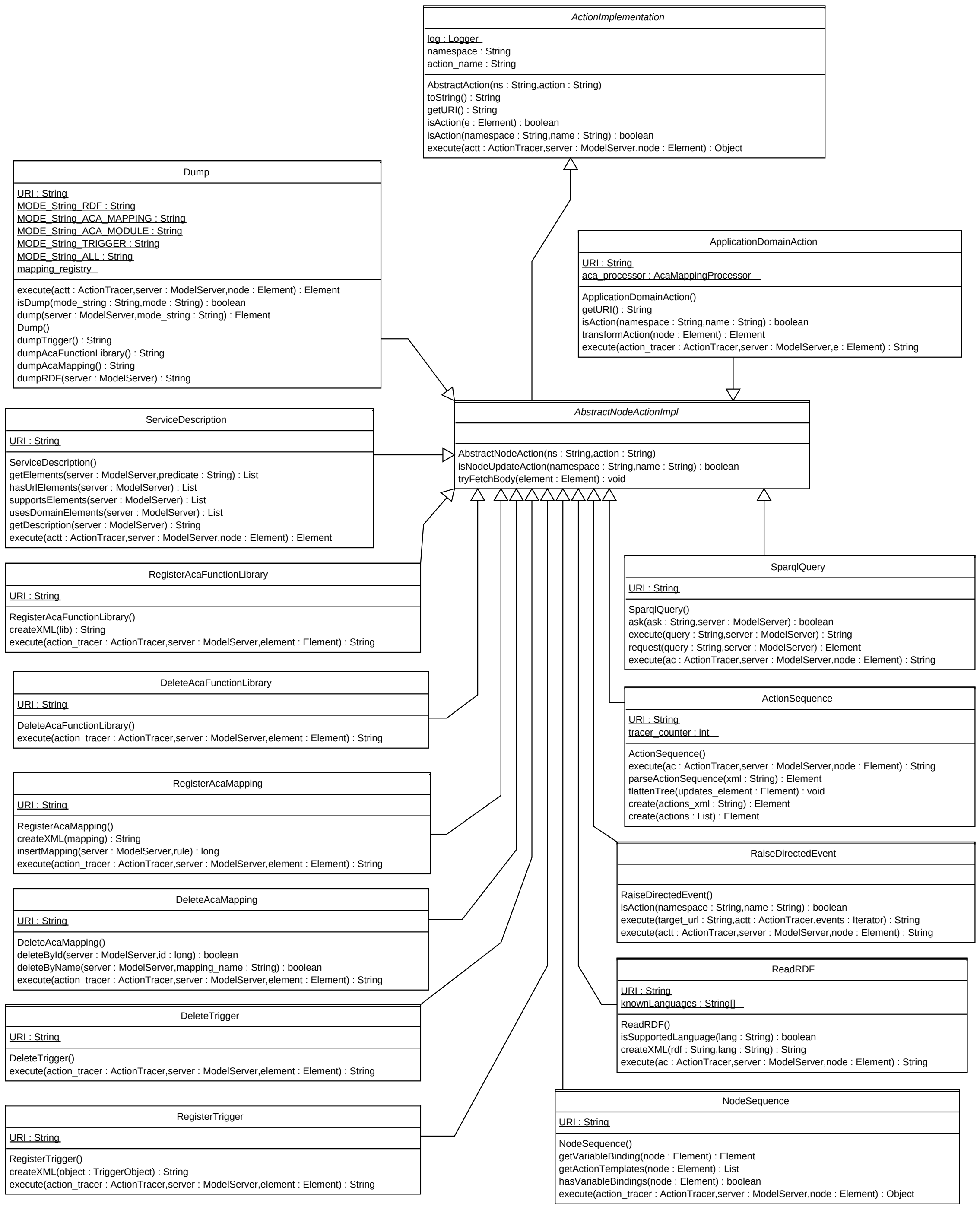

ActionImplementation
log : Logger
namespace : String
action_name : String
AbstractAction(ns : String,action : String)
toString() : String
getURI() : String
isAction(e : Element) : boolean
isAction(namespace : String,name : String) : boolean
execute(actt : ActionTracer,server : ModelServer,node : Element) : Object
Dump
URI : String
MODE_String_RDF : String
MODE_String_ACA_MAPPING : String
MODE_String_ACA_MODULE : String
MODE_String_TRIGGER : String
MODE_String_ALL : String
mapping_registry
execute(actt : ActionTracer,server : ModelServer,node : Element) : Element
isDump(mode_string : String,mode : String) : boolean
dump(server : ModelServer,mode_string : String) : Element
Dump()
dumpTrigger() : String
dumpAcaFunctionLibrary() : String
dumpAcaMapping() : String
dumpRDF(server : ModelServer) : String
ApplicationDomainAction
URI : String
aca_processor : AcaMappingProcessor
ApplicationDomainAction()
getURI() : String
isAction(namespace : String,name : String) : boolean
transformAction(node : Element) : Element
execute(action_tracer : ActionTracer,server : ModelServer,e : Element) : String
AbstractNodeActionImpl
AbstractNodeAction(ns : String,action : String)
isNodeUpdateAction(namespace : String,name : String) : boolean
tryFetchBody(element : Element) : void
ServiceDescription
URI : String
ServiceDescription()
getElements(server : ModelServer,predicate : String) : List
hasUrlElements(server : ModelServer) : List
supportsElements(server : ModelServer) : List
usesDomainElements(server : ModelServer) : List
getDescription(server : ModelServer) : String
execute(actt : ActionTracer,server : ModelServer,node : Element) : Element
RegisterAcaFunctionLibrary
URI : String
RegisterAcaFunctionLibrary()
createXML(lib) : String
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
SparqlQuery
URI : String
SparqlQuery()
ask(ask : String,server : ModelServer) : boolean
execute(query : String,server : ModelServer) : String
request(query : String,server : ModelServer) : Element
execute(ac : ActionTracer,server : ModelServer,node : Element) : String
DeleteAcaFunctionLibrary
URI : String
DeleteAcaFunctionLibrary()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
ActionSequence
URI : String
tracer_counter : int
ActionSequence()
execute(ac : ActionTracer,server : ModelServer,node : Element) : String
parseActionSequence(xml : String) : Element
flattenTree(updates_element : Element) : void
create(actions_xml : String) : Element
create(actions : List) : Element
RegisterAcaMapping
URI : String
RegisterAcaMapping()
createXML(mapping) : String
insertMapping(server : ModelServer,rule) : long
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
RaiseDirectedEvent
RaiseDirectedEvent()
isAction(namespace : String,name : String) : boolean
execute(target_url : String,actt : ActionTracer,events : Iterator) : String
execute(actt : ActionTracer,server : ModelServer,node : Element) : String
DeleteAcaMapping
URI : String
DeleteAcaMapping()
deleteById(server : ModelServer,id : long) : boolean
deleteByName(server : ModelServer,mapping_name : String) : boolean
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
ReadRDF
URI : String
knownLanguages : String[]
ReadRDF()
isSupportedLanguage(lang : String) : boolean
createXML(rdf : String,lang : String) : String
execute(ac : ActionTracer,server : ModelServer,node : Element) : String
DeleteTrigger
URI : String
DeleteTrigger()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
NodeSequence
URI : String
NodeSequence()
getVariableBinding(node : Element) : Element
getActionTemplates(node : Element) : List
hasVariableBindings(node : Element) : boolean
execute(action_tracer : ActionTracer,server : ModelServer,node : Element) : Object
RegisterTrigger
URI : String
RegisterTrigger()
createXML(object : TriggerObject) : String
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String

Figure 5.2.2: Application Node Actions

5.2.2 Class Structure of the `ActionImplementation` Classes

As mentioned in the previous section, `org.semwebtech.applnode.aca.ActionImplementation` objects are very important for the action execution. The methods `isAction(String ns, String name)` and `execute(ActionTracer, ModelServer, Element)` are of highest importance. The latter was already mentioned. Using `isAction`, it is possible to check if the particular object is intended for a certain action URI.

All actual `ActionImplementation` classes are declared as static member classes of the two abstract classes `AbstractRDFUActionImpl` and `AbstractNodeActionImpl`. These types provide protected functions that are used in their sub classes. As said earlier, there is a set of `ActionImplementation` instances in the `ActionExecuter` containing one instance of each type. So when the RDF update language or the node actions should be extended, it is only necessary to provide a new implementation class in the class structure shown in Figure 5.2.3 and Figure 5.2.2 and of course to add this to the `ActionImplementation` set of the `ActionExecuter`.

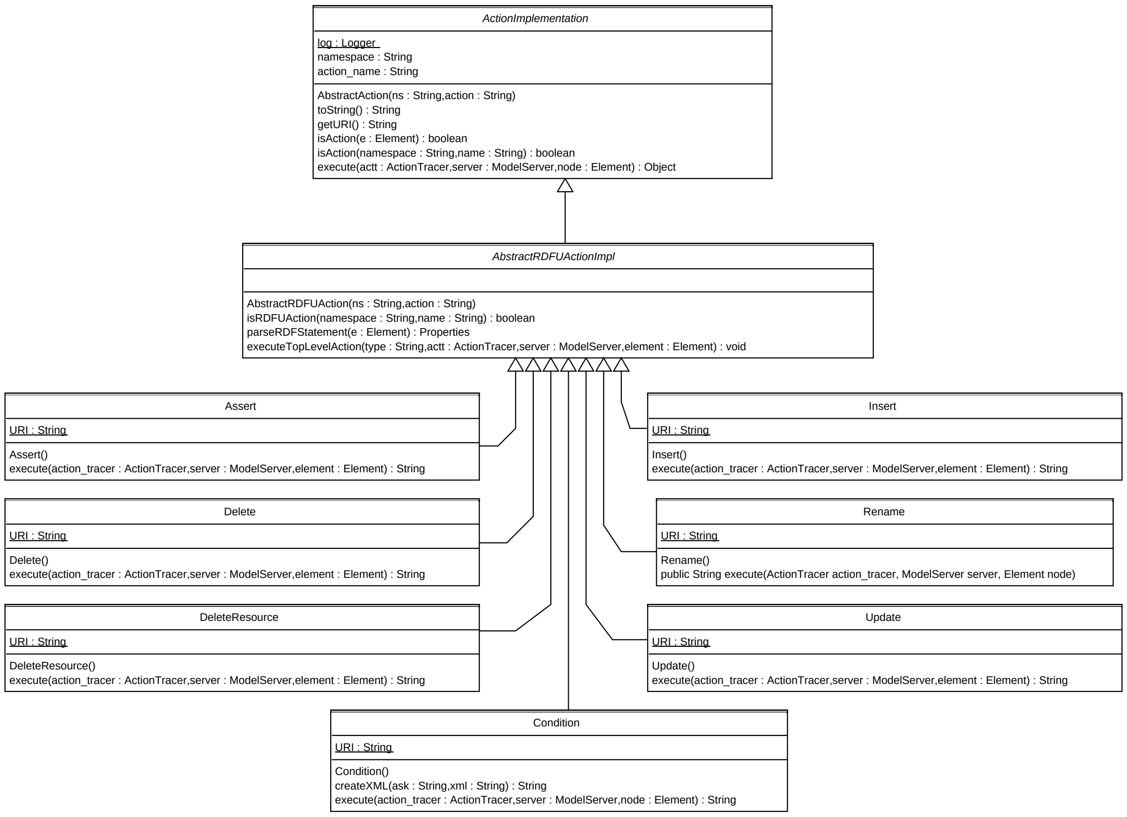

ActionImplementation
log : Logger
namespace : String
action_name : String
AbstractAction(ns : String,action : String)
toString() : String
getURI() : String
isAction(e : Element) : boolean
isAction(namespace : String,name : String) : boolean
execute(actt : ActionTracer,server : ModelServer,node : Element) : Object
AbstractRDFUActionImpl
AbstractRDFUAction(ns : String,action : String)
isRDFUAction(namespace : String,name : String) : boolean
parseRDFStatement(e : Element) : Properties
executeTopLevelAction(type : String,actt : ActionTracer,server : ModelServer,element : Element) : void
Assert
URI : String
Assert()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
Insert
URI : String
Insert()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
Delete
URI : String
Delete()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
Rename
URI : String
Rename()
public String execute(ActionTracer action_tracer, ModelServer server, Element node)
DeleteResource
URI : String
DeleteResource()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
Update
URI : String
Update()
execute(action_tracer : ActionTracer,server : ModelServer,element : Element) : String
Condition
URI : String
Condition()
createXML(ask : String,xml : String) : String
execute(action_tracer : ActionTracer,server : ModelServer,node : Element) : String

Figure 5.2.3: RDF Update Actions

5.2.3 Database Access within the ACA Wrapper

Figure 5.2.4 sketches the database access within the ACA wrapper. Both ACA mappings and XQuery function libraries are stored within a relational database. Just like within the Reactive Web Service, the open database system PostgreSQL is used for this task. The original database schema was extended to store the necessary information needed for the action translation func-

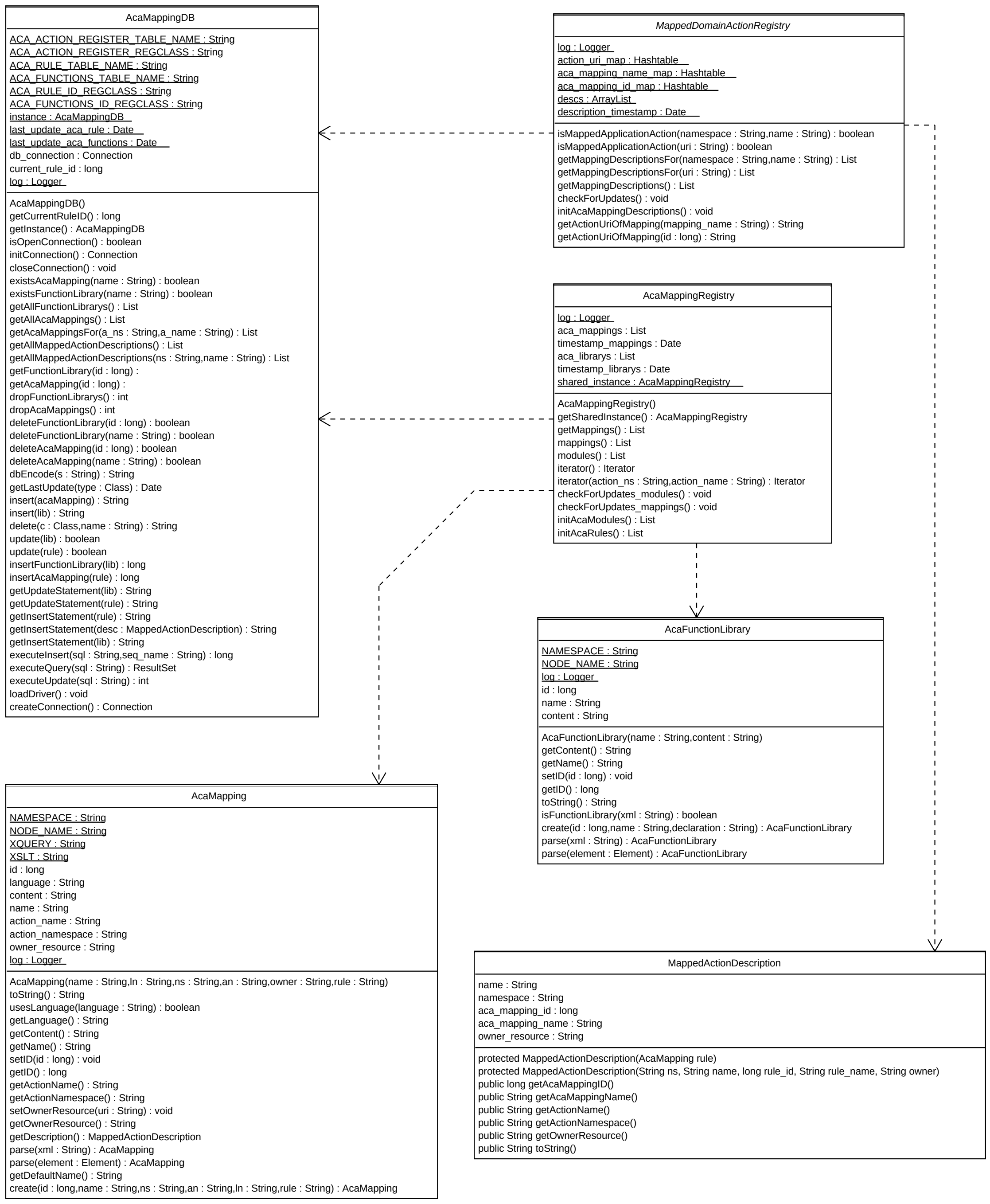

AcaMappingDB
ACA_ACTION_REGISTER_TABLE_NAME : String
ACA_ACTION_REGISTER_REGCLASS : String
ACA_RULE_TABLE_NAME : String
ACA_FUNCTIONS_TABLE_NAME : String
ACA_RULE_ID_REGCLASS : String
ACA_FUNCTIONS_ID_REGCLASS : String
instance : AcaMappingDB
last_update_aca_rule : Date
last_update_aca_functions : Date
db_connection : Connection
current_rule_id : long
log : Logger
AcaMappingDB()
getCurrentRuleID() : long
getInstance() : AcaMappingDB
isOpenConnection() : boolean
initConnection() : Connection
closeConnection() : void
existsAcaMapping(name : String) : boolean
existsFunctionLibrary(name : String) : boolean
getAllFunctionLibrarys() : List
getAllAcaMappings() : List
getAcaMappingsFor(a_ns : String,a_name : String) : List
getAllMappedActionDescriptions() : List
getAllMappedActionDescriptions(ns : String,name : String) : List
getFunctionLibrary(id : long) :
getAcaMapping(id : long) :
dropFunctionLibrarys() : int
dropAcaMappings() : int
deleteFunctionLibrary(id : long) : boolean
deleteFunctionLibrary(name : String) : boolean
deleteAcaMapping(id : long) : boolean
deleteAcaMapping(name : String) : boolean
dbEncode(s : String) : String
getLastUpdate(type : Class) : Date
insert(acaMapping) : String
insert(lib) : String
delete(c : Class,name : String) : String
update(lib) : boolean
update(rule) : boolean
insertFunctionLibrary(lib) : long
insertAcaMapping(rule) : long
getUpdateStatement(lib) : String
getUpdateStatement(rule) : String
getInsertStatement(rule) : String
getInsertStatement(desc : MappedActionDescription) : String
getInsertStatement(lib) : String
executeInsert(sql : String,seq_name : String) : long
executeQuery(sql : String) : ResultSet
executeUpdate(sql : String) : int
loadDriver() : void
createConnection() : Connection
MappedDomainActionRegistry
log : Logger
action_uri_map : Hashtable
aca_mapping_name_map : Hashtable
aca_mapping_id_map : Hashtable
descs : ArrayList
description_timestamp : Date
isMappedApplicationAction(namespace : String,name : String) : boolean
isMappedApplicationAction(uri : String) : boolean
getMappingDescriptionsFor(namespace : String,name : String) : List
getMappingDescriptionsFor(uri : String) : List
getMappingDescriptions() : List
checkForUpdates() : void
initAcaMappingDescriptions() : void
getActionUriOfMapping(mapping_name : String) : String
getActionUriOfMapping(id : long) : String
AcaMappingRegistry
log : Logger
aca_mappings : List
timestamp_mappings : Date
aca_librarys : List
timestamp_librarys : Date
shared_instance : AcaMappingRegistry
AcaMappingRegistry()
getSharedInstance() : AcaMappingRegistry
getMappings() : List
mappings() : List
modules() : List
iterator() : Iterator
iterator(action_ns : String,action_name : String) : Iterator
checkForUpdates_modules() : void
checkForUpdates_mappings() : void
initAcaModules() : List
initAcaRules() : List
AcaFunctionLibrary
NAMESPACE : String
NODE_NAME : String
log : Logger
id : long
name : String
content : String
AcaFunctionLibrary(name : String,content : String)
getContent() : String
getName() : String
setID(id : long) : void
getID() : long
toString() : String
isFunctionLibrary(xml : String) : boolean
create(id : long,name : String,declaration : String) : AcaFunctionLibrary
parse(xml : String) : AcaFunctionLibrary
parse(element : Element) : AcaFunctionLibrary
AcaMapping
NAMESPACE : String
NODE_NAME : String
XQUERY : String
XSLT : String
id : long
language : String
content : String
name : String
action_name : String
action_namespace : String
owner_resource : String
log : Logger
AcaMapping(name : String,ln : String,ns : String,an : String,owner : String,rule : String)
toString() : String
usesLanguage(language : String) : boolean
getLanguage() : String
getContent() : String
getName() : String
setID(id : long) : void
getID() : long
getActionName() : String
getActionNamespace() : String
setOwnerResource(uri : String) : void
getOwnerResource() : String
getDescription() : MappedActionDescription
parse(xml : String) : AcaMapping
parse(element : Element) : AcaMapping
getDefaultName() : String
create(id : long,name : String,ns : String,an : String,ln : String,rule : String) : AcaMapping
MappedActionDescription
name : String
namespace : String
aca_mapping_id : long
aca_mapping_name : String
owner_resource : String
protected MappedActionDescription(AcaMapping rule)
protected MappedActionDescription(String ns, String name, long rule_id, String rule_name, String owner)
public long getAcaMappingID()
public String getAcaMappingName()
public String getActionName()
public String getActionNamespace()
public String getOwnerResource()
public String toString()

Figure 5.2.4: ACA database classes

tionality. Classes that deal with the database storage are in the package

`org.semwebtech.applnode.aca.database`.

There are three data classes that represent the objects stored in the database. These are `AcaMapping`, `AcaFunctionLibrary` and `MappedActionDescription`. The latter is responsible for the storage of relations between ACA mappings and application domain actions. That information is not stored directly with the mapping content but externally in a separate relation with a foreign key to the actual mapping data.

The database access is provided centrally by the class `AcaMappingDB`. It encapsulates all information about relation names and schema and therefore simplifies the handling of possible modifications of the database schema. Two classes `MappedDomainActionRegistry` and `AcaMappingRegistry` provide cached access to the database content. It would be inefficient to query the database by calling the methods of `AcaMappingDB` every time a mapping or similar information is used. Thus the two registry classes provide a cached database state in the memory that automatically refreshes when a database update was performed.
The relations used for storage of the ACA functionality are defined as follows:

```
-- +++++++++++++++++++++++++++++++++++++++++++
-- relation that contains the ACA Mappings
--
CREATE TABLE acarule
(
  -- primary key --
  id serial NOT NULL,

  -- unique name --
  name varchar(256) NOT NULL,

  -- rule content, language (xquery or xslt) --
  rule text NOT NULL,
  language varchar(32) NOT NULL,

  CONSTRAINT acarule_pkey PRIMARY KEY (id),
  CONSTRAINT acarule_unique_name UNIQUE (name)
) ;

-- ++++++++++++++++++++++++++++++++++++++++++++
-- relation that contains the XQuery modules
--
CREATE TABLE acafunctionlibrary
(
  -- primary key --
  id serial,
  -- unique name --
  name varchar(256) NOT NULL,

  -- mudule declaration --
  declaration text NOT NULL,

  CONSTRAINT acafunction_pkey PRIMARY KEY (id),
  CONSTRAINT acafunction_unique_name UNIQUE (name)
) ;

-- ++++++++++++++++++++++++++++++++++++++++++++
-- relation containing the information that maps
-- ACA Rule to a specific Application Domain Action
--
CREATE TABLE acaactionregister
(
  -- primary key --
  id serial NOT NULL,

  -- Application Domain Action (namespace, name) --
  namespace varchar(256) NOT NULL,
  name varchar(256) NOT NULL,

  -- corresponding ACA Mapping (id, name, owner) --
  aca_id int4 NOT NULL,
  aca_name varchar(256) NOT NULL,
  owner_resource varchar(256),

  CONSTRAINT acaactionregister_pkey PRIMARY KEY (id),
  CONSTRAINT acaregister_acarule_name FOREIGN KEY (aca_name)
      REFERENCES acarule (name) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT acaregister_acarule_id FOREIGN KEY (aca_id)
      REFERENCES acarule (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE CASCADE
) ;
```

[Filename:config/db-script-aca.sql]

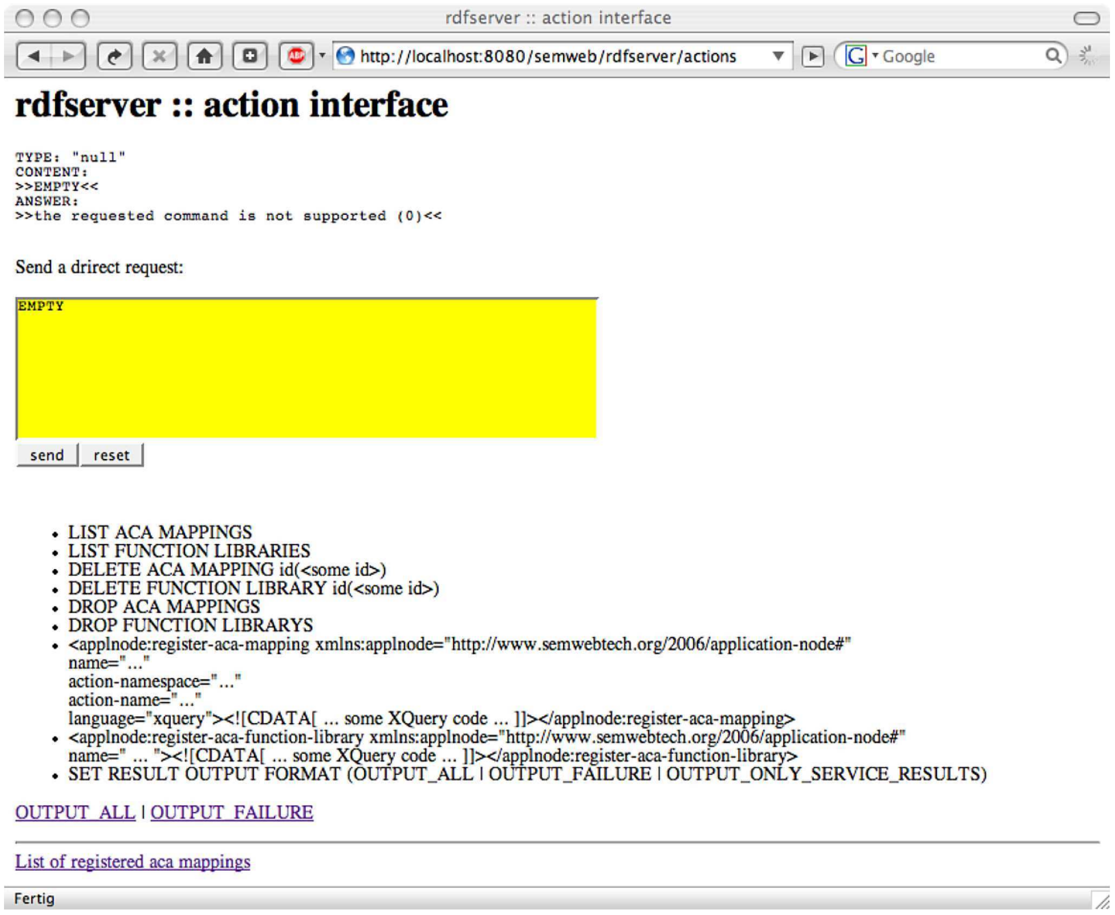
5.3 Action Interface Servlets

The Action interface is mainly provided by a single Servlet that is intended to receive XML action messages. Altogether, there are three Servlets that build the interface. These are (with their URLs as currently configured in the prototype):

- `ActionServlet` - Main Servlet of the ACA wrapper (relative URL: /semweb/rdfserver/actions and /semweb/rdfserver/services).
- `ModuleAccessorServlet` - Servlet used to locally access and modify stored XQuery modules (relatice URL: /semweb/aca/modules).
- `MappingAccessorServlet` - Servlet used to locally access and modify stored ACA mappings (relatice URL: /semweb/aca/mappings).

Additionally, there are two Servlets that build the original Web Service. These are:

- `RDFTriggerServlet` - Main Servlet of the original RDF Web Service.
(relative URL: /semweb/rdfserver)
- `TriggerServlet` - Dummy service that is used as Event Broker. It opens received messages in a GUI window. (relatice URL: /semweb/triggerserver).


rdfserver :: action interface
http://localhost:8080/semweb/rdfserver/actions
Google

rdfserver :: action interface

```
TYPE: "null"
CONTENT:
>>EMPTY<<
ANSWER:
>>the requested command is not supported (0)<<
```

Send a drirect request:

```
EMPTY
```

- LIST ACA MAPPINGS
- LIST FUNCTION LIBRARIES
- DELETE ACA MAPPING id(<some id>)
- DELETE FUNCTION LIBRARY id(<some id>)
- DROP ACA MAPPINGS
- DROP FUNCTION LIBRARYS
- <applnode:register-aca-mapping xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
name="..."
action-namespace="..."
action-name="..."
language="xquery"><![CDATA[... some XQuery code ...]]></applnode:register-aca-mapping>
- <applnode:register-aca-function-library xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
name=" ... "><![CDATA[... some XQuery code ...]]></applnode:register-aca-function-library>
- SET RESULT OUTPUT FORMAT (OUTPUT_ALL | OUTPUT_FAILURE | OUTPUT_ONLY_SERVICE_RESULTS)

OUTPUT_ALL | OUTPUT_FAILURE

List of registered aca mappings

Fertig

Figure 5.3.1: ActionServlet screenshot

The `ActionServlet` is responsible for the processing of XML mark-upped actions including the administrative node actions, RDF update actions and of course application domain actions. The messages are parsed into the DOM and forwarded to the ACA executing classes as described earlier in Section 5.2.

Additionally to XML messages this Servlet can also process the following plain text commands:

- list aca mappings - Returns a text representaion of all registered ACA mappings.
- list function libraries - Returns a text representaion of all stored XQuery modules
- delete aca mapping [id | name] (<<ID or NAME>>) - Deletes the ACA mapping corresponding to a given primary key value or mapping name.
- delete function library [id | name] (<<ID or NAME>>) - Deletes the XQuery module corresponding to a given primary key value or module name.
- drop aca mappings - Removes all ACA mappings from the database
- drop function libraries - Removes all XQuery modules from the database.
- set output result format (OUTPUT_ALL | OUTPUT_FAILURE | OUTPUT_ONLY_SERVICE_RESULTS) - Changes the answering behavior of the action interface.
  When set to OUTPUT_ONLY_SERVICE_RESULTS the interface only returns variable bindings as answer to a query, its service description and a requested dump file. Using the OUTPUT_FAILURE option, failure messages that occurred are additionally included in the Servlet response. OUTPUT_ALL lets the response reassemble a complete list of the executed operations.

When enabled by set output result format, the result of an executed action is usually an applnoderesult element.

Example 14 *Take for example a Domain Action* uni:add-student *in form of the following message:*

```
<uni:add-student xmlns:uni="http://localhost/test.owl#">
    <uni:uri>stud_peter</uni:uri>
    <uni:name>Peter</uni:name>
</uni:add-student>
```

Assume there is an ACA mapping for that action with the following XQuery definition:

```
import module namespace uni = "http://localhost/test.owl#"
     at  "http://localhost:8080/semweb/aca/modules/uni";

for $stud in //uni:add-student
let $uri := aca:getProperty($stud, "uni:uri")
let $stud_uri := uni:person_uri($uri)
let $name := aca:getProperty($stud, "uni:name")
return
(
     rdfu:insert($stud_uri, rdf:ns("type"), uni:ns("Student")),
     rdfu:insert($stud_uri, rdf:ns("type"), owl:ns("Thing")),
     rdfu:insert($stud_uri, uni:ns("name"), rdfu:stringLiteral($name))
)
```

So when the message above is sent to the ACA interface, it is processed by that mapping to the following RDF update sequence:

```
<applnode:actionSequence
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#">
  <rdfu:insert>
    <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
    <rdf:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:predicate>
    <rdf:object>http://localhost/test.owl#Student</rdf:object>
  </rdfu:insert>
  <rdfu:insert>
     <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
     <rdf:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:predicate>
     <rdf:object>http://www.w3.org/2002/07/owl#Thing</rdf:object>
  </rdfu:insert>
  <rdfu:insert>
     <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
     <rdf:predicate>http://localhost/test.owl#name</rdf:predicate>
     <rdf:object>Peter^^&lt;http://www.w3.org/2001/XMLSchema#string&gt;</rdf:object>
  </rdfu:insert>
</applnode:actionSequence>
```

When the result is enabled by the OUTPUT_ALL option, the following messege is returned as a response:

```
<applnode:result
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
  xmlns:mars="http://www.semwebtech.org/2006/mars#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#">
  <applnode:executed-action
    applnode:action-uri="http://www.semwebtech.org/lang/2006/rdfupdate#insert"
    mars:Action="http://localhost/test.owl#add-student" >
    <rdf:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:predicate>
    <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
    <rdf:object>http://localhost/test.owl#Student</rdf:object>
    <![CDATA[Statement added successfully.]]>
  </applnode:executed-action>
  <applnode:executed-action
    applnode:action-uri="http://www.semwebtech.org/lang/2006/rdfupdate#insert"
    mars:Action="http://localhost/test.owl#add-student" >
    <rdf:predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:predicate>
    <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
    <rdf:object>http://www.w3.org/2002/07/owl#Thing</rdf:object>
    <![CDATA[Statement added successfully.]]>
  </applnode:executed-action>
  <applnode:executed-action
    applnode:action-uri="http://www.semwebtech.org/lang/2006/rdfupdate#insert"
    mars:Action="http://localhost/test.owl#add-student" >
    <rdf:predicate>http://localhost/test.owl#name</rdf:predicate>
    <rdf:subject>http://localhost/test.owl#stud_peter</rdf:subject>
    <rdf:object>Peter^^<http://www.w3.org/2001/XMLSchema#string></rdf:object>
    <![CDATA[Statement added successfully.]]>
  </applnode:executed-action>
</applnode:result>
```

Note that the attribute mars:Action *marks a particular RDF update action to be performed as a result of a mapping of the application domain action* http://localhost/test.owl#add-student".

Failures would be returned as an applnode:failure *element instead of* applnode:executed-action. *If OUTPUT_FAILURE or OUTPUT_ONLY_SERVICE_RESULTS is set as output format when the* uni:add-student *is executed as shown above, nothing would be returned.*

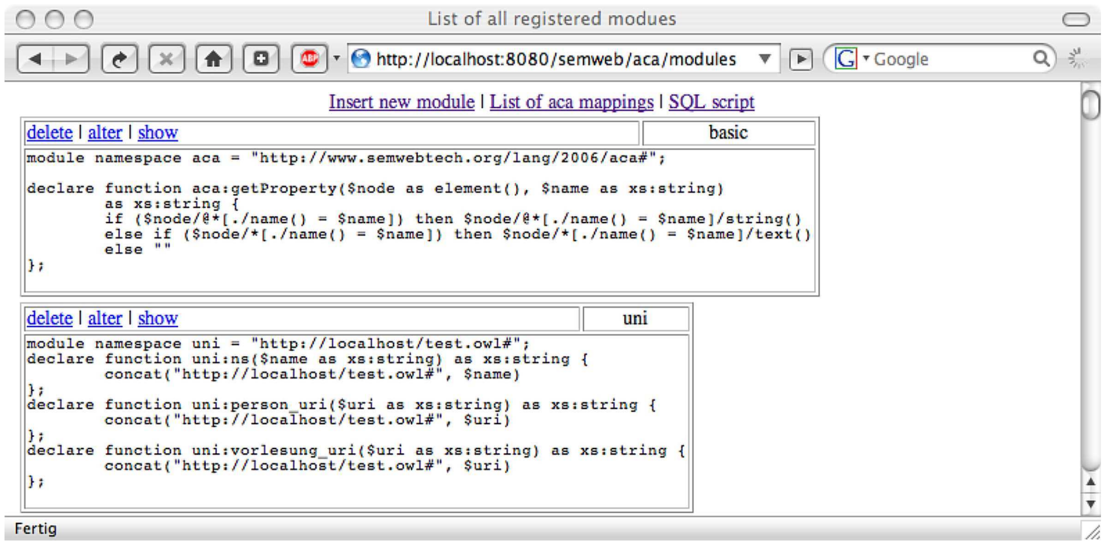
Figure 5.3.1 shows a screenshot of the action interface when loaded in a web browser. The server is not intended to be used with a browser but it is nevertheless possible and useful for administrative purposes. The yellow text input area enables the user to directly send requests to the server. The links at the bottom of the page change the output format of the ACA interface or lead to the ACA mapping Registry. The latter is provided by the two Servlets `MappingAccessorServlet` and `ModuleAccessorServlet` as mentioned above. These servlets support the listing of registered mappings and modules (see Figure 5.3.2) and their modification via the alter link, deletion via the delete link and the insertion of new modules and mappings. Figure 5.3.3 shows the insert form for a new ACA mapping.

Both Servlets enable access to the corresponding content. So, either mappings or modules can be accessed via HTTP. When following the show link on the list pages (Figure 5.3.2) the module or entry is shown in plain text. That is especially useful for the XQuery modules that are imported in ACA mappings. Every module that has been registered is accessible under the relative URL /semweb/aca/modules/<<MODULE NAME>>.

5.4 Extension of the RDF Web Server Client

Apart from the interface provided by the Servlets described in the latter section, there is a client GUI that enables the administration of the Web Service. That client was created with the original Web Service. It has been extended to be additionally used to administer the ACA transformation mechanism. That has been done by adding a second main window to the client that uses the old infrastructure and just extends the functionality. That new dialog is shown in Figure 5.4.1. There is a button Trigger client that opens the old dialog.

The new dialog provides the ability to register and delete ACA mappings and XQuery modules via HTTP messages. There is also the possibility to send XML actions to the server. The top


List of all registered modues
http://localhost:8080/semweb/aca/modules
Google
Insert new module | List of aca mappings | SQL script
delete | alter | show
basic
delete | alter | show
uni
Fertig

```
module namespace aca = "http://www.semwebtech.org/lang/2006/aca#";

declare function aca:getProperty($node as element(), $name as xs:string)
        as xs:string {
        if ($node/@*[./name() = $name]) then $node/@*[./name() = $name]/string()
        else if ($node/*[./name() = $name]) then $node/*[./name() = $name]/text()
        else ""
};
```

```
module namespace uni = "http://localhost/test.owl#";
declare function uni:ns($name as xs:string) as xs:string {
        concat("http://localhost/test.owl#", $name)
};
declare function uni:person_uri($uri as xs:string) as xs:string {
        concat("http://localhost/test.owl#", $uri)
};
declare function uni:vorlesung_uri($uri as xs:string) as xs:string {
        concat("http://localhost/test.owl#", $uri)
};
```

Figure 5.3.2: ModuleAccessorServlet: List of registered modules

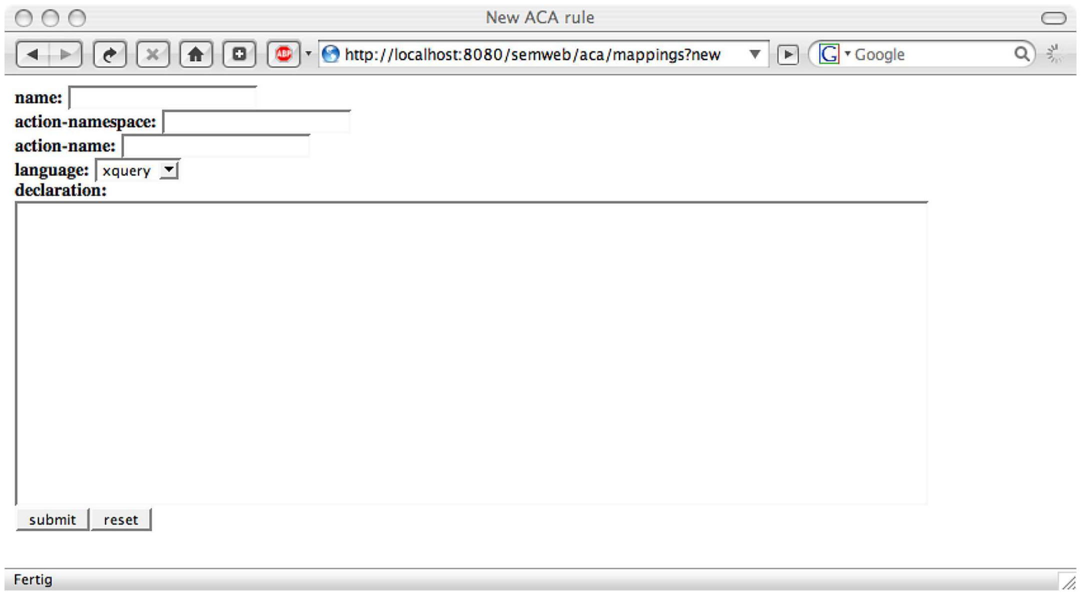
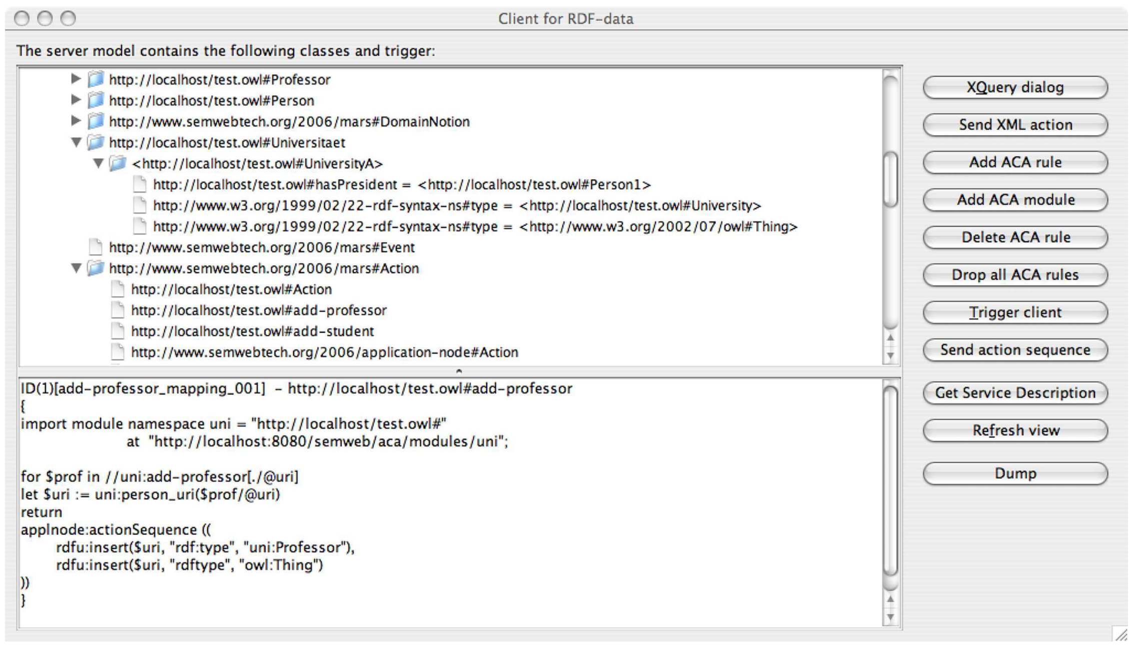


Figure 5.3.3: Form to insert a new ACA Mapping

button on the right hand side of the client window opens the XQuery dialog that is shown in Figure 5.4.2. That dialog may be used to test XQuery expressions locally before including them in mappings or function library modules. The dialog has two text input areas. The one at the bottom is expected to contain a XQuery query that is evaluated using an XML document given in the upper text area as context node. Therefore it is possible to test the transformation of XML marked-up actions with a given XQuery expression. The result of that transformation is shown in a new window. That dialog creates the same as transformation result than the actual ACA wrapper infrastructure. Thus the same default XQuery modules are imported automatically.



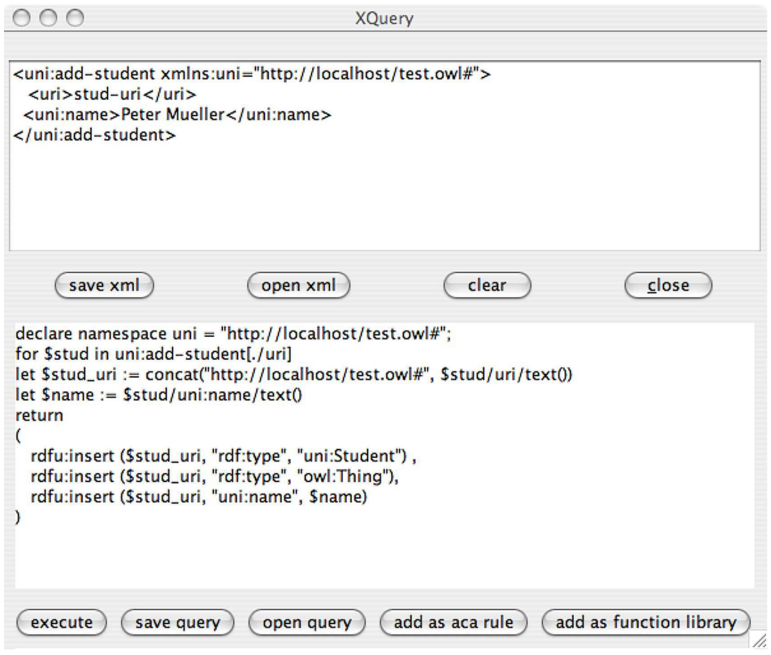
The server model contains the following classes and trigger:

```
http://localhost/test.owl#Professor
http://localhost/test.owl#Person
http://www.semwebtech.org/2006/mars#DomainNotion
http://localhost/test.owl#Universitaet
    <http://localhost/test.owl#UniversityA>
        http://localhost/test.owl#hasPresident = <http://localhost/test.owl#Person1>
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type = <http://localhost/test.owl#University>
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type = <http://www.w3.org/2002/07/owl#Thing>
http://www.semwebtech.org/2006/mars#Event
http://www.semwebtech.org/2006/mars#Action
    http://localhost/test.owl#Action
    http://localhost/test.owl#add-professor
    http://localhost/test.owl#add-student
    http://www.semwebtech.org/2006/application-node#Action
```

```
ID(1)[add-professor_mapping_001]  - http://localhost/test.owl#add-professor
{
import module namespace uni = "http://localhost/test.owl#"
               at  "http://localhost:8080/semweb/aca/modules/uni";

for $prof in //uni:add-professor[./@uri]
let $uri := uni:person_uri($prof/@uri)
return
applnode:actionSequence ((
      rdfu:insert($uri, "rdf:type", "uni:Professor"),
      rdfu:insert($uri, "rdftype", "owl:Thing")
))
}
```

Figure 5.4.1: Client Main Window Screenshot



```
<uni:add-student xmlns:uni="http://localhost/test.owl#">
  <uri>stud-uri</uri>
 <uni:name>Peter Mueller</uni:name>
</uni:add-student>
```

```
declare namespace uni = "http://localhost/test.owl#";
for $stud in uni:add-student[./uri]
let $stud_uri := concat("http://localhost/test.owl#", $stud/uri/text())
let $name := $stud/uni:name/text()
return
(
   rdfu:insert ($stud_uri, "rdf:type", "uni:Student") ,
   rdfu:insert ($stud_uri, "rdf:type", "owl:Thing"),
   rdfu:insert ($stud_uri, "uni:name", $name)
)
```

Figure 5.4.2: XQuery Dialog Screenshot

5.5 Examples and Testing

The local implementation of the action interface includes a mechanism to execute predefined sequences of actions. It is provided by the protected local action applnode:actionSequence as described in Section 4.4. This action is used internally when an application domain action is executed because usually such a process involves the execution of several data model level update instructions.

But additionally, the infrastructure provided by these local actions can be used for the specification of test cases. That is because practically, a test case is nothing less than the execution of a stack of instructions in a certain order that intend a particular result.

Using an applnode:actionSequence, it is possible to define a test scenario for the action interface by describing a list of actions. These will usually include the insertion of certain statements in the knowledge base and the registration of ACA mappings followed by certain application domain action that trigger the mappings and are transformed to derived data level actions. This process can be monitored either by the result returned from the action interface or by adding queries to the test case that output information about the success of the mapping by querying data that correlates with the mapping that should be tested.

There are some predefined test cases that can be found under /semweb/examples/ when using the original configuration of the prototype. There also are ACA mappings that can be found under the relative URL /semweb/test/resources/.
An example for such a test case is given by the following XML document:

```
<node:actionSequence
  xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:aca="http://www.semwebtech.org/lang/2006/aca#"
  xmlns:node="http://www.semwebtech.org/2006/application-node#">

  <!-- setup -->
  <rdfu:assert>
    <rdf:subject rdf:resource="http://localhost/test.owl#add-student" />
    <rdf:predicate rdf:resource="http://www.w3.org/2000/01/rdf-schema#subClassOf" />
    <rdf:object rdf:resource="http://www.semwebtech.org/2006/mars#Action" />
  </rdfu:assert>
  <rdfu:assert
    <rdf:subject rdf:resource="http://localhost/test.owl#add-vorlesung" />
    <rdf:predicate rdf:resource="http://www.w3.org/2000/01/rdf-schema#subClassOf" />
    <rdf:object rdf:resource="http://www.semwebtech.org/2006/mars#Action" />
  </rdfu:assert>
  <rdfu:assert
    rdf:subject rdf:resource=http://localhost/test.owl#hoert-vorlesung" />
    rdf:predicate rdf:resource="http://www.w3.org/2000/01/rdf-schema#subClassOf" />
    rdf:object rdf:resource="http://www.semwebtech.org/2006/mars#Action" />
  </rdfu:assert>
```

```
<node:register-aca-mapping
  node:name="mapping_xquery_newStudent_1"
  node:action-namespace="http://localhost/test.owl#"
  node:action-name="add-student"
  node:language="xquery"
  node:url="http://localhost:8080/semweb/test/resources/mapping_add-student_1.xq" />
<node:register-aca-mapping
  node:name="mapping_xquery_newVorlesung_1"
  node:action-namespace="http://localhost/test.owl#"
  node:action-name="add-vorlesung"
  node:language="xquery"
  node:url="http://localhost:8080/semweb/test/resources/mapping_add-vorlesung_1.xq" />
<node:register-aca-mapping
  node:name="mapping_xquery_hoertVorlesung_1"
  node:action-namespace="http://localhost/test.owl#"
  node:action-name="hoert-vorlesung"
  node:language="xquery"
  node:url="http://localhost:8080/semweb/test/resources/mapping_hoert-vorlesung_1.xq" />

<!-- application domain actions -->
<uni:add-student xmlns:uni="http://localhost/test.owl#">
  <uni:uri>stud_peter</uni:uri>
  <uni:name>Peter</uni:name>
</uni:add-student>
<uni:add-vorlesung xmlns:uni="http://localhost/test.owl#">
  <uni:uri>vorl_semweb</uni:uri>
  <uni:name>Semantic Web</uni:name>
</uni:add-vorlesung>
<uni:hoert-vorlesung xmlns:uni="http://localhost/test.owl#">
  <uni:stud>http://localhost/test.owl#stud_peter</uni:stud>
  <uni:vorl>http://localhost/test.owl#vorl_semweb</uni:vorl>
</uni:hoert-vorlesung>

<node:query><![CDATA[
        select ?stud ?x ?y where { ?stud rdf:type uni:Person . ?stud ?x ?y }
]]></node:query>

<!-- clean up -->
<node:delete-aca-mapping node:name="mapping_xquery_newStudent_1" />
<node:delete-aca-mapping node:name="mapping_xquery_newVorlesung_1" />
<node:delete-aca-mapping node:name="mapping_xquery_hoertVorlesung_1" />
<rdfu:delete-resource rdf:resource="http://localhost/test.owl#stud_peter" />
<rdfu:delete-resource rdf:resource="http://localhost/test.owl#vorl_semweb" />
<rdfu:delete-resource rdf:resource="http://localhost/test.owl#add-student" />
<rdfu:delete-resource rdf:resource="http://localhost/test.owl#add-vorlesung" />
```

```
  <rdfu:delete-resource rdf:resource="http://localhost/test.owl#hoert-vorlesung" />

</node:actionSequence>
```

[Filename:/semweb/examples/TestCase_xquery_1.xml]

At first, this test ensures that three application domain action definitions (uni:add-student, uni:add-vorlesung and uni:hoert-vorlesung) are known to the local knowledge base. After that, three ACA mappings are registered that are followed by three corresponding Domain Actions and a query. Finally, the modified resources are deleted and the mappings are de-registered.

A second way to test the action interface is provided by JUnit tests that directly use the ACA wrapper by calling the methods of the `Action` class. These JUnit test cases can be found in the Java package `org.semwebtech.applnode.test`. There is the convenience class

`org.semwebtech.applnode.test.AbstractActionTestCase`

that should be extended. It provides methods like

`toDOM(String xml)`,

`executeAction(Element)`,

`insertXQueryMapping(..)`,

`insertWorldAction(String uri)` or

`sparql(String query)` etc.

An example JUnit test that is defined in the `TestAction4` class has the following Java code:

```
public void testMapping1() {
    String mapping = resourceToString("mapping_add-student_1.xq");
    String name = "mapping_add-student_1";
    String action_ns = uni_ns;
    String action_name = "add-student";
    try {
        insertWorldAction(unins(action_name));
        insertXqueryMapping(name, action_ns, action_name, mapping);
        executeAction(addStudent("stud_peter", "Peter"));
        assertEquals("Peter", getObject(unins("stud_peter"), unins("name")));
    } finally {
        deleteWorldAction(unins(action_name));
        deleteMapping(name);
        deleteResource(unins("stud_peter"));
    }
}
```

The method `resourceToString(String filename)` enables to access the same resources that can be found under /semweb/test/resources/. Using `getObject(String subject_uri, String predicate_uri)` a query of the local model can be executed that will return the corresponding object.

Most methods from `AbstractActionTestCase` implicitly test their success and may throw an `AssertionFailedError` if they fail.

5.6 Installation of the ACA Reactive RDF Web Service

The prototype implementation of the ACA Reactive Web Service uses the Java Servlet technology as already mentioned. Therefore, it has to be installed in a Java Container to be used. For such an deployment, usually a Java WAR file is used that contains all necessary classes and configuration files. This WAR file along a JAR file containing the GUI client can be automatically created using the `ant` tool [2]. The provided `ant build.xml` file contains build targets for simple compiling, creation of the archives and additionally the automatic deployment of the WAR file. Of course this requires that the `build.xml` file is configured accordingly.

The Web Service is configured by a configuration property file that will be described in the next section. This file is indented to be found in the WEB-INF directory of the WAR file. There will also be search for the log4j [9] configuration file `log4j.properties`, two SQL script files named `db_script_aca.sql` and `db_script_trigger.sql` and an XML file named `nodeInit.xml` containing node actions that are executed upon the initialization of the `ActionServlet`. The configuration file has to contain a valid database URL and user account information for the application to be able to open a database connection to the underlying relational database where the RDF data, the triggers and the ACA mappings are stored. In fact, the only precondition that is required for a successful installation of the Web Service is that these database informations are valid. All necessary tables are created automatically using the mentioned SQL scripts. The `nodeInit.xml` file defines initial ACA mapping, Trigger and XQuery module registrations.

5.7 Configuration

Basically, the Web Service is configured by a properties file named `configuration.properties`. Before installing the Web Service it is therefore necessary to update this file. It contains the following properties:

- org.semwebtech.service.uri - Service URI of the particular Domain Service provided by this installation. This URI is used to declare mars:supports or mars:uses-domain statements that are included in the Service Description of the Domain Service.
- dburl, dbuser, dbpasswd - Information used for the database connection. There are three of each of these properties because it is intended that the triggers, the mappings and the model data may be stored in separate databases or be accessed by different users. See the example configuration file that can be found under B.1 to see how these properties are used in detail.
- org.semwebtech.applnode.servlet.ModelServer.model.name - URI for the knowledge base model that is used for storage in the database.

- org.semwebtech.applnode.servlet.ModelServer.reasonerurl - URL of the DIG reasoner that will be used.
- org.semwebtech.events.broker.url - URL of the Event Broker
- org.semwebtech.applnode.servlet.rdfserver.url - URL of the main Web Service. That is the URL the `RDFTriggerServlet` responds to. It is nedded because the two Serlvets `RDFTriggerServlet` and `ActionServlet` call each other via HTTP.
- org.semwebtech.applnode.servlet.rdfserver.actions.url - URL the `ActionServlet` responds to.
- org.semwebtech.applnode.servlet.ModuleAccessorServlet.url - URL of the `ModuleAccessorServlet`. That URL is needed because the XQuery modules that are imported automatically upon mapping execution are loaded from this URL.

Additionally, the following namespace properties have to set accordingly:

- org.semwebtech.lang.logvars.namespace
- org.semwebtech.lang.aca.namespace
- org.semwebtech.lang.rdfu.namespace
- org.semwebtech.applnode.namespace
- org.semwebtech.mars.namespace
- org.w3.rdf.namespace
- org.w3.rdfs.namespace
- org.w3.owl.namespace

Chapter 6

Conclusion

With the RDF Web Service that provides trigger functionality for the raising of events and the ACA wrapper for the mapping of ontology actions to the local data model, it is possible to create a service that is fully integrated into the MARS Framework. Therefore, for the first time, a test scenario can be implemented that uses real Domain Services for that create domain events and consume domain actions. So far, the different services like Domain Broker, ECA Engine and also the RDF Web Service were only tested using stubs and dummy-implementations for the remote framework services.

The Domain Services are important for such a test case scenario as this is where the application logic is implemented. The other services provide the infrastructure for the event-driven communication and the possibility to specify and execute active rules, but without Domain Services a "living" example network using the technology of the framework can't be implemented.

Therefore, the next step in the framework development will be the definition of "real" Domain Services that emit events and react upon action requests. At first, these services are restricted to belong to only one application domain. The reason for this limitation lies in the current implementation. The prototypical Web Service only supports one registered Event Broker. Thus, the assignment of raised events to different Event Brokers is not possible. Though it would be easy to provide such functionality, this is not necessary at the moment. It is more important to join the different components of the framework together for a first time.

The main task will be to bring the communication interfaces of the different components into line. As the development of the framework has progressed certain interfaces have changed. The first step is to employ all components in a simple scenario that breaths life into the framework. The complexity is of minor importance. Afterwards, the test scenario can be extended. While doing this the components will presumably have to be adapted to fulfill new requirements.

Appendix A

Abbreviations

ECA:	Event-Condition-Action
ECE:	Event-Condition-Event derivation rule
ACA:	Action-Condition-Action implementation rule
DSR:	Domain Service Registry
LSR:	Languages and Services Registry
XML:	Extensible Markup Language
XSLT:	Extensible Stylesheet Language (XSL) Transformations
WAR:	Web Archive
JAR:	Java Archive

Appendix B

ACA RDF Web Service Configuration Files

B.1 Configuration Properties File

```
# SERVICE PROPERTIES ============================================
# URI of service
org.semwebtech.service.uri = "http://localhost:8080/semweb/uni-service"

# DATABASE PROPERTIES ===========================================
# --- JENA/RDF --------------------------------------------------
# URL of RDF database server
org.semwebtech.applnode.servlet.ModelServer.dburl = "localhost/test1"
# RDF database user id
org.semwebtech.applnode.servlet.ModelServer.dbuser = "schenk"
# RDF database password
org.semwebtech.applnode.servlet.ModelServer.dbpasswd = "irgendeinpassowrt"

# --- TRIGGER ---------------------------------------------------
# URL of the database containing the trigger
org.semwebtech.applnode.trigger.TriggerDB.dburl = "localhost/test1"
# user id of the trigger database
org.semwebtech.applnode.trigger.TriggerDB.dbuser = "schenk"
# password for the trigger database
org.semwebtech.applnode.trigger.TriggerDB.dbpasswd = "irgendeinpassowrt"

# --- ACA/ACTION ------------------------------------------------
# URL of the database containing the aca rules
org.semwebtech.applnode.aca.AcaMappingDB.dburl = "localhost/test1"
# user id for the aca database
org.semwebtech.applnode.aca.AcaMappingDB.dbuser = "schenk"
# password for the aca database
```

```
org.semwebtech.applnode.aca.AcaMappingDB.dbpasswd = "irgendeinpassowrt"


# MODELSERVER PROPERTIES ==========================================
# name of the model used for storage in database (has to be qname)
org.semwebtech.applnode.servlet.ModelServer.model.name = /
     "http://org.semwebtech.applnode/model/uni"
# URL of the model. The modelserver will try to read from this URL
# org.semwebtech.applnode.servlet.ModelServer.model.fetchurl = /
     "http://localhost:8080/semweb/RDF/uni.owl"
# language used in resource identified by fetchurl (RDF/XML, N3, ...)
org.semwebtech.applnode.servlet.ModelServer.model.inputlanguage = "RDF/XML"
# URL of the external Reasoner: PELLET
org.semwebtech.applnode.servlet.ModelServer.reasonerurl = "http://localhost:8081"

# URL PROPERTIES ==================================================
# URL of the EventBroker, Domain Broker?
org.semwebtech.events.broker.url = "http://localhost:8080/semweb/triggerserver"
# URL of the RDFTriggerServer
org.semwebtech.applnode.servlet.rdfserver.url = "http://localhost:8080/semweb/rdfserver"
# URL of the RDFTriggerServers action interface
org.semwebtech.applnode.servlet.rdfserver.actions.url = /
     "http://localhost:8080/semweb/rdfserver/actions"
# base URL of ModuleAccessorServlet
# so that ${BASE_URL}/MODULE_NAME is a valid URL
org.semwebtech.applnode.servlet.ModuleAccessorServlet.url= /
     "http://localhost:8080/semweb/aca/modules"

# NAMESPACE PROPERTIES ============================================
# logvars namespace
org.semwebtech.lang.logvars.namespace = "http://www.semwebtech.org/lang/2006/logic#"
# ACA namespace
org.semwebtech.lang.aca.namespace = "http://www.semwebtech.org/lang/2006/aca#"
# rdf update namespace
org.semwebtech.lang.rdfu.namespace = "http://www.semwebtech.org/lang/2006/rdfupdate#"
# namespace of the application node
org.semwebtech.applnode.namespace = "http://www.semwebtech.org/2006/application-node#"
# mars
org.semwebtech.mars.namespace = "http://www.semwebtech.org/2006/mars#"
# RDF namespace
org.w3.rdf.namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
# RDFS namespace
org.w3.rdfs.namespace="http://www.w3.org/2000/01/rdf-schema#"
#OWL namespace
org.w3.owl.namespace="http://www.w3.org/2002/07/owl#"
```

[Filename:config/configuration.properties

B.2 Node Initialization Actions

```
<applnode:actionSequence xmlns:rdfu="http://www.semwebtech.org/lang/2006/rdfupdate#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:aca="http://www.semwebtech.org/lang/2006/aca#"
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#">

  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/mars-domain.n3"
    input-language="N3" />
  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/uni-domain.n3"
    input-language="N3" />
  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/uni-services.n3"
    input-language="N3" />
  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/uni-instances.n3"
    input-language="N3" />
  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/node-domain-actions.n3"
    input-language="N3" />
  <applnode:read-rdf url="http://localhost:8080/semweb/RDF/rdf-update-actions.n3"
    input-language="N3" />

  <applnode:register-aca-function-library
    name="applnode"
    url="http://localhost:8080/semweb/aca-module-defs/applnode.xq" />
  <applnode:register-aca-function-library
    name="basic"
    url="http://localhost:8080/semweb/aca-module-defs/basic.xq" />
  <applnode:register-aca-function-library
    name="rdfu"
    url="http://localhost:8080/semweb/aca-module-defs/rdfu.xq" />
  <applnode:register-aca-function-library
    name="rdfs"
    url="http://localhost:8080/semweb/aca-module-defs/rdfs.xq" />
  <applnode:register-aca-function-library
    name="owl"
    url="http://localhost:8080/semweb/aca-module-defs/owl.xq" />
  <applnode:register-aca-function-library
    name="rdf"
    url="http://localhost:8080/semweb/aca-module-defs/rdf.xq" />
  <applnode:register-aca-function-library
    name="uni"
    url="http://localhost:8080/semweb/aca-module-defs/uni.xq" />
  <applnode:register-aca-mapping
    name="add-professor_mapping_001"
    action-namespace="http://localhost/test.owl#"
    action-name="add-professor"
    owner-resource="uri"
    language="xquery"
    url="http://localhost:8080/semweb/aca-mapping-defs/add-professor_mapping_001.xq" />

  <applnode:register-trigger name="testtrigger1"
    url="http://localhost:8080/semweb/trigger-defs/testtrigger1.txt" />
  <applnode:register-trigger
    url="http://localhost:8080/semweb/trigger-defs/test_uniquePraesident.txt" />
  <applnode:register-trigger
    url="http://localhost:8080/semweb/trigger-defs/test_delete_1.txt" />
  <applnode:register-trigger
    url="http://localhost:8080/semweb/trigger-defs/test_ask_1.txt" />

</applnode:actionSequence>
```

[Filename:config/nodeInit.xml

Appendix C

ACA Mapping XQuery modules

C.1 Application Node Module

```
module namespace applnode = "http://www.semwebtech.org/2006/application-node#";

declare function applnode:delete-aca-mapping($name as xs:string) {
        <applnode:delete-aca-mapping applnode:name="{$name}"/>
};

declare function applnode:delete-aca-function-library($name as xs:string) {
        <applnode:delete-aca-function-library applnode:name="{$name}"/>
};

declare function applnode:delete-trigger($name as xs:string) {
        <applnode:delete-trigger applnode:name="{$name}"/>
};

declare function applnode:condition ($ask as xs:string,
 $actions as element()*) {
      <applnode:condition ask="{$ask}">
           $actions
      </applnode:condition>
};

declare function applnode:register-aca-mapping($name as xs:string,
                                     $action_ns as xs:string,
                                     $action_name as xs:string,
                                     $lang as xs:string,
                                     $def as xs:string) {
      <applnode:register-aca-mapping>
        <applnode:name>{$name}</applnode:name>
        <applnode:action-namespace>{$action_ns}</applnode:action-namespace>
        <applnode:action-name>{$action_name}</applnode:action-name>
        <applnode:language>{$lang}</applnode:language>
        {
         $def
        }
      </applnode:register-aca-mapping>
};

declare function applnode:register-trigger($def as xs:string) {
      <applnode:register-trigger>
        {
          $def
        }
      </applnode:register-trigger>
};

declare function applnode:read-rdf($url, $lang as xs:string) {
      <applnode:read-rdf>
        <applnode:url>{$url}</applnode:url>
        <applnode:language>{$lang}</applnode:language>
      </applnode:read-rdf>
};

declare function applnode:raiseEvent($event as element()*) {
      <applnode:raise-event>
        {
          $event
        }
      </applnode:raise-event>
};

declare function applnode:raiseEventWhen($when, $events as element()*) {
        <applnode:raise-event>
        {(
            <applnode:when>{string($when)}</applnode:when>,
            $events
        )}
        </applnode:raise-event>
};

declare function applnode:raiseDirectedEvent($target_url,
                                             $event as element()*) {
      <applnode:raise-directed-event applnode:target-url="{$target_url}">
        {
             $event
        }
      </applnode:raise-directed-event>
};

declare function applnode:raiseDirectedEventWhen($when,
                                                 $target_url,
                                                 $events as element()*) {
        <applnode:raise-directed-event applnode:target-url="{$target_url}">
        {(
            <applnode:when>{string($when)}</applnode:when>,
            $events
        )}
        </applnode:raise-directed-event>
};

declare function applnode:actionSequence($actions as element()*) as element() {
        <applnode:actionSequence
         xmlns:rdfu="http://www.semwebtech.org/2006/rdfupdate#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#">
        {
             for $action in $actions
             return $action
        }
        </applnode:actionSequence>
};

declare function applnode:nodeSequence($actions as element()*) as element() {
        <applnode:nodeSequence
         xmlns:rdfu="http://www.semwebtech.org/2006/rdfupdate#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#">
        {
             for $action in $actions
             return $action
        }
        </applnode:nodeSequence>
};
```

[Filename:aca-mapping-defs/applnode.xq]

C.2 ACA Module

```
module namespace aca = "http://www.semwebtech.org/lang/2006/aca#";

declare function aca:getProperty($node as element(), $name as xs:string)
        as xs:string {
        if ($node/@*[./name() = $name]) then $node/@*[./name() = $name]/string()
        else if ($node/*[./name() = $name]) then $node/*[./name() = $name]/text()
        else ""
};
```

[Filename:aca-mapping-defs/basic.xq]

C.3 OWL Module

```
module namespace owl = "http://www.w3.org/2002/07/owl#";
declare function owl:ns($name as xs:string) as xs:string {
concat("http://www.w3.org/2002/07/owl#", $name)
};
```

[Filename:aca-mapping-defs/owl.xq]

C.4 RDF Module

```
module namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
declare function rdf:ns($name as xs:string) as xs:string {
concat("http://www.w3.org/1999/02/22-rdf-syntax-ns#", $name)
};
```

[Filename:aca-mapping-defs/rdf.xq]

C.5 RDFS Module

```
module namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";
declare function rdfs:ns($name as xs:string) as xs:string {
concat("http://www.w3.org/2000/01/rdf-schema#", $name)
};
```

[Filename:aca-mapping-defs/rdfs.xq]

C.6 RDF Update Module

```
module namespace rdfu = "http://www.semwebtech.org/lang/2006/rdfupdate#";
declare namespace rdfs = "http://www.w3.org/2000/01/rdf-schema#";
import module namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
at  "http://localhost:8080/semweb/aca/modules/rdf";

declare function rdfu:stringLiteral($obj) {
        if (exists($obj)) then
```

```
            concat('"',$obj, '"^^<http://www.w3.org/2001/XMLSchema#string>')
        else $obj
};
declare function rdfu:intLiteral($obj) {
        if (exists($obj)) then
            concat($obj, '^^<http://www.w3.org/2001/XMLSchema#int>')
        else $obj
};
declare function rdfu:doubleLiteral($obj) {
        if (exists($obj)) then
            concat($obj, '^^<http://www.w3.org/2001/XMLSchema#double>')
        else $obj
};
declare function rdfu:booleanLiteral($obj) {
        if (exists($obj)) then
            concat($obj, '^^<http://www.w3.org/2001/XMLSchema#boolean>')
        else $obj
};
declare function rdfu:isLiteral($obj as xs:string) as xs:boolean {
        if (starts-with($obj, '"')) then true()
        else if (contains($obj, '^^')) then true()
        else false()
};

declare function rdfu:insert($sub, $pre, $obj) {
         if (exists($obj)) then
         (
            <rdfu:insert>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
            </rdfu:insert>
         ) else ()
};
declare function rdfu:delete($sub, $pre as xs:string, $obj) {
         if (exists($obj)) then
         (
            <rdfu:delete>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
            </rdfu:delete>
         ) else ()
};
declare function rdfu:retract($sub, $pre as xs:string, $obj) {
         if (exists($obj)) then
         (
            <rdfu:retract>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
            </rdfu:retract>
         ) else ()
};
declare function rdfu:assert($sub, $pre as xs:string, $obj) {
         if (exists($obj)) then
         (
            <rdfu:assert>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
            </rdfu:assert>
         ) else ()
};
declare function rdfu:delete-resource($sub) {
         if (exists($sub)) then
         (
            <rdfu:delete-resource rdf:resource="{string($sub)}" />
         ) else ()
};
declare function rdfu:update-subject($sub, $pre as xs:string, $obj, $new) {
         if (exists($obj) and exists($new)) then
         (
            <rdfu:update>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
              <rdfu:set>
                     <rdf:subject rdf:resource="{string($new)}" />
              </rdfu:set>
            </rdfu:update>
         ) else ()
};
declare function rdfu:update-predicate($sub, $pre as xs:string, $obj, $new) {
         if (exists($obj) and exists($new)) then
         (
            <rdfu:update>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
              <rdfu:set>
                     <rdf:predicate rdf:resource="{string($new)}" />
              </rdfu:set>
            </rdfu:update>
         ) else ()
};
declare function rdfu:update-object($sub, $pre as xs:string, $obj, $new) {
         if (exists($obj) and exists($new)) then
         (
            <rdfu:update>
          <rdf:subject rdf:resource="{string($sub)}" />
          <rdf:predicate rdf:resource="{string($pre)}" />
              {  if (rdfu:isLiteral($obj))
                 then <rdf:object>{string($obj)}</rdf:object>
                 else <rdf:object rdf:resource="{string($obj)}" />
              }
              <rdfu:set>
                 {  if (rdfu:isLiteral($new))
                    then <rdf:object>{string($new)}</rdf:object>
                    else <rdf:object rdf:resource="{string($new)}" />
                 }
              </rdfu:set>
            </rdfu:update>
         ) else ()
};
declare function rdfu:rename($old, $new) {
         if (exists($old) and exists($new)) then
         (
            <rdfu:rename rdf:resource="{string($old)}" >
              <rdfu:new-value rdf:resource="{string($new)}" />
            </rdfu:rename>
         ) else ()
};
declare function rdfu:rename-property-of-class($old, $new, $class) {
         if (exists($old) and exists($new) and exists($class)) then
         (
            <rdfu:rename rdf:resource="{string($old)}" >
              <rdfu:new-value rdf:resource="{string($new)}" />
              <rdfs:Class rdf:resource="{string($class)}" />
            </rdfu:rename>
         ) else ()
};
```

[Filename:aca-mapping-defs/rdfu.xq]

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] Apache Ant, a java-based build tool. `http://ant.apache.org/`.

[3] Description Logic Implementation Group (DIG). http://dl.kr.org/dig/.

[4] Document object model (DOM). `http://www.w3.org/DOM/`, 1998.

[5] Sun microsystems, inc. the source for java developers. `http://java.sun.com/`.

[6] Jena: A java framework for semantic web applications. `http://jena.sourceforge.net`.

[7] Michael Kay. SAXON: the XSLT and XQuery processor. `http://saxon.sf.net/`.

[8] Tobias Knabke. Development and implementation of a domain broker for the semantic web. Master Thesis, Univ. Göttingen, 2006.

[9] Apache logging services. `http://logging.apache.org/log4j/docs/index.html`.

[10] Wolfgang May, Franz Schenk, and Elke von Lienen. Extending an owl web node with reactive behavior. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 4187, pages 134–148. Springer, 2006.

[11] Notation3 (N3) a readable rdf syntax. `http://www.w3.org/DesignIssues/Notation3`.

[12] OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, 2004.

[13] Pellet: An OWL DL reasoner. Maryland Information and Network Dynamics Lab, `http://www.mindswap.org/2003/pellet`.

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

[15] RDF/XML syntax specification (revised). `http://www.w3.org/TR/rdf-syntax-grammar/`.

[16] Resource Description Framework (RDF) Schema specification. `http://www.w3.org/TR/rdf-schema/`, 2000.

[17] Daniel Schubert. Development of a prototypical event-condition-action engine for the semantic web. Bachelor Thesis, Univ. Göttingen, 2005.

[18] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2006.

[19] Elke von Lienen. Entwicklung eines RDF-Web-Services mit Trigger-Funktionalität. Diplomarbeit, TU Clausthal (in german), 2006.

[20] W3C – the world wide web consortium. http://www.w3.org/.

[21] W3C RDF validation service. http://www.w3.org/RDF/Validator.

[22] XQuery: A Query Language for XML. http://www.w3.org/TR/xquery, 2001.

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