Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2009-05

Masterarbeit

im Studiengang "Angewandte Informatik"

RDF Rules in the MARS Framework

Daniel Schubert

am Lehrstuhl für

Datenbanken & Informationssysteme

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

20. April 2009

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-17 20 10
Fax +49 (5 51) 39-1 46 93
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 20. April 2009

Master Thesis

RDF Rules in the MARS Framework

Daniel Schubert

April 20, 2009

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

Abstract

MARS (Modular Active Rules in the Semantic Web) is a framework for describing and implementing reactive behavior in the Semantic Web by the use of rules that follow the Event-Condition-Action (ECA) paradigm. Such ECA rules consist of an event specification (defining the sequence of events that leads to the execution of the rule), a condition (used to gather additional information and decide whether or not to execute the action) and an action specification (describing the actions to be taken in case the condition is satisfied).

By separately implementing the ECA semantics, the MARS framework allows for embedding arbitrary languages to specify the actual semantics of the individual components. This modularity facilitates the heterogeneity of the potential sublanguages, while simultaneously supporting their meta-level homogeneity.

Until now, ECA rules were specified and executed at the XML level, using the markup language ECA-ML which is provided by the framework. In order to turn the rules and their components into objects of the Semantic Web and support reasoning about them, it is necessary to express their semantics on the higher, ontology-based RDF/OWL level.

Therefore, this thesis introduces the respective MARS ontology for ECA rules along with an ontology for component languages and their implementing services. Based on these ontologies, it discusses the potential ways for handling such RDF rules in the MARS framework and implements an according proof-of-concept service.

Contents

List of Figures

1 Introduction

The Internet and in particular the World Wide Web is in continuous flux, changing from a web of static resources and isolated web applications towards a web of interconnected, loosely coupled and autonomously evolving information systems. In this *Semantic Web*, data is not only given in a human-readable form but also structured and marked up to allow for computerized processing.

While this already enables the nodes of the Semantic Web to query the presently existing information, they also need a way to become aware of changes to that information and to react to such *events* according to their respective tasks.

This behavior can adequately be formalized by the use of *Event-Condition-Action (ECA)* rules. Similar to triggers in active databases, ECA rules are evaluated when the node becomes aware of the occurrence of the specified event. It then evaluates the condition defined by the rule and finally, if applicable, executes the action specification.

This approach forms the basis of the MARS framework (Modular Active Rules for the Semantic Web) [4; 6] that deals with the specification and implementation of ECA rules in the Semantic Web. Instead of forcing specific languages to be used for defining a rule, it only specifies the semantics of the outer rule level and how the individual components of the rule communicate with each other by the use of variables.

One of the core services of the framework is the ECA engine [14]. This service controls the whole (distributed) evaluation process and invokes other services that can deal with the respective languages as needed while keeping track of the current variable bindings.

Although the ECA rules of the MARS framework provide the required functionality to implement the behavior of a Semantic Web node, they are currently not ”semantic” themselves. The rules are specified at the XML level which forces them to have a very detailed knowledge about their environment[1] and eliminates the possibility of a detailed reasoning about them and their components in the future.

Thus, it is necessary to express the ECA rules at a higher, more abstract level which also incorporates semantic concepts. These requirements are met by the Resource Description Framework (RDF) which is a W3C recommendation and part of their Semantic Web stack. The

[1] They must know exactly which data sources to query and how to access them.

main topic of this thesis therefore is the integration of RDF-based ECA rules in the MARS framework.

The next chapter starts by giving an introduction to the Semantic Web (including XML and RDF) and domain ontologies in general before introducing the concept of ECA rules. Furthermore, it outlines the architecture of the MARS framework followed by an exemplary evaluation of an XML-based ECA rule.

Chapter 3 then gives a comprehensive overview of the MARS ontology, introducing meta-level notions about domains, languages, services, and expressions. Additionally, it shows the ontology of ECA rules along with two exemplary language ontologies (OWLQ and SNOOP).

The evaluation of RDF rules is discussed in detail in Chapter 4, explaining why it is currently not preferable to evaluate them directly at the RDF level but to convert them to XML and register them at an existing ECA engine beforehand. Section 4.1 then presents an overview of the conversion process as a whole, followed by a description of a DTD-driven mapping procedure in Section 4.2. Afterwards, Section 4.3 discusses several approaches to apply this procedure. The chosen and implemented approach is then demonstrated by an exemplary conversion shown in Section 4.4. Finally, Section 4.5 deals with the reordering of the query components to ensure that their evaluation order is safe.

After the theoretical background has been explained, Chapter 5 gives a brief overview of how the rule conversion service was implemented before Chapter 6 finally concludes the thesis and gives and outline of the necessary steps towards a rule evaluation that is executed directly at the semantic level.

2 The MARS Framework

2.1 Semantic Web

In addition to the current form of data representation in the World Wide Web which is mainly intended for human readability and thus focused on visual layout, data in the Semantic Web is also marked up in a form processable by computers.

Similar to the W3C recommendations that standardize data representation and layout inside the World Wide Web (HTML, CSS, etc.), there is a stack of recommendations that deal with the architecture of the Semantic Web. The foundation of this stack is formed by URIs[1] and Unicode. While URIs provide a way to unambiguously identify resources, the use of Unicode as the standard data encoding avoids possible internationalization and localization issues. On top of this, XML provides a way to mark up semistructured data and to easily exchange it between different nodes.

Up to this point the mentioned concepts only deal with how data is encoded, marked up and identified, but as the name Semantic Web implies, there has to be a way to express semantic concepts. This task is performed by RDF and OWL which add the ability to express real-world concepts and their relations and allow for reasoning about them.

XML. The Extensible Markup Language (XML) [17] provides a convenient way for marking up semistructured data in a generic and self-explanatory manner. At first glance, the data-centric language XML looks very similar to the layout-centric language HTML[2]. Documents of both languages are serialized using plain text files which makes them readable by even the simplest editor. Furthermore, both languages apply the concept of element tags and attributes to mark up their content, but in contrast to HTML which provides a predefined set of tags with given layout semantics, XML is designed to be very flexible and extensible and thus does only minimally restrict the use of element names and document structure.

However, XML documents must strictly adhere to several syntax rules, in particular there has to be a closing tag for every element and all elements must be nested properly. If a document

[1] Uniform Resource Identifier
[2] In fact, there is an XML-based variant of HTML, called XHTML

```
<?xml version="1.0"?>
<!DOCTYPE CarRental SYSTEM "car-rental.dtd">
<CarRental xmlns="http://car-rental.nop">
    <cars>
        <car name="Golf" class="B"/>
        <car name="C4" class="B"/>
        <car name="Corolla" class="B"/>
        <car name="Passat" class="C"/>
        <car name="Focus" class="C"/>
        <car name="C6" class="D"/>
    </cars>
    <customers>
        <customer name="John Doe" mail="john@doe.nop" max-price="60">
            <car>Golf</car>
            <car>Passat</car>
        </customer>
        <customer name="Lisa Miller" mail="lisa@miller.nop" max-price="70">
            <car>Corolla</car>
        </customer>
        <customer name="Jack Miller" mail="jack@miller.nop" max-price="85">
            <car>Focus</car>
        </customer>
    </customers>
    <cities>
        <city name="Paris">
            <car name="Golf" price="65"/>
            <car name="C4" price="50"/>
            <car name="C6" price="150"/>
        </city>
        <city name="Munich">
            <car name="Passat" price="80"/>
        </city>
    </cities>
</CarRental>
```

Figure 2.1: The XML Database of a Fictional Car-Rental Company

satisfies all syntax rules, it is called *well-formed*. Because of the strict enforcement of these syntactical rules, XML parsers are algorithmically very simple and efficient.

XML documents can contain additional restrictions wrt. their vocabulary (element and attribute names) and structure. These restrictions can either be embedded directly or via an external schema language like DTD or XML Schema. If all of the additional requirements are met as well, the document is said to be *valid*.

To be able to intermix notions from different XML vocabularies in a single document, each element and attribute can be assigned to a *namespace* (usually given as a URI). Thus, it will still be uniquely identifiable, even if another element or attribute of the same name exists in a different vocabulary.

```
<!ELEMENT CarRental (cars, customers, cities)>

<!ELEMENT cars (car*)>
<!ATTLIST car name CDATA #REQUIRED
              price CDATA #REQUIRED
              class CDATA #IMPLIED>

<!ELEMENT customers (customer*)>
<!ELEMENT customer (car*)>
<!ATTLIST customer name CDATA #REQUIRED
                   mail CDATA #IMPLIED
                   max-price CDATA #IMPLIED>

<!ELEMENT cities (city*)>
<!ATTLIST city name CDATA #REQUIRED>
```

Figure 2.2: An Exemplary DTD (see Figure 2.1)

Example 2.1 *Figure 2.1 shows an exemplary, valid XML document that represents the database of a fictional car-rental company. It contains a list of cars and their classes, information about the company's customers and stores in a hierarchical way the cars that are available at each of the company's branches.*

DTD. Document Type Definitions are used to define the schema of XML documents for a particular application. Therefore, a DTD contains a list of all allowed elements, their mandatory and optional attributes and the respective content. The latter can either be simple character data or a list of further elements that may be nested inside the parent.

Example 2.2 *An exemplary DTD that can be used to validate the XML document of the previous example is shown in Figure 2.2.*

RDF. While XML provides a reasonable way to store and exchange data between nodes, the Resource Description Framework (RDF) [11] aims at representing knowledge (i.e. facts) about resources. In contrast to XML with its nearly arbitrary document structure, it is based on an extremely simple data model.

Expressed in *triples*, every RDF statement associates a subject and an object with a predicate (i.e. a verb). Since a resource can be anything that is identifiable by a URI, statements cannot only be made about existing web resources but also about real-world concepts like people, colors or prices.

From a theoretical point of view, an RDF document defines a directed, labeled graph that allows multiple connections between its nodes. In this graph, each directed edge represents an RDF statement with the label of the edge being the predicate, the start node being the subject and the end node being the object.

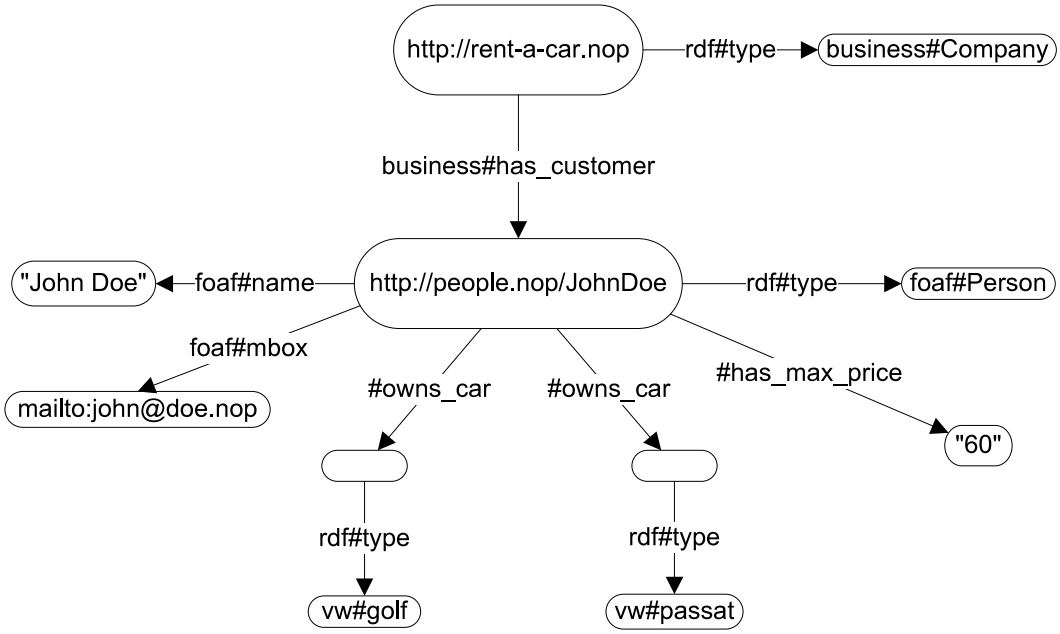


Figure 2.3: Excerpt of an RDF Graph Representing a Fictional Car-Rental Company

Example 2.3 *Consider again the car-rental company of the previous examples. An excerpt of an RDF graph that represents this company is shown in Figure 2.3. The company itself is identified by the URI* `http://rent-a-car.nop` *and is of the type* `business#Company`*. It has one customer,* `http://people.nop/JohnDoe`*, whose name is given using the according FOAF*[3] *properties. Furthermore, the customer owns two cars, a* `vw#golf` *and a* `vw#passat` *and is willing to pay no more than 60€ a day for renting a car.*

RDF/XML. While there are a number of formats to serialize an RDF graph[4], this thesis solely uses the RDF/XML format [13].

As the name suggests, this representation uses XML to store the triples of an RDF document. Its root is always constituted by an `<rdf:RDF/>` element which contains `<rdf:Description/>` elements for every resource it describes. The description element may have an `rdf:about` attribute that defines the URI of the resource it talks about. Inside, it contains an element for every triple in which the current resource is the subject. The names of these elements represent the predicates of the triples and contain either the object itself or a reference to its URI as the value of an `rdf:resource` attribute.

[3]Friend Of A Friend [3] is an ontology describing people and their relations.
[4]Most notably N3 [9].

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:cars="http://www.semwebtech.org/domains/2006/industry/cars"
         xmlns:vw="http://www.semwebtech.org/domains/2006/industry/cars/vw/"
         xmlns="http://rent-a-car.nop/">

    <business:company rdf:about="http://rent-a-car.nop">
        <business:has_customer rdf:resource="http://people.nop/JohnDoe" />
    </business:company>

    <foaf:Person rdf:about="http://people.nop/JohnDoe">
        <foaf:name>John Doe</foaf:name>
        <foaf:mbox rdf:resource="mailto:john@doe.nop" />
        <owns_car><vw:golf /></owns_car>
        <owns_car><vw:passat /></owns_car>
        <has_max_price>60</has_max_price>
    </foaf:Person>
</rdf:RDF>
```

Figure 2.4: The RDF/XML Representation of Figure 2.3

Since resources often contain at least one `rdf:type` predicate, RDF/XML allows to drop the `<rdf:Description/>` element and exchange it with the qualified name of the type information.

Example 2.4 *Am exemplary RDF/XML document that represents the RDF graph of the previous example is shown in Figure 2.4. Note that it does not use* `<rdf:Description/>` *elements but the abbreviated form with the type information contained in the root node of the resources.*

RDFS and OWL. On top of XML and RDF in the Semantic Web stack is the language RDF Schema (RDFS) [12]. While RDF itself defines the general semantics of its subject-predicate-object expressions, RDFS is used to apply them to a specific application domain. It therefore allows to specify the vocabulary of RDF documents by introducing the concept of a class hierarchy and provides a way to make assertions about the classes of the subject and object of an RDF triple[5].

While RDFS already provides basic concepts for describing ontologies, its usability is very limited with respect to more complex real-world scenarios. Therefore, the Web Ontology Language (OWL) [10] extends it to also support the notions of cardinality and relations between different classes (e.g. transitivity). This also allows for the support of reasoning about RDF documents.

[5]For example, an according definition of the predicate "is parent of" would allow to conclude that its subject and object are persons.

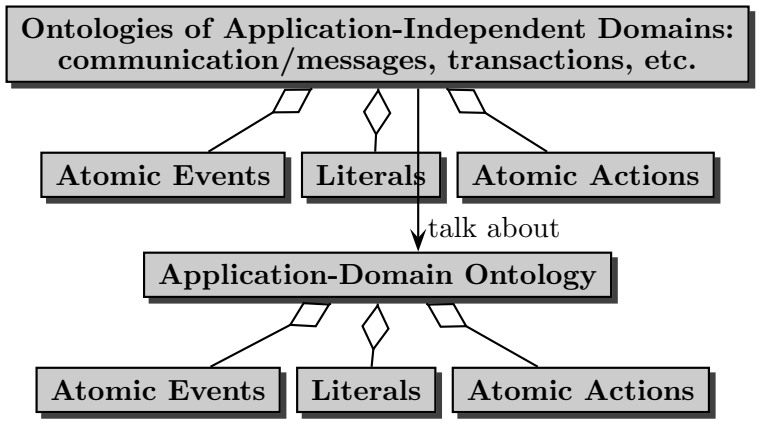


Figure 2.5: Kinds and Components of Ontologies (from [1])

2.2 Domain Ontologies

An ontology of a domain is a collection of statements used to describe all aspects that constitute that particular domain. The ontology not only contains information about the static aspects of the domain like classes and their relations, but also talks about its dynamic aspects, including events and actions.

As Figure 2.5 shows, two kinds of domain ontologies can be distinguished, those that are specific to a particular application (e.g. traveling or banking) and those that serve a more general purpose (e.g. messaging or transactions) and are therefore application-independent.

A Semantic Web application then usually combines several ontologies of both kinds but at least uses its main application domain ontology to perform its tasks.

Considering the dynamic aspects of a domain, an ontology also talks about the correlation between atomic actions and events, defining their pre- and post-conditions. Any action may hereby directly result in one or more atomic events. For example, the action `book-flight LH1234` results in the atomic event `flight-booked LH1234` but may additionally – if this was the last available seat – result in the atomic event `flight-fully-booked LH1234`.

2.3 ECA Rules

The use of Event-Condition-Action rules to describe the behavior of Semantic Web nodes is one of the main concepts upon which the MARS framework is built. This section will describe the semantics of ECA rule evaluation, the language binding of the individual rule components and present the XML-based markup language ECA-ML.

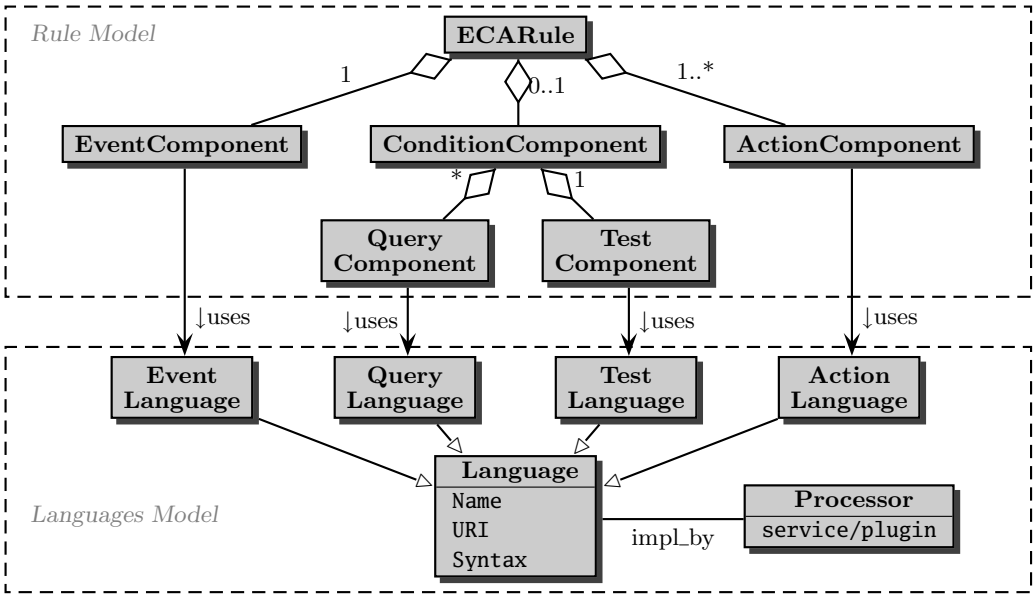


Figure 2.6: ECA Rule Components and Corresponding Languages (from [1])

2.3.1 Rule and Languages Model

As their name implies, ECA rules consist of an *event component*, a *condition component* and an *action component*.

While the event and action components specify upon which (series of) event(s) the rule should be evaluated and what actions are to be taken as a result, the condition component serves the purpose of gathering additional information and deciding for which data the actions are executed. For this reason, the condition component is split into a *query component* and a *test component* as shown in the upper part of Figure 2.6. The semantics of an ECA rule can then best be described as follows:

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

Example 2.5 *Consider again the fictional car-rental company of the previous examples. The company might want to automatically send offers to its customers whenever they book a flight to a destination where the company has a branch.*

An ECA rule to describe this functionality would then specify the event to be a flight booking. The query component might gather additional information about the customer (e.g. his e-mail address, his preferred type of car and the maximum price he is willing to pay) and after the test

component checked if there is at least one car that fits the requirements, the action component would specify to send an according offer to the customer.

Complementing the rule model, Figure 2.6 also shows that every component of a rule uses a corresponding language to specify its content. Every language is identified by its name and a unique URI and may be implemented by an arbitrary number of language processors. The MARS framework does not restrict the possible languages but only requires that for every language being used inside a rule there must be a least one service that is capable of evaluating expressions of that language.

At the level of XML, typical languages used in the event component are XPath [18] for specifying *atomic events* and SNOOP [2] for more complex expressions over several atomic events, so-called *composite events*. Query components may either use functional languages like XPath and XQuery [19] or logical languages like Datalog. Finally, an example for a very sophisticated language to be used in the action component is CCS [8] which is able to express complex processes.

2.3.2 Variables

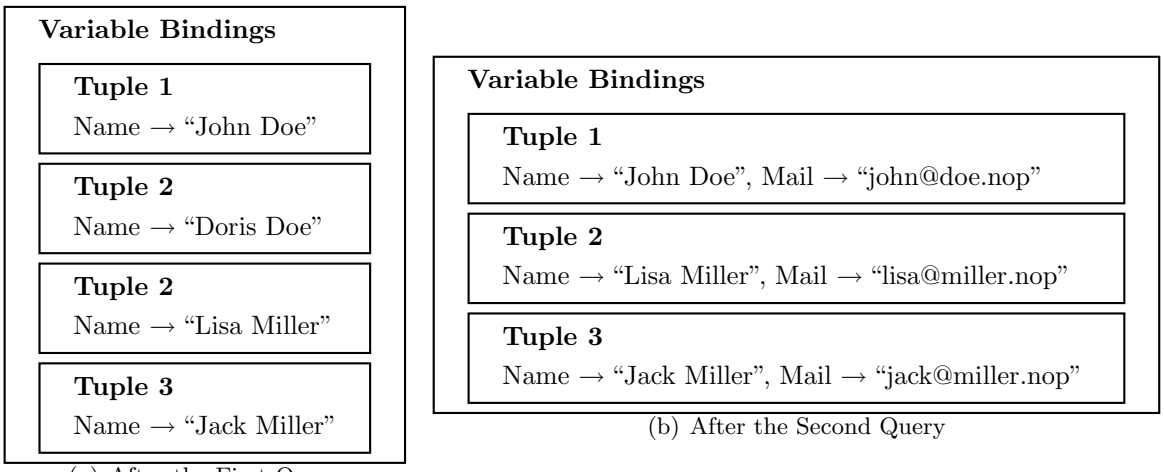
While the last section showed that arbitrary languages can be used in the different components of a rule, there still has to be a uniform way for the components to propagate their results to the following queries, tests and actions. Therefore, every rule component may make use of logical variables and bind an arbitrary number during its evaluation and/or use previously bound variables as input.

Since obviously a query can return more than just a single value for a variable, the variable bindings of ECA rules in the MARS framework are represented by sets of tuples. During the evaluation of a rule, query and test results are joined with the existing variable bindings by performing a natural equi-join. Every variable returned by the query (or test) that had been bound before acts as a join variable. Thus, at all times each tuple contains the same amount of variable bindings.

The use of variables in rule components is further detailed in Section 3.3.3.

Example 2.6 *Consider two subsequent queries of an ECA rule. The first returns the customer list of the fictional car-rental company, the second queries an open social networking site in order to gather information about their e-mail addresses.*

After the first query, the variable bindings might consist of four tuples, binding the variable "Name" to the values "John Doe", "Doris Doe", "Lisa Miller" and "Jack Miller" as shown in



(a) After the First Query

(b) After the Second Query

Figure 2.7: Multiple Tuples of Variable Bindings

Figure 2.7(a). Doris Doe, however, might not be enlisted at the networking site. Thus, no information about her is returned.

When the results of the second query have been joined with the previously existing ones, the final set of tuples looks as shown in Figure 2.7(b) with the tuple containing Doris Doe dropped during the join.

2.3.3 ECA-ML

The MARS framework provides the XML-based markup language ECA-ML [7] as a serialization format for ECA rules. Its structure and element naming largely follows the rule model shown in Figure 2.6. Being the root element, `<eca:rule/>` contains child elements for each subcomponent, namely `<eca:event/>`, `<eca:query/>`, `<eca:test/>` and `<eca:action/>` as illustrated in Figure 2.8.

Each of these subelements then contains the respective component itself, either in XML markup (if possible) or as the content of an `<eca:opaque/>` element. The association with the language of the component is done based on the type of expression used.

Atomic expressions exclusively belong to a single language which is therefore directly identified by the XML namespace of that expression. *Composite expressions* on the other hand are constructed by connecting multiple subexpressions of possibly different languages using a composer element of another language[6]. In this case, the language of the component is defined by the XML namespace of the composer (which also is the root element of the expression).

[6]For example, composers of the event algebra SNOOP include `AND`, `OR`, `SEQUENCE`, etc.

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

Figure 2.8: A Generic XML Rule in ECA-ML Markup

The last type, *opaque expressions*, are special in that their purpose is to integrate non-framework-aware services as well as languages without an XML markup into the MARS framework. Therefore, the surrounding `<eca:opaque/>` element has to contain the necessary information to facilitate their evaluation.

A more detailed explanation of ECA-ML can be found in [14].

2.4 Framework Architecture

So far, this chapter dealt with the fundamentals of the Semantic Web followed by an introduction to ECA-style rules in the MARS framework. This section now gives an overview of the distributed architecture of the framework and illustrates the cooperation between the different kinds of its components.

Figure 2.9 (which will be explained in detail in Example 2.7) shows that there are two major types of services, namely language and application domain services. The former provide the generic ability to evaluate (or execute) expressions of their particular language while the latter offer tasks like the relaying of events, queries and actions dependent on their respective domain.

ECA Engine. From the viewpoint of a client, the most important service of the MARS framework is the ECA engine. Its main purpose is to manage and evaluate the set of registered ECA rules by utilizing several of the remote language- and domain-specific services.

Upon the registration of a rule, the ECA engine inspects its event component and registers it at an appropriate event detection engine. When the given event pattern occurs, the event detection engine notifies the ECA engine of its occurrence and transmits the detected sequence of events along with the gathered variable bindings.

The ECA engine hereupon triggers the evaluation of the respective rule using the variable bindings of the event component as a basis. During the sequential evaluation every rule component is sent to an appropriate language processor together with the variable bindings that were specified as being required by the component. The language processor then handles the actual evaluation of the component and responds – in case of a query or a test component – with either a functional result (which has to be bound to a variable by the ECA engine) or with a set of variable bindings as well. Back at the ECA engine, the resulting variable bindings are joined with the initial variable bindings and the evaluation of the rule advances to the next component. This process is repeated until all components of the rule have been evaluated.

Language-Specific Services. As stated earlier, the purpose of language-specific services is to process expressions of their language, taking into account the set of variables bindings received from the initiating service[7].

Analogous to the components of an ECA rule, there are three main types of language services, namely event detection services, query/test services and action services. All of them are basically independent from any application domain but may interact with domain-specific services to perform their tasks.

Domain-Specific Services. These are services that can be attributed to a domain which may either be specific to a particular application (e.g. the travel domain) or have a more generic purpose (e.g. messaging) in which case it is said to be *application-independent* (cf. Section 2.2). In both cases the services that are provided directly relate to their domain.

For application domains there may also be one or more central domain nodes that act as an interface to that domain. Their purpose is to collect atomic events from inside the domain and relay them to registered services on the outside and to forward atomic actions back to the appropriate services inside the domain.

The Languages and Services Registry. Up to now this section only described how the different kinds of services interact with each other to perform their tasks but disregarded the basic question of how a service can discover other relevant services.

[7] Note that this does not necessarily have to be the ECA engine itself. Since every component may be composed of subexpressions of other languages, every language processor must be capable of invoking every other language processor if necessary.

Since the MARS framework is designed to be very open wrt. the languages and services used by its clients and inside the application domains, it provides a dynamic registry to enable service discovery which is called the Languages and Services Registry (LSR).

The MARS framework provides a set of abstract language types (cf. Section 3.2.1) and then allows its clients to add information about concrete languages of these classes at the LSR. For example, the MARS ontology defines the language type `QueryLanguage` and so a client may state that the language XQuery is of this type and provide information about its URI. But most important, the client can also register a service which handles expressions of that language by specifying its URL, capabilities and how it is to be invoked.

Therefore, the LSR can be used by any service to look up other services which fulfill his present requirements to perform a given task. Internally, most services of the MARS prototype use a common component – the Generic Request Handler (GRH) – to handle the actual communication with other services. The GRH takes a language URI and the name of a task as input, queries the LSR to find out which service to call and asks that service to actually perform the requested task.

The role of the LSR in connecting the various parts of the MARS framework on the ontology level is further detailed in Section 3.2.3.

Example 2.7 *The typical use case for the MARS framework is illustrated in Figure 2.9, where a client registers an ECA rule that uses notions of several languages (*`snoop`*,* `match` *and* `ccs`*) and application domains (*`travel` *and* `smtp`*) at an arbitrary ECA engine (1.1).*

Since the event component of the rule is written in SNOOP, the ECA engine registers it at a SNOOP event detection engine (1.2). In this example, the event component is composed of several atomic events whose description is given using notions of the `match` *namespace. Thus, the SNOOP event detection engine registers them at an appropriate atomic event matcher (AEM) (1.3). The AEM then analyzes the event descriptions and discovers that the events belong to the* `travel` *domain. Therefore, it registers itself to be notified about the relevant events by the domain broker of that domain (1.4).*

At this point the event component is fully registered and the actual detection of the described event pattern begins. The domain broker of the `travel` *domain continuously receives atomic events from nodes inside its domain (in this case Lufthansa (2.1a) and SNCF (2.1b)) and relays them to the registered services, now including the AEM (2.2).*

When then AEM finds an atomic event to match one of its registered event descriptions, it notifies the respective service (in this case, the SNOOP event detection engine) by sending the actual event along with any potentially gathered variable bindings (3). At some point, the composite event registered at the SNOOP engine is detected and the ECA engine gets a

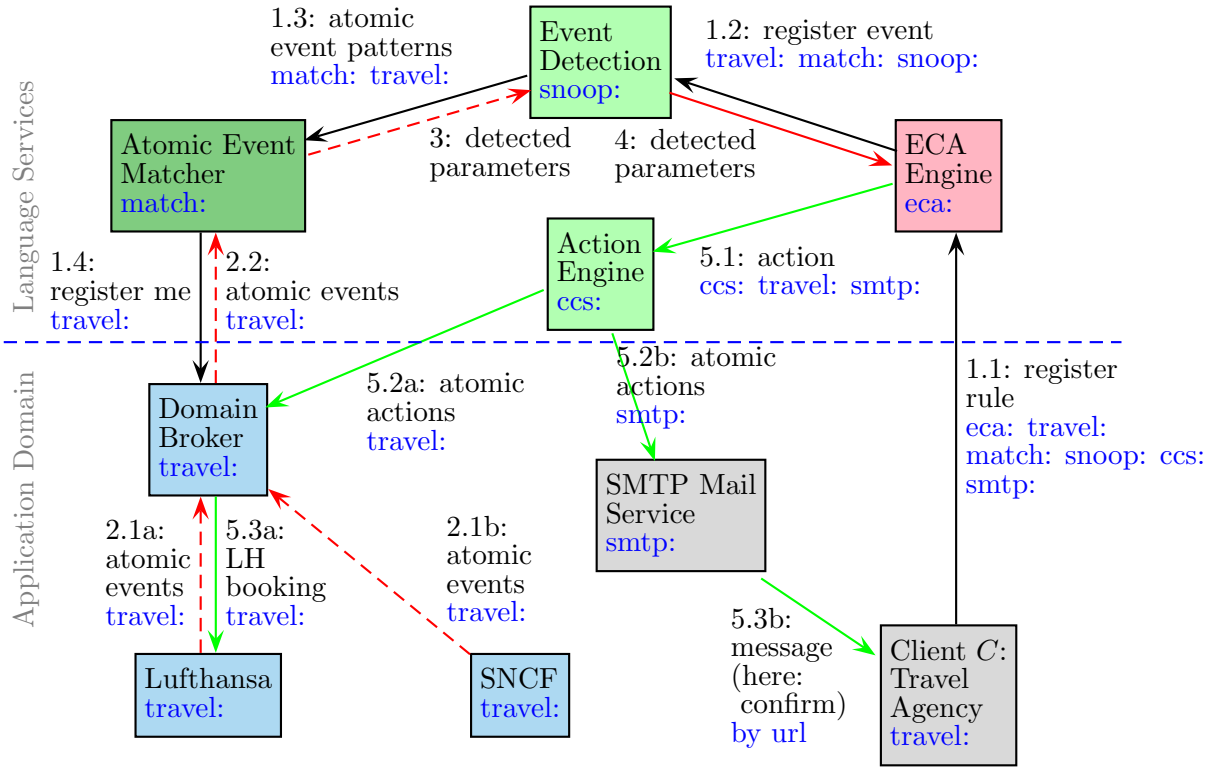


Figure 2.9: The Exemplary Registration and Execution of an ECA Rule

notification containing the detected sequence of events together with the variable bindings (4). This notification causes the respective rule at the ECA engine to be evaluated.

As described earlier, the ECA engine takes the received variable bindings as a basis for the evaluation of all query and test components of the rule in the given order and joins the results with the existing variable bindings (this part is omitted in the example).

Finally, the action components of the rule are executed. In the current example, there is one composite action component written in CCS which is send to the CCS engine (5.1). The composite action consists of two atomic actions, the first belonging to the `travel` *domain, the second to the* `smtp` *(messaging) domain. Therefore, the CCS engine sends the first action to the domain broker of the* `travel` *domain (5.2a) which relays it to the responsible domain node (5.3b). The second action is sent directly to a SMTP mail service (5.2b) which in turn sends a confirmation mail to the client (5.3b). This completes the evaluation of the rule for the detected sequence of events*[8].

[8]Note that a rule may be evaluated an arbitrary number of times. Any detected sequence of events that conforms to the description given in the event component causes the rule to be evaluated.

This chapter gave an introduction to the Semantic Web in general and explained its fundamental ideas and concepts. Afterwards, it introduced the notion of (application-related and application-independent) domain ontologies before explaining the concept of Event-Condition-Action rules in detail. Finally, it gave an overview of the architecture of the MARS framework and the interdependencies of its services.

The next chapter will now introduce the MARS ontology that describes the concepts of the MARS framework regarding domains, languages, services and ECA rules.

3 The MARS Ontology

The previous chapter dealt with the architecture of the MARS framework, gave an overview of its service infrastructure and explained the process of rule evaluation. In order to lift the framework functionality to the semantic level, its concepts have to be described in an ontology as well.

In general, there are four types of concepts that have to be described:

- meta concepts
- domain ontologies
- languages
- services

At first, it is necessary to describe the *meta concepts* applied in the MARS framework to fully capture the environment in which it is deployed. This mainly includes elementary notions like classes, properties, events, actions, languages and services.

Domain ontologies have already been shortly introduced in Section 2.2. They describe the static and dynamic notions of a particular domain, i.e. classes, their relationships, events and actions. The domain ontologies are not part of the MARS ontology itself but play a significant role when the MARS framework is applied to one or more specific domains.

Languages in general can be categorized into several language families. With respect to the MARS framework, the most important high-level language families (which can be described by ontologies) include ECA languages like ECA-ML (cf. Section 3.4.2), event algebras like SNOOP (cf. Section 3.5.2), query languages like OWLQ (cf. Section 3.5.1), and process algebras like CCS. On a lower level, there are programming languages that do not define a language ontology (e.g. XQuery, SQL, or atomic event matching formalisms) but are still relevant to the framework.

Finally, there are the *services* that actually evaluate expressions of the respective languages and therefore implement their semantics. Depending on what type of language a service implements, it offers a different set of tasks regarding that language (e.g. an event detection service will offer different tasks than a query evaluation service).

Figure 3.1 shows an overview of the MARS sub-ontologies that will now be explained in the subsequent sections.

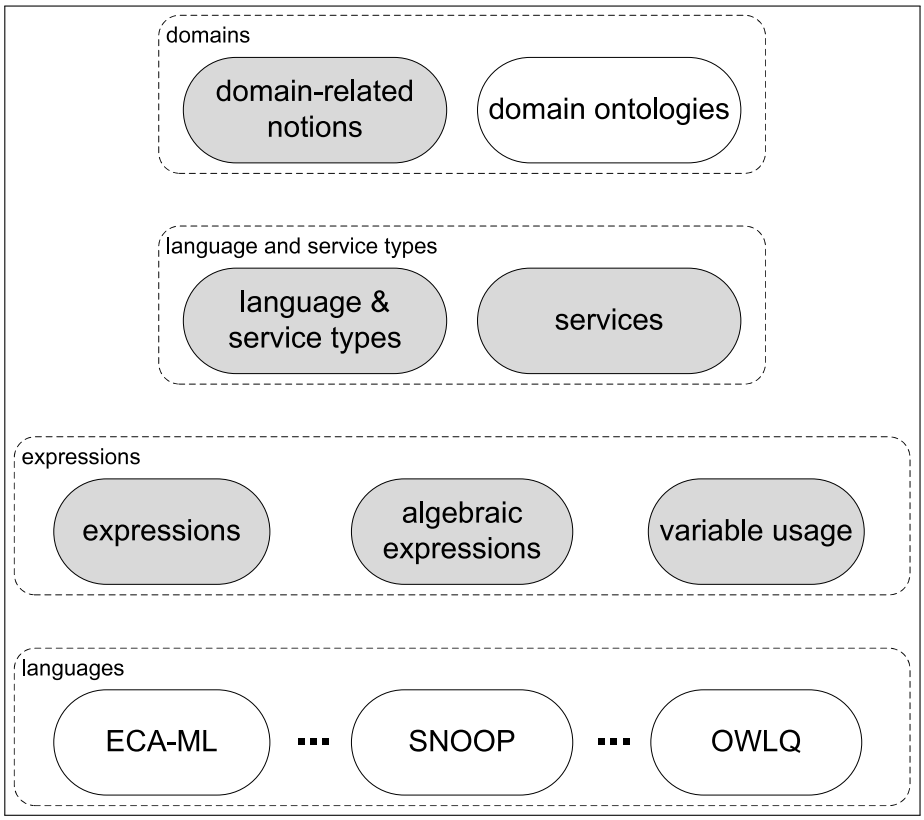


Figure 3.1: The MARS Sub-Ontologies and Language Ontologies

3.1 Domain-Related Notions

The domain-related part of the MARS ontology at a meta-level defines the concept of a domain itself. Important classes of this sub-ontology are *Domain* and *DomainNotion* with its subclasses *Class*, *Property*, *Event* and *Action*. Each *DomainNotion* is related to its *Domain* by the use of the *belongs-to-domain* property.

Figure 3.2 shows the sub-ontology of domain-related notions as an RDF/XML document.

3.2 Languages and Services

A central design goal of the MARS framework is the openness towards the diversity of applicable languages. Therefore, the sub-ontologies that deal with these concepts are not particular to any special language but only define the notions of language classes and the respective service classes that implement them. The actual mapping of languages and services then happens at the language and service registry which will be shown in Section 3.2.3.

```
<owl:Class rdf:ID="Domain"/>
<owl:Class rdf:ID="DomainNotion"/>

<owl:Class rdf:ID="Class">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="#DomainNotion"/>
</owl:Class>
<owl:Class rdf:ID="Property">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:subClassOf rdf:resource="#DomainNotion"/>
</owl:Class>
<owl:Class rdf:ID="Action">
  <rdfs:subClassOf rdf:resource="#DomainNotion"/>
</owl:Class>
<owl:Class rdf:ID="Event">
  <rdfs:subClassOf rdf:resource="#DomainNotion"/>
</owl:Class>

<rdf:Property rdf:ID="belongs-to-domain">
  <rdfs:domain rdf:resource="#DomainNotion"/>
  <rdfs:range rdf:resource="#Domain"/>
</rdf:Property>
```

Figure 3.2: The Sub-Ontology of Domain-Related Notions

3.2.1 The Languages and Services Sub-Ontology

As the name implies, this ontology mainly deals with the different kinds of languages and services that are used by the MARS framework. It introduces the notions *LanguageClass* and *ServiceClass* and states that every language class is implemented by a particular service class. Additionally, a language class may be assigned one or more *ResultTypes*, e.g. *VariableBindings* or *Answers*.

An individual *Language* then belongs to a *LanguageClass* and is therefore implemented by an arbitrary number of *Services*, each of which belongs to the according *ServiceClass*. It may also be related to information about different kinds of possible markup documents, e.g. by *hasStripedDTD* and *hasStrippedDTD*. These will be utilized by the DTD-driven mapping of RDF rules to XML (cf. Sections 4.2 and 4.4).

The ontology furthermore lists the existing language classes and connects them with their appropriate service classes. For example, the language class *ECALanguage* is implemented by an instance of the service class *ECAService* while a *QueryLanguage* is implemented by a service from the class *QueryService* and so on.

Finally, each *Service* provides a number of *Tasks* which can be invoked using the information given by a *TaskDescription*. Since on a meta-level, all services of a particular *ServiceClass* share a common interface, there is also a property named *meta-provides-task* which can be used to

```
<owl:Class rdf:ID="LanguageClass"/>
<owl:Class rdf:ID="Language"/>
<owl:Class rdf:ID="ResultType"/>

<LanguageClass rdf:ID="RuleLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
</LanguageClass>
<LanguageClass rdf:ID="ECALanguage">
  <rdfs:subClassOf rdf:resource="#RuleLanguage"/>
  <has-service-type rdf:resource="#ECAService"/>
</LanguageClass>
<LanguageClass rdf:ID="QueryLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
  <has-service-type rdf:resource="#QueryService"/>
</LanguageClass>

<rdf:Property rdf:ID="hasStripedDTD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
  <rdfs:range rdf:resource="#DTD"/>
</rdf:Property>
<rdf:Property rdf:ID="hasStrippedDTD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
  <rdfs:range rdf:resource="#DTD"/>
</rdf:Property>

<owl:Class rdf:ID="ServiceClass"/>
<owl:Class rdf:ID="Service"/>

<rdf:Property rdf:ID="has-service-type">
  <rdfs:domain rdf:resource="#LanguageClass"/>
  <rdfs:range rdf:resource="#ServiceClass"/>
</rdf:Property>
<rdf:Property rdf:ID="is-implemented-by">
  <rdfs:domain rdf:resource="#Language"/>
  <rdfs:range rdf:resource="#Service"/>
</rdf:Property>

<ServiceClass rdf:ID="ECAService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
  <ServiceClass rdf:ID="QueryService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>

<owl:Class rdf:ID="Task"/>
<owl:Class rdf:ID="Task-Description"/>

<rdf:Property rdf:ID="provides-task">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#Task"/>
</rdf:Property>
<rdf:Property rdf:ID="meta-provides-task">
  <rdfs:domain rdf:resource="#ServiceClass"/>
  <rdfs:range rdf:resource="#Task"/>
</rdf:Property>
```

Figure 3.3: An Excerpt of the Sub-Ontology of Language and Service Types

```
<LanguageClass rdf:ID="DomainLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
  <has-service-type rdf:resource="#DomainService"/>
</LanguageClass>

<ServiceClass rdf:ID="DomainService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<rdf:Property rdf:ID="uses-domain">
  <rdfs:domain rdf:resource="#DomainService"/>
  <rdfs:range rdf:resource="#Domain"/>
  <owl:inverseOf rdf:resource="#has-service"/>
</rdf:Property>
<rdf:Property rdf:ID="supports">
  <rdfs:domain  rdf:resource="#DomainService"/>
</rdf:Property>

<ServiceClass rdf:ID="DomainBroker">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<rdf:Property rdf:ID="has-domain-broker">
  <rdfs:domain rdf:resource="#Domain"/>
  <rdfs:range rdf:resource="#DomainBroker"/>
</rdf:Property>
```

Figure 3.4: Domain Languages and Services

associate a *Task* with a whole service class instead of a single service. This property is heavily used by the services sub-ontology that is the subject of the next section.

Figure 3.3 illustrates the previously explained concepts by showing an excerpt of the respective ontology given as RDF/XML.

Additionally, the ontology defines notions regarding domain languages and services as shown in Figure 3.4. A *DomainLanguage* deals with the notions of a domain (classes, properties, events and actions) and differs from the previously mentioned language types in that it only supports the actions of the respective domain directly while evaluating queries requires the additional support of a (usually external) query language.

A *DomainService* may then implement an arbitrary number of domains and *support* (a subset of) their notions. In a similar way, a *DomainBroker* may also support multiple domains and be responsible for the relaying of their atomic events and actions.

The complete languages and services ontology can be found in the design whitepaper [4] and at the project homepage [6].

```
<!DOCTYPE rdf:RDF [<!ENTITY mars "http://www.semwebtech.org/mars/2006/mars">]>
<ServiceClass rdf:about="#ECAService">
  <meta-provides-task rdf:resource="&mars;/eca-service#register-rule"/>
  <meta-provides-task rdf:resource="&mars;/eca-service#deregister-rule"/>
  <meta-provides-task rdf:resource="&mars;/eca-service#disable-rule"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/eca-service#enable-rule"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/eca-service#incoming-events"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/eca-service#receive-detected-event"/>
  <meta-provides-task rdf:resource="&mars;/eca-service#receive-query-answer"/>
  <meta-provides-task rdf:resource="&mars;/eca-service#give-service-description"/> <!-- opt -->
</ServiceClass>
<ServiceClass rdf:about="#QueryService">
  <meta-provides-task rdf:resource="&mars;/qs#evaluate-query"/>
  <meta-provides-task rdf:resource="&mars;/qs#evaluate-test"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#register-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#analyze-variables"/>  <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#deregister-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#invoke-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#validate-pattern"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#analyze-variables"/> <!-- opt -->
  <meta-provides-task rdf:resource="&mars;/qs#give-service-description"/> <!-- opt -->
</ServiceClass>
<ServiceClass rdf:about="#DomainService">
  <meta-provides-task rdf:resource="&mars;/domain-node#receive-query"/>
  <meta-provides-task rdf:resource="&mars;/domain-node#receive-action"/>
  <meta-provides-task rdf:resource="&mars;/domain-node#give-service-description"/> <!-- opt -->
</ServiceClass>
```

Figure 3.5: An Excerpt of the Sub-Ontology of Services

3.2.2 The Services Sub-Ontology

As explained in the previous section, each *LanguageClass* is implemented by an according *ServiceClass*. The services of that class are then required to provide certain *Tasks* in order to support the semantics of the implemented *LanguageClass*. An *ECAService*, for example, provides tasks regarding the management and execution of ECA rules while a *QueryService* deals with the evaluation (and possibly the pre-registration) of queries.

Therefore, each *ServiceClass* is tailored to the respective language type by defining the *Tasks* that represent the uniform functionality to be provided by the services of that class[1]. This information can then also be utilized to query the languages and services registry (which will be explained in the next section) for services that implement a certain *Language* and provide the desired tasks.

Figure 3.5 exemplarily shows the tasks common to the service classes *ECAService* and *QueryService* on the language side and *DomainService* on the domain-related side.

[1]Note that the list of tasks provided by a service class is not compulsive. A service may therefore omit some of the optional tasks if it does not implement the respective functionality.

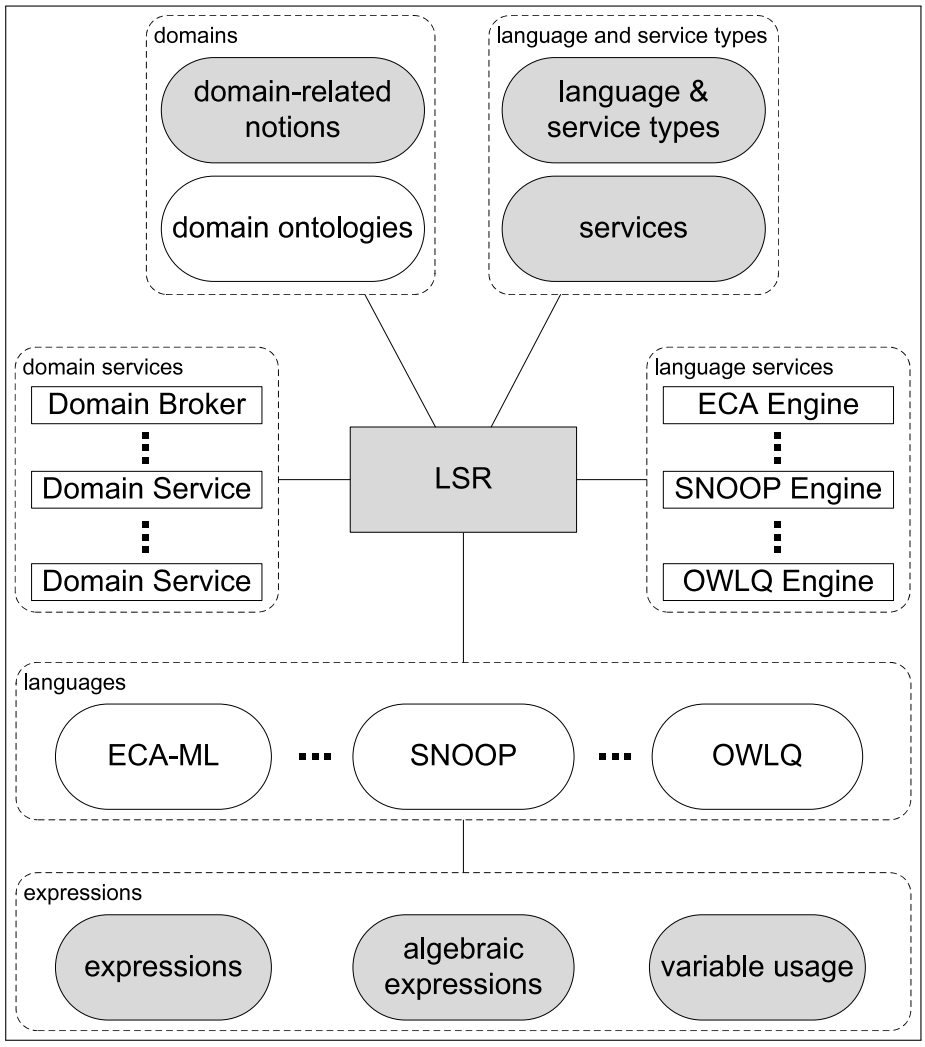


Figure 3.6: The LSR Connecting Ontologies and Services

3.2.3 The Languages and Services Registry RDF Model

The languages and services registry (LSR, cf. Section 2.4) connects the MARS ontology, the language ontologies and the domain ontologies. Additionally, it provides information about the actual languages and their implementing services as illustrated in Figure 3.6. In order to achieve this functionality in a way that accords to the idea of the Semantic Web, the LSR defines an RDF model that contains the respective information.

As an example, Figure 3.7 shows an excerpt of the LSR that deals with the rule language ECA-ML. In addition to giving its URI and name, the LSR lists several documents that define the markup of that language. Finally, the LSR contains information about a service that

```
<mars:ECALanguage rdf:about="http://www.semwebtech.org/languages/2006/eca-ml#">
 <mars:shortname>eca-ml</mars:shortname>
 <mars:name>ECA Markup Language</mars:name>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.rdf"/>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.dtd"/>
 <mars:hasStripedDTD rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml-striped.dtd"/>
 <mars:hasStrippedDTD rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.dtd"/>
 <mars:is-implemented-by>
  <mars:ECAService
     rdf:about="http://www.semwebtech.org/services/2007/eca-engine"
     xml:base="http://www.semwebtech.org/services/2007/eca-engine/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/eca-service#register-rule"/>
     <provided-at rdf:resource="register"/>
     <method>POST</method>
     <Reply-To>none</Reply-To>
     <Subject>body</Subject>
     <input>element register</input>
     <variables>no</variables>
    </TaskDescription>
   </has-task-description>
   ...
  </mars:ECAService>
 </mars:is-implemented-by>
</mars:ECALanguage>

<mars:Domain rdf:about="http://www.semwebtech.org/domains/2006/travel#">
 <mars:name>Traveling</mars:name>
 <mars:shortname>travel</mars:shortname>
 <mars:hasStripedDTD rdf:resource="http://www.semwebtech.org/domains/2006/travel-striped.dtd"/>
</mars:Domain>

<mars:DomainService rdf:about="http://www.semwebtech.org/nodes/2007/mars-railway">
  <mars:name>MARS Worldwide Railway Services</mars:name>
  <mars:uses-domain rdf:resource="http://www.semwebtech.org/domains/2006/travel"/>
  <!-- list of all names in the domain (events, actions, concepts, properties)
       that are supported by that application node -->
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#RailStation"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#RailConnection"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#from"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#to"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#duration"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#distance"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railDelay"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railDelayed"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railBook"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railBooked"/>
</mars:DomainService>
```

Figure 3.7: An Excerpt of the LSR

implements this language, namely the ECA engine. The service definition contains a list of task descriptions that state the specific details of how the respective tasks are to be invoked.

The LSR also keeps information about its known domains. Thus, Figure 3.7 additionally

shows an exemplary excerpt of the LSR that is concerned with the `travel` domain. Besides the definition of the domain itself, it also contains information about a domain service which supports various notions from the `railway` sub-ontology.

3.3 Expressions

A language is generally defined by giving its expressions and their semantics. The MARS framework provides an ontology that deals with the concept of expressions and consists of three sub-ontologies regarding general expressions, algebraic expressions and variables.

3.3.1 The General Expressions Sub-Ontology

This is the generic part of the expressions ontology as it covers algebraic as well as non-algebraic expressions. Figure 3.8 shows that any expression is either an *AtomicExpression*, a *CompositeExpression* or a *Variable*. A composite expression is characterized by the fact that it consists of several subexpressions.

Due to the fact that an atomic expression of one language can at the same time be a composite expression of another language, every expression may be related to two different languages. For example, an `eca#Event` is atomic in the ECA language but its definition may be a `snoop#And` which is a composite expression in the SNOOP language (cf. Section 3.5.2). In this case, the respective node belongs to both languages and therefore represents a language border.

3.3.2 The Algebraic Expressions Sub-Ontology

Algebraic expressions are a special form of expressions that define a term structure. Composite expressions of algebraic languages not only consist of subexpressions but also connect these by the use of operators of that language.

Speaking in terms of the respective sub-ontology, an *AlgebraicExpression* is a sub-class of an *Expression* which belongs to a language from the *AlgebraicLanguage* class. A *CompositeAlgebraicExpression* then is every *AlgebraicExpression* that is also a *CompositeExpression* as defined in the previous section. Furthermore, a *CompositeAlgebraicExpression* always has an *Operator* that connects its subexpressions which is provided by its language. While the semantics of the *Operator* are further defined by the language ontology itself, its *"arity"* property states exactly how many subexpressions it may connect.

The ontology for algebraic expressions is not shown here but can be found in [4] and at the project homepage [6].

```
<!DOCTYPE rdf:RDF [<!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#">]>
<rdf:RDF xml:base="http://www.semwebtech.org/mars/2006/expressions"
       xmlns:expr="http://www.semwebtech.org/mars/2006/expressions#">

 <!-- general languages and expressions -->
 <owl:Class rdf:about="Expression"/>

 <!-- note: not functional since the same node can e.g. be an ECA-ML and a SNOOP-Node -->
 <owl:Property rdf:about="#language">
  <rdfs:domain rdf:resource="#Expression"/>
  <rdfs:range rdf:resource="&mars;Language"/>
  <owl:maxCardinality>2</owl:maxCardinality>
 </owl:Property>

 <rdfs:Class rdf:about="#Expression">
  <owl:unionOf rdf:parseType="Collection">
   <rdfs:Class rdf:about="#Variable"/>
   <rdfs:Class rdf:about="#AtomicExpression">
    <owl:disjointWith rdf:resource="#Variable"/>
   </rdfs:Class>
   <rdfs:Class rdf:about="#CompositeExpression">
    <owl:disjointWith rdf:resource="#Variable"/>
    <owl:disjointWith rdf:resource="#AtomicExpression"/>
   </rdfs:Class>
  </owl:unionOf>
 </rdfs:Class>

 <rdf:Property rdf:about="#has-subexpression">
  <rdfs:domain rdf:resource="#CompositeExpression"/>
  <rdfs:range rdf:resource="#Expression"/>
 </rdf:Property>
</rdf:RDF>
```

Figure 3.8: The Sub-Ontology of General Expressions

3.3.3 The Variable Usage Sub-Ontology

As described in Section 2.3.2, the components of a rule propagate their results to the subsequent components by the use of variables. In addition to these variables, which are associated with the rule itself, variables may also be local to an expression. Thus, the relationship between a variable and an expression may be one of the following (as described in [4]):

uses-variable: a variable occurs in any way in (the context of) an expression.

free variables: a variable occurs free. Such variables belong to the "outer interface" of an expression.

bound variables: a variable occurs bound (i.e., in the scope of a quantifier). Such variables are not relevant to the outside of an expression.

```
<rdf:RDF xml:base="http://www.semwebtech.org/mars/2006/expressions"
         xmlns:expr="http://www.semwebtech.org/mars/2006/expressions#">
 <rdf:Property rdf:about="#uses-variable">
  <!-- variable occurs in some way in this expression -->
  <owl:inverseOf rdf:resource="#variable-occurs-in"/>
  <rdfs:domain rdf:resource="#Expression"/>
  <rdfs:range rdf:resource="#Variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-free-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-bound-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-positive-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-negative-variable">
  <rdfs:subPropertyOf rdf:resource="#has-input-variable"/>
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#scopes-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-input-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#has-output-variable">
  <!-- mainly for non-logical things -->
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
 <rdf:Property rdf:about="#bind-to-variable">
  <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
 </rdf:Property>
</rdf:RDF>
```

Figure 3.9: The Sub-Ontology of Variable Usage

scoping of variables: a quantifier scopes a variable, that (usually) occurs *free* in a subexpression and is then *bound* by the quantifier.

input variables: these variables must be bound when evaluating an expression. In logical frameworks, *negative* variables are input variables.

output variables: these variables are bound to values when evaluating an expression (e.g., OUT variables in procedures). If such a variable is already bound before evaluating an expression, it *may* be used as a join variable. For logical frameworks, it is recommended to use the notion of *positive* variables instead.

The complete sub-ontology of variable usage is shown in Figure 3.9 as RDF/XML.

3.4 ECA Rules at the Semantic Level

3.4.1 Implications of Semantic ECA Rules

When expressing ECA rules in RDF and OWL, they and their components directly refer to their respective (sub-)ontologies. Together with the component languages and their processors, this leads to a fully resource-based view of the framework as a whole. Thus, every rule, rule component, event etc. itself becomes an identifiable resource of the Semantic Web.

By lifting the rules to the semantic level, several kinds of reasoning about them become possible. One important aspect is that rules can be checked for validity, detecting incorrect uses of subcomponent languages (e.g. an action language used in the event component). Furthermore, every rule component can be analyzed with respect to its usage of variables. This eliminates the need for explicitly declaring the variables on the ECA level and additionally provides a way to ensure a *safe* order of evaluation of the rule components[2].

On a higher level, it becomes possible to reason about the behavior of a node as a whole, drawing conclusions considering the complete set of its registered rules. Furthermore, rules may be expressed at an even more abstract level, leaving the task to transform the abstract instructions into a concrete, evaluable language to the respective services.

3.4.2 The ECA-ML Ontology

The ontology of ECA rules in the MARS framework is similar to their XML representation. The main classes are *Rule*, *Event*, *Query*, *Test* and *Action* with the according properties *has-event*, *has-query*, *has-test* and *has-action*.

Additionally, the components of a rule may be *Opaque* (cf. Section 2.3.3) in order to support legacy and non-framework-aware services. In this case, the content of the component is not an RDF-enabled language but a string or an XML literal. The literal is given as the content of an *OpaqueSpec* and accompanied by the properties *language*, *uri* and *method* to specify the details of how the respective service is to be invoked.

In contrast to the XML representation of ECA rules, the query and test components of the RDF/OWL model do not have an inherent order in which they are to be evaluated. In fact, from a modeling point of view an ECA rule consists of an event specification, a condition component and an action specification.

Therefore, the ECA-ML ontology also contains the concepts *AtomicCondition*, *Conjunctive-Condition* and *ListCondition* with the implication being that a user can register rules that

[2]For example, every negative occurrence of a variable must be preceded by a positive one (the variable must be bound before its value can be used).

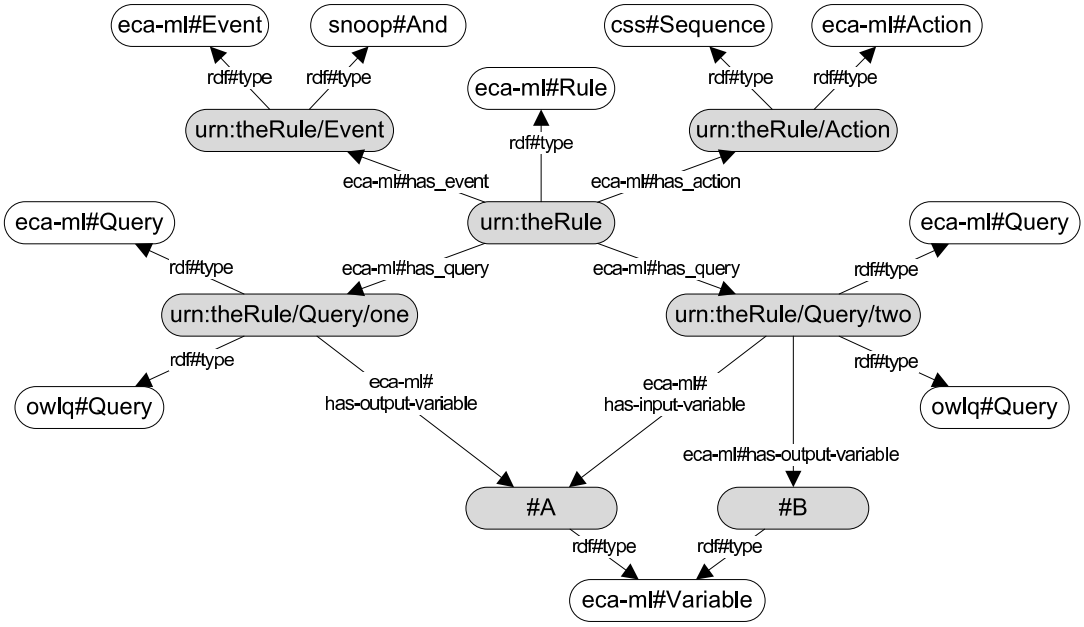


Figure 3.10: An Exemplary Rule Graph

specify a *ConjunctiveCondition* and leave the task of bringing its *AtomicConditions* into an executable order up to the rule evaluation service. The latter can achieve this by turning the *ConjunctiveCondition* into a *ListCondition* which is a subclass of the RDFS concept of an ordered *List*.

Finally, the ontology imports the variable usage concepts of the expressions ontology shown in Section 3.3.3. It is noteworthy that the properties *has-input-variable* and *has-result-variable* may be omitted by the user if there is at least one service for the respective language that offers the *analyze-variables* (cf. Figure 3.5) task.

The complete ECA-ML ontology can be found in [4] and at the project homepage [6].

Example 3.1 *Figure 3.10 shows an excerpt of the RDF graph of an exemplary ECA rule which is represented by the node* `urn:theRule`. *The event component of the rule (*`urn:theRule/Event`*) contains a composite* And *expression of the SNOOP language (cf. Section 3.5.2). Therefore, the node represents a language border as it is of the types* `eca-ml#Event` *and* `snoop#And` *at the same time. The event component of the rule will be shown in greater detail in Figure 3.11.*

In a similar way, the action component (`urn:theRule/Action`*) is given as a CCS sequence which also represents a composite expression in that language.*

Furthermore, the rule contains two queries (`urn:theRule/Query/one` *and* `/two`*), both of which are stated in OWLQ (cf. Section 3.5.1). The first one uses the variable* `#A` *positively (binds it to a value) while the second one uses* `#A` *as input and binds the variable* `#B`.

3.5 Exemplary Language Ontologies

To complete the description of ontologies that are used by the MARS framework, this section will now exemplarily show the query language OWLQ and the event algebra language SNOOP.

3.5.1 The OWLQ Ontology

The logic-based query language OWLQ (specified in [4]) provides an ontology that allows to describe queries including the use of join conditions by variables. Because of its logical semantics, OWLQ is very well suited to describe atomic events, queries and tests in ECA rules (which are based on logical variables).

An OWLQ *Query* defines the classes that are relevant to it by the use of common OWL notions. A definition can either define a new class (e.g. as an intersection of an existing class and an *owl#Restriction*), or use *owl#equivalentClass* to refer to an already existing one.

A query may declare variables ranging over instances of a class. It may also declare and possibly constrain variables ranging over the values (URIs or literals) of a property wrt. a value, a class membership, or another variable.

The OWLQ ontology therefore defines the following basic notions:

- a *Query* has the properties *definesClass*, *hasConstraint*, *useVariable*, and *resultVariable*.
- the notion *Class* serves for defining relevant classes. The definition can either define a new class by OWL means or refer to an existing *owl#Class*.
- the property *scopesVariable* declares variables ranging over instances of such a class.
- a *has*{*Mandatory*|*Optional*}*VariableDefinition* is associated with a *Variable* `v` and defines another variable ranging over the value(s) of an *rdf#Property* with respect to `v`.
- a *Constraint* specifies additional conditions that relate variables, e.g. a constraint may state that `v1` has to be *lessThanOrEqual* `v2`.
- an OWLQ fragment may further state its variable usage by the notions of *inputVariable* which must already be bound externally before its evaluation (e.g. if the query is embedded in a rule) and *resultVariable* which specifies that the respective variable occurs in the result projection.

For the embedding in MARS, there are special subclasses of *Query* that are analogous to the respective classes in ECA-ML, namely *EventSpec* and *Test*. This is shown in conjunction with SNOOP in example 3.2.

The complete OWLQ ontology can be found in [4] and at the MARS homepage [6].

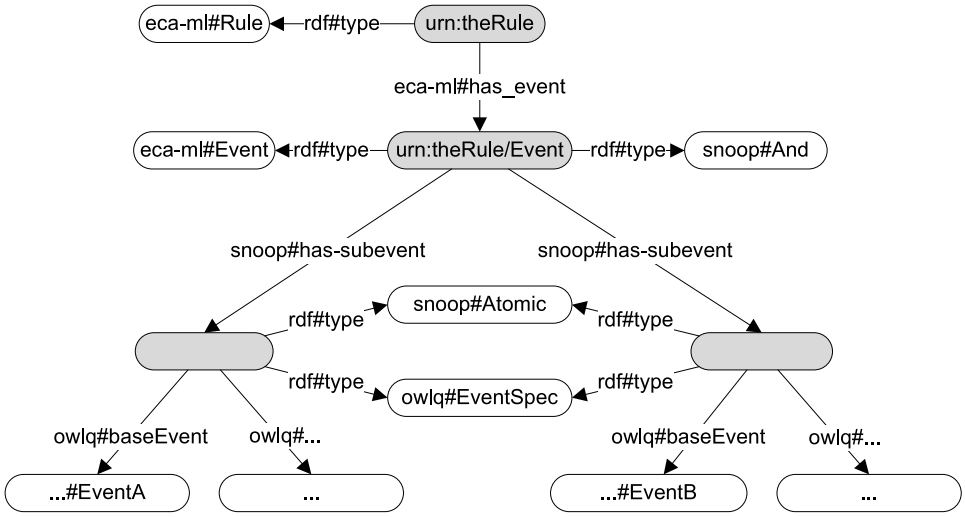


Figure 3.11: The Event Component of an Exemplary Rule Graph

3.5.2 The SNOOP Ontology

Since SNOOP [2] is an algebraic language, it imports notions from the respective sub-ontology (cf. Section 3.3.2) and the main classes represent its operators and give their arity. Furthermore, the SNOOP ontology defines the notion *Event* which is either an *AtomicEvent* or a *CompositeEvent* consisting of an *Operator* and an according number of subevents.

The operators defined by the SNOOP ontology include those concerning bag-events like *And*, *Or*, *Any*, and *MultiOccurrence* with the respective property *has-subevent*. Furthermore, *Any* and *MultiOccurrence* can be associated with the property *number-of-occurrences* to specify the multiple occurrence of an event.

Finally, a sequence of events can be described using the *Sequence* operator. Since nodes in the RDF model do not have an inherent ordering, the subevents can be modeled in two ways. Either two subevents (which again can be sequences) always are connected to the *Sequence* operator by the properties *first* and *second*, or a subevent may refer to its successor by *followedBy* property.

As with the previous ontologies, the complete SNOOP ontology can be found in [4] and at the project homepage [6].

Example 3.2 *Consider again the rule shown in Section 3.4.2. Figure 3.11 now shows the event component in greater detail.*

The root node of the event is a `snoop#And` *operator and therefore represents a composite expression of the SNOOP language. It has two subevents, both of which are* `Atomic` *in the SNOOP ontology. The actual specification of the subevents is then given in OWLQ by using the*

`owlq#EventSpec` *notion. The nodes representing the subevents therefore constitute a further language change.*

This chapter described in detail all necessary ontologies that are used by the MARS framework. The following chapter will now discuss how semantic ECA rules can actually be processed by the framework.

4 Evaluation of RDF Rules

The previous chapter introduced the MARS ontology in general and the ECA-ML ontology that models semantic ECA rules in particular. To actually process these rules in the MARS framework, there are two possible ways.

The first is to do the evaluation of the rule purely at the RDF level. While at first glance this option may seem to be preferable, it introduces several requirements the services in the MARS framework would have to meet. For one, all services would have to be able to deal with RDF input themselves which is currently only supported by the OWLQ engine.

Additionally, as the services would not be receiving the whole graph of the expression but rather only the URI of its root node, they were required to collect the actual nodes on their own. This directly leads to the problem that – at the RDF level – the rule graph does not have well-defined boundaries, i.e. it is not clear where the graph of a rule "ends" and where other fragments like domain definitions begin.

The second option to handle RDF rules is to map them to the XML level and register them at an according ECA engine (cf. Section 2.4). This way the preexisting environment with its (legacy) XML-enabled services can be left unchanged and still be incorporated into the "semantic world".

In this thesis, the second approach was chosen and implemented.

4.1 The Overall Conversion Procedure

For RDF rules to be mapped correctly to XML, several intermediate steps are necessary.

1. At first, the rule is processed by a mapping tool that constructs a temporary XML serialization (the so-called *striped* form) of the RDF graph according to respective DTD documents.

2. The striped XML document is then *stripped* of any redundant predicate elements according to the actual language DTD documents. While this already yields a valid rule marked up in ECA-ML, there is still one aspect, namely variables, that needs to be taken care of.

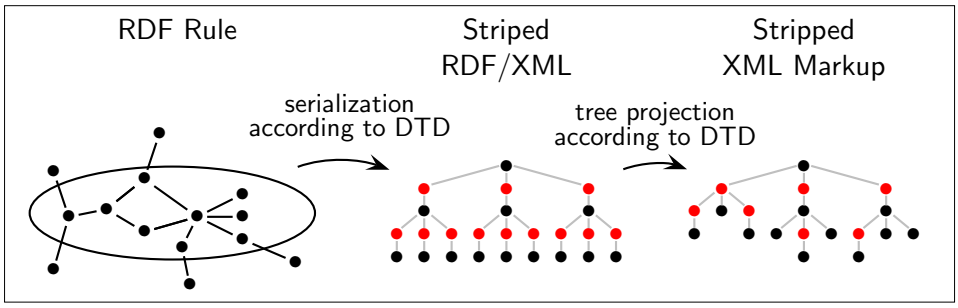


Figure 4.1: Obtaining XML Markup from RDF

3. At the RDF level, the rule components are not required to specify their use of variables. Instead, language processors offer a task that analyzes the component and extracts this information for further use. As the XML markup requires the variable usage to be given explicitly at the outer rule level, it has to be added to the XML rule.

4. When the variable usage is known, the query components of the rule need to be ordered since there was no order given at the RDF level. Therefore, the rule components are sorted according to a topological sort order which ensures a *safe* evaluation of the rule wrt. the use of variables.

5. Finally, the rule is registered in valid ECA-ML markup at an ECA engine for further evaluation.

The first two steps are explained in detail in the following section. More information about steps 3 and 4 can be found in Section 4.5.

4.2 DTD-Driven Mapping of RDF to XML

As stated in the last section, the first step of the rule conversion process is to map the RDF graph of the rule to the XML level, serializing it to an intermediate striped RDF/XML format and stripping it from redundant elements afterwards, as shown in Figure 4.1.

Since there is no way of knowing the exact boundaries of the rule fragment inside the RDF graph, the serialization has to be controlled by traversing the graph according to a predefined DTD which specifies the structure of the striped format.

Given the name of the root element, the tool *rdf2xml* (which is part of the MARS distribution) uses the DTD to learn about the currently allowed child elements. For those elements that also appear as a node in the RDF graph, it constructs an XML element and adds it to the resulting document tree at the appropriate position. Using the newly created element as the starting

point, this process is then repeated in a depth-first manner. When there are no more possible child elements left at the current level, the process is resumed at the next higher level.

When finished, the result is a striped RDF/XML representation of the relevant part of the RDF graph. While this already constitutes a usable XML mapping of the graph, it still contains unnecessary intermediate elements (like `<eca:has-event/>`, `<eca:has-query/>` and `<eca:has-action/>`) and even more important, it disregards the attribute definitions of the destination language by storing attribute values as subelements.

In a second step, *rdf2xml* therefore applies a tree projection to the formerly generated tree by which these intermediate elements are removed, ID/IDREF references are created and subelements are turned into attributes of the destination language where applicable. This finally results in the valid XML document of the destination language that is needed to proceed with the conversion process.

Although *rdf2xml* is capable of handling RDF graphs that contain nodes from multiple namespaces by dynamically loading the necessary DTD documents at runtime, it is not possible to use it to convert complete ECA rules directly as shown in the next section.

4.3 Discussion of Mapping Approaches

4.3.1 Mapping at the RDF Level

The obvious first approach to mapping the RDF graph of an ECA rule to the XML level is to directly invoke the *rdf2xml* tool discussed in the previous section. While this has the advantage of working purely on the conceptual RDF level, independent of any particular serialization format like RDF/XML or N3, there is a chance that the mapping procedure will fail, depending on which languages were used to define the rule components.

On the upper level of a rule, the ECA-ML nodes represent a tree-like structure in which there is no path between the individual rule components. While two components may refer to a single node (e.g. a variable as shown in Figure 3.10), the node itself does not have any predicates that only apply to one of the referring components.

The language OWLQ (cf. Section 3.5.1) on the other hand represents a graph by itself. Again, two queries may refer to the same variable, but in contrast to ECA-ML, an OWLQ variable may be the basis for further variable definitions. Thus, when embedded into an ECA rule, it is not generally decidable to which query these definitions belong.

Example 4.1 *Consider again the rule graph shown in Figure 3.10. The first query component of the rule binds the variable* `#A` *which is then used by the second query component to determine*

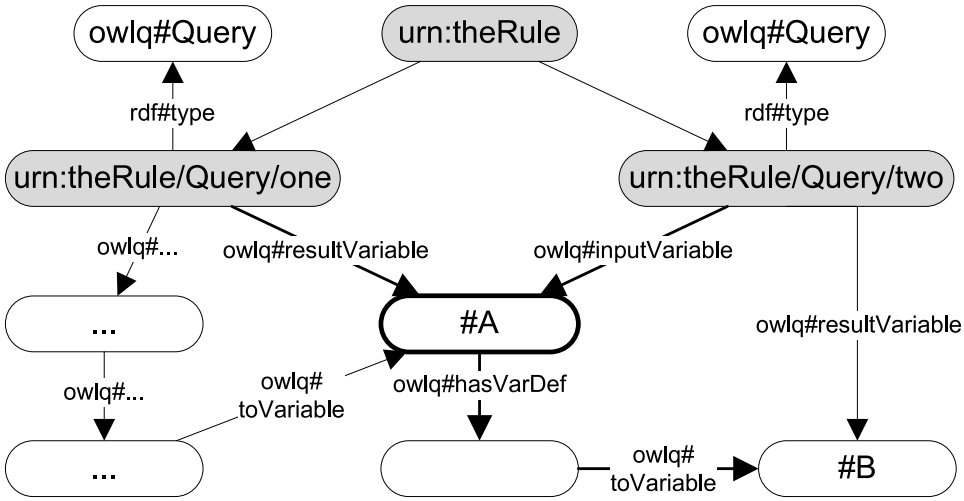


Figure 4.2: An RDF Graph of Two Exemplary OWLQ Queries

the value of variable `#B`. *Figure 4.2 shows a more detailed graph of the two query components, stated as OWLQ queries.*

Although the variable `#B` *is only bound by the second query, there is a path that connects the first query with its definition. This causes* rdf2xml *to falsely convert the two queries as shown in Figure 4.3, where the binding of the variable* `#B` *is displaced and already happens in the first query (when the respective data is not yet available).*

4.3.2 Mapping at the RDF/XML Level

Since the mapping at the RDF level does not work as intended, another approach tries to utilize the RDF serialization format RDF/XML to map the RDF graph to XML. In contrast to the former approach, this method purely works on the syntactical level which allows it to cleanly separate between the distinct rule components, eliminating the problem with positive and negative variable occurrences.

However, this approach lacks the complete semantics of RDF, making it impossible to access additional information stored in the RDF model of the respective language. This is necessary whenever the RDF model of the language defines additional (sub-)classes that do not appear as elements in the DTD of the language.

Furthermore, working without the semantics of RDF is making it difficult to ensure a correct tag hierarchy regarding nodes that contain multiple `rdf:type` definitions.

```
<eca:Rule>
  <eca:Query>
    <owlq:Query>
      <owlq:resultVariable rdf:resource="#A" />
      <owlq:useVariable>
        <owlq:Variable rdf:about="#A">
          <owlq:hasVariableDefinition>
            <owlq:VariableDefinition>
              <owlq:onProperty rdf:resource="urn:..." />
              <owlq:toVariable rdf:resource="#B" />
            </owlq:VariableDefinition>
          </owlq:hasVariableDefinition>
        </owlq:Variable>
      </owlq:useVariable>
      <owlq:definesClass>
        ...
      </owlq:definesClass>
    </owlq:Query>
  </eca:Query>

  <eca:Query>
    <owlq:resultVariable rdf:resource="#B" />
    <owlq:inputVariable rdf:resource="#A">
      ...
    </owlq:inputVariable>
    ...
  </eca:Query>
</eca:Rule>
```

Figure 4.3: The Falsely Converted Queries

4.3.3 Combining the RDF and RDF/XML Level

The first two isolated mapping approaches – working either on the semantic RDF level or the syntactic RDF/XML level – both failed due to the necessity to use the particular characteristics of the other level. To overcome the observed shortcomings and ensure a correct RDF to XML mapping, both levels therefore have to be used in combination.

The basic idea is to recursively traverse the RDF/XML tree and split it up, so that every subtree belongs to exactly one `MARS#Language`. These subtrees are then individually interpreted as an RDF model and mapped separately by *rdf2xml*. This way, variable occurrences cannot interfere with the boundaries of the rule components. Afterwards, the converted subtrees are relinked at their former positions in the tree.

On a more theoretical level, the RDF/XML tree is split up using the following algorithm. Starting with the root element of the rule and the ECA namespace as the current namespace:

- The current element represents a resource at the RDF level. It is of the form `ns:R`, with `ns` being the namespace and `R` being the name of the element. All subelements must now be either `rdf:type` definitions or predicate elements of the form `ns:P` and `ns':P` respectively.

- If the current element represents a language change, i.e. if the current element either belongs to a different namespace than its parent or contains an `rdf:type` definition indicating that it belongs to an additional namespace on the RDF level and this namespace represents another `MARS#Language`:
  1. Create a deep copy that contains the element itself, any `rdf:type` definitions and all subelements that do not belong to the namespace `ns` or belong to the namespace `ns'` and are not defined by the DTD of the language.
  2. Store the copy with a reference to the URI of the original element (using the value of the `rdf:about` attribute) and process it separately using the new `MARS#Language` as the new current namespace.
  3. In the original tree, keep all type definitions, predicate subelements of the namespace `ns` and those of the namespace `ns'` that are defined by the DTD of the current language and remove all other subelements.
- Traverse all subelements of the current element that are not `rdf:type` definitions. Since the current element represents a resource at the RDF level, its subelements must represent predicates. For each of the predicate subelements, take their child (which must be a resource element again) and apply the algorithm.

The algorithm obviously does not create completely disjoint subtrees, as the root of each subtree is usually an element of the old namespace, containing an `rdf:type` definition with the new namespace. However, this does not represent a problem since each subtree is interpreted as an RDF model later on and *rdf2xml* is given the namespace it should use to map the graph to XML. Since the element of the old namespace is not defined by the DTD of the new language, it is dropped silently.

When the RDF/XML subtrees have been mapped by *rdf2xml* the complete tree is reassembled according to the previously stored references to the `rdf:about` attributes. Afterwards, the rule components are analyzed wrt. their use of variables and reordered to ensure a safe evaluation. The conversion process is shown as a whole in the next section, the variable analysis and reordering of the rule components is shown in Section 4.5.

4.4 An Exemplary Conversion

As described in the previous section, the first step of the conversion process is to recursively traverse the RDF/XML tree and split it along the language borders.

```
<rdf:RDF>
  <eca:Rule rdf:about="urn:theRule">
    <eca:has-event>
      <eca:Event rdf:about="urn:theRule/Event">
        <rdf:type rdf:resource="urn:snoopy#And"/>
        <snoop:has-subevent>
          <snoop:Atomic rdf:about="urn:theRule/Event/one">
            <rdf:type rdf:resource="urn:owlq#EventSpec"/>
            <owlq:baseEvent rdf:resource="urn:events#EventA"/>
            <owlq:...>
            </owlq:...>
          </snoop:Atomic>
        </snoop:has-subevent>
        <snoop:has-subevent>
          <snoop:Atomic rdf:about="urn:theRule/Event/two">
            <rdf:type rdf:resource="urn:owlq#EventSpec"/>
            <owlq:baseEvent rdf:resource="urn:events#EventB"/>
            <owlq:...>
            </owlq:...>
          </snoop:Atomic>
        </snoop:has-subevent>
      </eca:Event>
    </eca:has-event>
    ...
  </eca:Rule>
</rdf:RDF>
```

Figure 4.4: The RDF/XML Serialization of an Event Component

A language border is reached when the current RDF node is of exactly two types from different namespaces that are a subclass of a `MARS#Language`, e.g. `ECA#Event` and `SNOOP#Sequence`, or `ECA#Query` and `OWLQ#Query`.

When a language border is reached, the respective subtree is disconnected from its parent and processed separately. Its root element in the original tree is replaced by a copy that only contains the identifying URI and the previously existing RDF type information.

Example 4.2 *Consider again the event component shown in Example 3.2. Figure 4.4 shows the RDF/XML serialization of the component, consisting of the two atomic events* `EventA` *and* `EventB` *combined by SNOOP's* `And` *operator.*

In the current example, nodes belonging to two languages are "urn:theRule/Event"*, which is an* `ECA-ML#Event` *as well as a* `SNOOP#And`*, and the nodes* "urn:theRule/Event/one" *and* "urn:theRule/Event/two"*, both of which are of the types* `SNOOP#Atomic` *and* `OWLQ#EventSpec`*. The separated parts of the exemplary event component are shown in Figure 4.5.*

When all subtrees have been separated, they are individually imported into an RDF model and then converted by the *rdf2xml* tool described in Section 4.2.

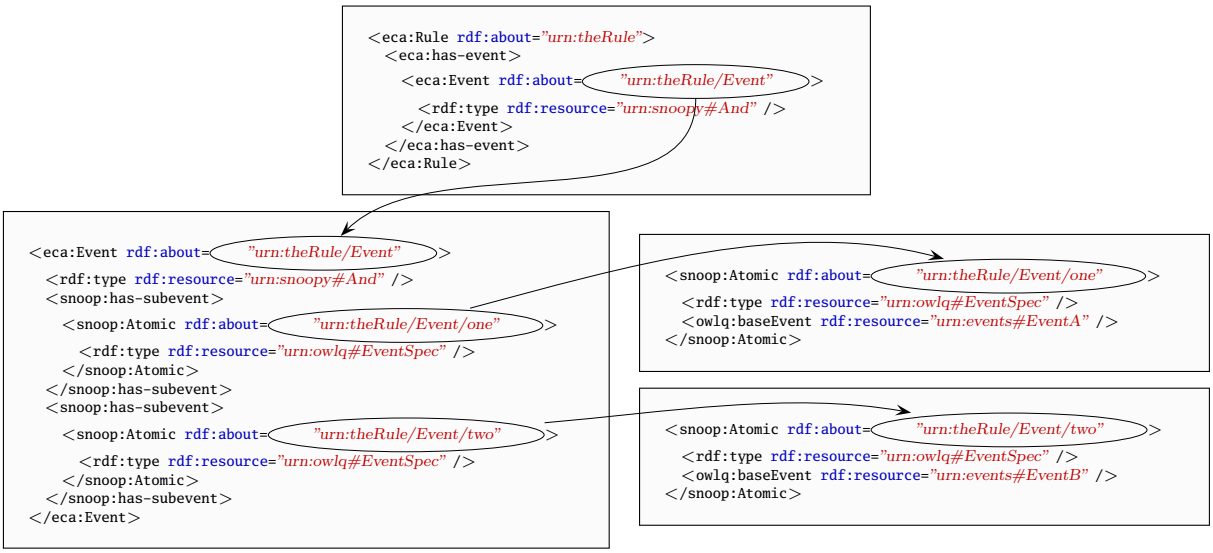


Figure 4.5: The Separated Parts of the Event Component

Starting from a given root node, *rdf2xml* traverses the RDF graph and constructs an intermediary XML fragment containing the relevant triples in a so-called ”striped” form as defined by a respective DTD. In a subsequent step, the XML fragment is then stripped from the redundant predicate elements using the according ”stripped” DTD.

The result of this step is wrapped inside a `<stop/>` element to indicate the language change. Afterwards, it is relinked with the striped fragment of its former parent element which then is stripped as well, omitting the subtrees in `<stop/>` elements.

Example 4.3 *The previously described steps are illustrated together in Figure 4.6. At the innermost level of the event component it shows the two (fully converted)* `OWLQ#EventSpec` *elements which are wrapped inside a* `<stop/>` *element and relinked with the intermediate striped form of the* `SNOOP#And` *tree.*

As the figure shows, this outer tree contains two `<snoop:has-subevent/>` *elements that are not part of the SNOOP language itself but belong to SNOOP's striped DTD. During the following stripping process they are removed along with the* `<stop/>` *element.*

Again, the resulting tree is relinked with its former parent, in this case the ECA rule itself, and the outer tree is stripped from its redundant predicate elements (`<eca:has-event/>`*).*

Finally, Figure 4.7 shows the completely converted event component that solely contains elements of the respective languages themselves.

Figure 4.6: The Converted Parts of the Event Component, Stripped and Reattached to the Respective Outer Element Nodes, Wrapped Inside a `<stop/>` Element

4.5 Reordering Query Components

After the rule has been converted to XML, its query components have to be ordered to ensure a *safe* evaluation order. Safe in this case means that every negative variable occurrence must be preceded by a positive one, guaranteeing that the variable has been bound to a value before it is used in another query.

```
<eca:Rule>
  <eca:Event>
    <snoop:And>
      <snoop:Atomic>
        <owlq:EventSpec rdf:about="urn:theRule/Event/two">
          <owlq:baseEvent rdf:resource="urn:events#EventB" />
          <owlq:usesClass>
            <owlq:Class rdf:about="urn:events#EventB" />
          </owlq:usesClass>
        </owlq:EventSpec>
      </snoop:Atomic>
      <snoop:Atomic>
        <owlq:EventSpec rdf:about="urn:theRule/Event/one">
          <owlq:baseEvent rdf:resource="urn:events#EventA" />
          <owlq:usesClass>
            <owlq:Class rdf:about="urn:events#EventA" />
          </owlq:usesClass>
        </owlq:EventSpec>
      </snoop:Atomic>
    </snoop:And>
  </eca:Event>
</eac:Rule>
```

Figure 4.7: The Completely Converted Event Component

4.5.1 Analyzing Component Variable Usage

Like the rest of the MARS framework, the variable usage analysis of rule components is designed to function decentralized, allowing completely new languages to be integrated without changing any of the core framework services.

Therefore, every language that is registered at the languages and services registry (LSR)[1] should specify at least one service that provides a task to analyze the variable usage of a fragment of that language. Any other service can then execute the respective task using the Generic Request Handler (GRH) which will take care of all details.

After the task has been executed for all rule components, the results are added to the mapped XML representation of the rule.

Example 4.4 *Consider again the rule shown in the previous examples with its query components written in OWLQ. When the rule conversion process is finished, the components of the rule are analyzed wrt. their usage of variables. Figure 4.8 shows the message that is sent to the* `MARS#analyze-variables` *task of the OWLQ engine to analyze the second query of the rule.*

The OWLQ engine then analyzes the query and returns its variable usage, marked up as shown in Figure 4.9. In this case, the query takes the variable `#A` *as input and binds the variable* `#B`.

[1] For a description of the LSR see Sections 2.4 and 3.2.3

```
<request>
    <owlq:Query rdf:about="urn:theRule/Query/two">
      <owlq:inputVariable rdf:resource="#A"/>
      <owlq:resultVariable rdf:resource="#B"/>
      ...
    </owlq:Query>
</request>
```

Figure 4.8: The Request to Analyze the Variable Usage of the Second Query

```
<rdf:RDF>
  <rdf:Description rdf:about="urn:theRule/Query/two">
    <mars:has-negative-variable rdf:resource="#A"/>
  </rdf:Description>
  <rdf:Description rdf:about="urn:theRule/Query/two">
    <mars:has-positive-variable rdf:resource="#B"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 4.9: The Response from the OWLQ Service

4.5.2 Safe Topological Order

When the variable usage of every rule component is known, the query components are ready to be reordered. Although there are a number of possible optimizations, the reordering algorithm is kept very simple for the time being.

Starting with the list of variables that have been bound in the event component, the following loop is executed: while there are unordered query components left, traverse the list of all remaining query components. If the current query component uses only input variables that are known to be safe, append it to the list of ordered query components and add its result variables to the list. If no suitable query component could be found, end with an error, otherwise continue traversing the remaining unordered query components.

Example 4.5 *Consider Figure 4.10(a) which shows a slightly modified excerpt of the rule used in the previous examples. The first query now depends on the variable* `Event` *which is bound by the event component and there is a third query that uses* `A` *and* `B` *as input and returns them along with variable* `C`.

The evaluation of this rule in the given order would obviously fail since the first query already uses the variables `A` *and* `B` *which are not yet bound.*

The event component of the example binds a single variable with the name `Event` *which is why only the second query component can be safely evaluated at this point. Since this in turn binds the variable* `A`*, the list of safe variables now consists of* `Event` *and* `A`.

```
<eca:Rule>
  ...
  <eca:Query>
1   <owlq:Query>
      <eca:uses-variable eca:name="A"/>
      <eca:uses-variable eca:name="B"/>
      <owlq:resultVariable rdf:resource="#A"/>
      <owlq:resultVariable rdf:resource="#B"/>
      <owlq:resultVariable rdf:resource="#C"/>
      ...
    </owlq:Query>
  </eca:Query>
  <eca:Query>
2   <owlq:Query>
      <eca:uses-variable eca:name="Event"/>
      <owlq:resultVariable rdf:resource="#A"/>
      ...
    </owlq:Query>
  </eca:Query>
  <eca:Query>
3   <owlq:Query>
      <eca:uses-variable eca:name="A"/>
      <owlq:resultVariable rdf:resource="#A"/>
      <owlq:resultVariable rdf:resource="#B"/>
      ...
    </owlq:Query>
  </eca:Query>
  ...
</eca:Rule>
```

(a) Excerpt of a Rule After the Mapping to XML

```
<eca:Rule>
  ...
  <eca:Query>
2   <owlq:Query>
      <eca:uses-variable eca:name="Event"/>
      <owlq:resultVariable rdf:resource="#A"/>
      ...
    </owlq:Query>
  </eca:Query>
  <eca:Query>
3   <owlq:Query>
      <eca:uses-variable eca:name="A"/>
      <owlq:resultVariable rdf:resource="#A"/>
      <owlq:resultVariable rdf:resource="#B"/>
      ...
    </owlq:Query>
  </eca:Query>
  <eca:Query>
1   <owlq:Query>
      <eca:uses-variable eca:name="A"/>
      <eca:uses-variable eca:name="B"/>
      <owlq:resultVariable rdf:resource="#A"/>
      <owlq:resultVariable rdf:resource="#B"/>
      <owlq:resultVariable rdf:resource="#C"/>
      ...
    </owlq:Query>
  </eca:Query>
  ...
</eca:Rule>
```

(b) The Query Components Reordered

Figure 4.10: Reordering of the Query Components

Looking again at the remaining query components, the first one still cannot be evaluated due to the missing binding of variable `B`*, while the preconditions for the third query are met. Thus, after the second pass, the list of safe variables consists of* `Event`*,* `A` *and* `B`*.*

At this point, all input variables of the first query are bound and it now is safe to be evaluated. The final rule with its reordered query components is shown in Figure 4.10(b).

This chapter showed the theoretical background for handling RDF rules in the MARS framework by mapping them to XML. The next chapter will now give a short overview of the prototypical implementation of such a mapping service.

5 Implementation

This chapter deals with the implementation details of handling RDF rules inside the MARS framework. As stated earlier, RDF rules are currently not evaluated directly at the semantic level but converted to XML and then registered at an appropriate ECA engine.

In order to integrate this feature without adding a new service type to the framework, the conversion service implements the interface of a typical ECA engine. By hiding the specific details about the handling of RDF rules from the client, it will not have to be changed when the rules are evaluated directly at the semantic level in the next stage of development.

5.1 Technologies

Like all other components of the MARS framework, the conversion service developed in this thesis was implemented using Sun's Java programming language [15]. Since the services themselves only use plain XML messages that are communicated via HTTP to perform their tasks, any other net-capable language would have been suitable as well. The advantage of using Java in this case was the already existing common infrastructure[1] that has been developed along with the other services.

The conversion service uses the Semantic Web framework Jena [5] which provides APIs for dealing with RDF and OWL ontologies as well as reasoning about them. Furthermore, it provides a way to evaluate SPARQL queries against the ontology models. Jena is also a prerequisite of the *rdf2xml* tool described in the next section which makes its use mandatory.

5.2 RDF2XML

The tool *rdf2xml* was developed in [16] and is used by the conversion service to separately map the subtrees of an RDF rule to XML. While it originally consisted of only a single class that implemented the whole mapping process, it was split up during the development of the conversion service. The task of building an intermediate XML document in a striped format is now performed by the class `RDFToXML` while the stripping of these documents (including striped documents not generated by *rdf2xml*) is done by the class `DTDStripper`.

[1] Messaging, task execution (GRH), languages and services registry (LSR), etc.

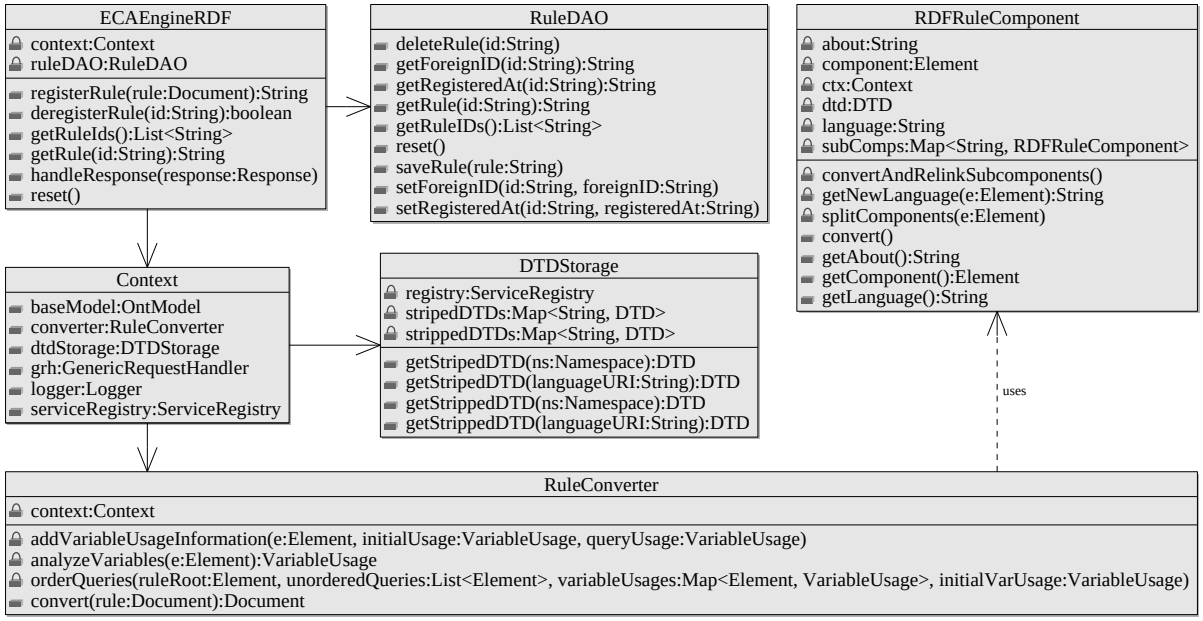


Figure 5.1: Class Diagram for the Rule Conversion Service

5.3 Rule Conversion Service

5.3.1 Architecture

The architecture of the RDF rule conversion service is relatively simple as shown in Figure 5.1. As stated earlier, the central class implements the interface of a typical ECA engine, hence the name `ECAEngineRDF`.

Nearly all classes keep a reference to the current `Context` object that in turn contains references to all other relevant objects needed during the conversion process such as the `GRH`, the `ServiceRegistry` and the `Logger` of the MARS framework.

Most importantly, it also contains a reference to the `RuleConverter` that controls the conversion process in its whole. The only public method of this class, `convert`, takes an RDF/XML representation of an ECA rule as input and returns the completely converted and reordered XML mapping of the rule.

5.3.2 Rule Conversion

As described in Section 4.1, the first step of the conversion process is to recursively split the rule along its language borders. A single such subtree is represented by an instance of

the class `RDFRuleComponent` which in addition to the component itself keeps references to the subcomponents it encloses and their respective `rdf:about` values. When an instance of `RDFRuleComponent` is created, it immediately traverses its RDF/XML tree and builds up the above mentioned structure.

After the rule has been split up, the `RuleConverter` calls the method `convert` on the root instance of `RDFRuleComponent`. This method then generates an RDF model of the subtree and utilizes *rdf2xml* to construct its intermediate striped RDF/XML representation. It then uses `convertAndRelinkSubcomponents` to recursively call `convert` on all subcomponents[2]. Afterwards, the subcomponents are put back into the tree according to their `rdf:about` values. To prevent the following stripping process from entering the (already stripped) subtrees, they are enclosed inside a `<stop/>` element as stated in Section 4.4. Finally, *rdf2xml* is used to strip the intermediate representation. This removes unnecessary predicate elements as well as the `<stop/>` elements and `rdf:about` attributes.

5.3.3 Query Reordering

When the rule has been fully converted, the `RuleConverter` invokes the `GRH` for each component of the rule to analyze their use of variables. The results are then incorporated into the XML rule so that the ECA engine will know which variables to pass along with the queries when the rule is evaluated.

In a further step the `RuleConverter` calls its private method `orderQueries` that implements the algorithm described in Section 4.5.2 to bring the queries into a safe topological order.

At last, the converted and reordered rule is registered at an ECA engine (via the `GRH`) which takes care of the actual evaluation of the rule. The rule conversion service itself only stores the original RDF rule (for possible future reasoning about it) along with information about the ECA engine and the ID of the rule.

[2]Note that this already returns the stripped representation of the component.

6 Conclusion

This thesis dealt with Event-Condition-Action rules at the semantic level and provided a practicable way for handling them inside the MARS framework.

Generally, there are two possible ways for handling semantic ECA rules:

1. Mapping the rules to the XML level and register them at an existing ECA engine.
2. Providing a service that evaluates the rules directly at the semantic level.

While the second option offers more possibilities wrt. the reasoning about the rules themselves and the behavior of Semantic Web nodes as a whole, it also requires a very detailed description of the participating domains and languages. Furthermore, the complete service landscape would have to be fully capable of dealing with RDF fragments that are distributed throughout the Semantic Web.

Therefore, this thesis implemented the first and – for the time being – more practical option. During the development of a prototypical mapping service for ECA rules, several problems were encountered and solved, mainly concerning the implications of the RDF model of the rules being a labeled graph that allows multiple edges to connect its nodes in contrast to the much simpler tree structure of a rule given as an XML document.

Areas of further work include the following:

Evaluation of RDF Rules: The long term goal is to lift all framework services to the semantic level and provide the necessary domain ontologies in order to being able to evaluate RDF rules directly and without mapping them to XML.

Inter-Rule Reasoning: By lifting ECA rules to the semantic level it becomes possible to reason about the behavior of Semantic Web nodes. Especially interesting is the reasoning about the interferences of rules, i.e. rules that trigger other rules, which can provide helpful information regarding correctness proofs and termination analysis.

A General Ontology of Semantics: This allows the ECA rules to be even more abstract, defining only their semantics and leaving the task of translating the abstract descriptions to concrete languages to the framework.

Inter-Rule Optimization: By defining ECA rules as RDF graphs, every component of a rule becomes an identifiable resource of the Semantic Web. Therefore, an individual component may be used by several rules simultaneously. In combination with the possibility of preregistering components (e.g. queries) at their respective services this allows for the optimized evaluation of multiple rules and multiple rule instances respectively.

In conclusion, the work of this thesis represents an intermediate step towards a fully ”semantic” implementation of the MARS framework.

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] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB*, pages 606–617, 1994.

[3] The Friend of a Friend (FOAF) project. `http://www.foaf-project.org/`.

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

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

[6] The MARS project homepage. `http://www.dbis.informatik.uni-goettingen.de/MARS/`.

[7] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the semantic web: Dealing with language heterogeneity. In *Rule Markup Languages (RuleML)*, number 3791 in LNCS, pages 30–44. Springer, 2005.

[8] Robin Milner. *A calculus of communicating systems*, volume 158 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1983.

[9] Notation 3 - A readable RDF syntax. `http://www.w3.org/DesignIssues/Notation3.html`, 2006.

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

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

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

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

[14] Daniel Schubert. Development of a Prototypical Event-Condition-Action Engine for the Semantic Web. Bachelor Thesis, Universität Göttingen, 2006.

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

[16] Heiko Vollmann. DTD-Driven Export of OWL Data to XML. Bachelor Thesis, Universität Göttingen, 2008.

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

[18] XML Path Language (XPath) version 1.0: 1999. `http://www.w3.org/TR/xpath`, 1999.

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