Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2006-03

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Development of a Prototypical Event-Condition-Action Engine for the Semantic Web

Daniel Schubert

am Lehrstuhl für

Datenbanken & Informationssysteme

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

03. Februar 2006

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

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

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

Göttingen, den 03. Februar 2006

Bachelor Thesis

Development of a Prototypical Event-Condition-Action Engine for the Semantic Web

Daniel Schubert

February 3, 2006

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

Abstract

The Semantic Web will consist of a large number of autonomously evolving information systems. In contrast to the Web of today, these information systems will not only be able to answer query requests but they will also be capable of reacting to events, such as database updates at remote systems.

To define the reactive behavior of an information system, the concept of Event-Condition-Action (ECA) rules can be applied. An ECA rule specifies an event, an optional condition and in case the condition is satisfied, the indented action that should be executed.

In this thesis, a prototypical evaluation service for the distributed evaluation of ECA rules is developed. Since it adheres strictly to current Web standards, it can easily be integrated into existing information systems and turn them into active participants of the Semantic Web.

Contents

List of Figures

1 Introduction

While today's World Wide Web is evolving into the Semantic Web, the provided information is becoming more thoroughly structured and computer understandable. This process leads to a new understanding of the Web as a whole.

Instead of solely being *data sources*, the nodes inside the Semantic Web represent autonomously evolving *information systems*. They are not only able to provide or update their own data, but may also request other information systems to do so with their local databases. Furthermore, these information systems can react to events (usually data updates at remote nodes) that may happen anywhere inside the Web. The latter implies that the Semantic Web must provide a way of propagating event occurrences among its nodes.

When a node becomes aware that an event occurred (either by being notified or by actively querying other nodes) it may then react to the same by first gathering additional information and checking conditions before finally, taking the appropriate actions. The behavior of a node may suitably be formalized by *Event-Condition-Action (ECA)* rules that separate these different aspects of reactivity.

Based on the framework for reactivity and evolution that was presented in [10], this thesis deals with the development of an evaluation service for ECA rules — the *ECA Engine*. Taking the existing architecture outline of the framework as a basis, it furthermore specifies the details of cooperation and communication between the different components.

Additionally, an exemplary information system is developed in order to provide a reasonable testing environment. This information system consists of several query processors and represents a fictional car-rental company.

The structure of this thesis is as follows. In the next chapter, a general outline of ECA rules and the ECA framework is given which is followed by the presentation of a markup language for ECA rules. How variables can be used for communication between the different components of a rule is shown in Chapter 4 along with a detailed description of the communication between the individual framework components. Chapter 5 deals with the process of rule evaluation and explains the abstract architecture of the ECA engine whose implementation is described afterwards. Finally, the thesis is concluded and the next steps in the process of implementing the complete ECA framework are identified.

2 The ECA Framework

The framework that forms the basis of this thesis was presented in [10] and provides an implementation of reactive behavior in the Semantic Web by the use of Event-Condition-Action rules. These rules separate the different concerns of triggering rule execution, gathering additional information, deciding if something should be done and defining what should be done.

After a short explanation of the basic terms in the next section, the detailed aspects of ECA-style rules in a Semantic Web environment are discussed in Section 2.2. Afterwards the general framework architecture is illustrated.

2.1 Semantic Web

Documents in the World Wide Web are usually marked up in HTML to provide a specific visual layout of the contained information. While this is sufficient for information exchange with human readers, there is no reliable way for a computer to process such data.

The Semantic Web aims at supplying a framework to add a well-defined meaning to the information inside these documents. This allows for the automated processing of the data including a machine-based reasoning. To achieve this goal the Semantic Web utilizes a stack of recommendations by the W3C [17] that will now be explained in short[1].

XML. The Extensible Markup Language defines a generic, text-based way of marking up semistructured data by the use of element tags and attributes. It is extremely flexible in that it not imposes any restrictions wrt. the naming and the semantics of these tags. See [18] for more details about XML.

XML Schema. The language XML Schema provides a way to formally restrict the structure of an XML document and defines the names and possible values of its elements and attributes. Thus, using XML Schema allows for the adaption of the general concept of XML to a specific application domain. More information can be found at [19].

[1]Note that XML Schema, RDF, RDF Schema and OWL also have XML representations.

RDF. The Resource Description Framework serves as a meta-data model for the description of concepts. It uses triples to associate two resources (a subject and an object) with a predicate. A resource may be anything that can be identified by a URI[2]. See [12] for more details.

RDF Schema. Similar to XML Schema and XML, RDF Schema specifies the "vocabulary" of an RDF document and transfers the general concept behind RDF to a specific application domain. Furthermore, it allows for the specification of properties and concepts and relations between different concepts. See [13] for more information about RDF Schema.

OWL. The Web Ontology Language is located at the same level as RDF Schema and serves a similar purpose. The main intention behind its development was the need for a much more expressive language than RDF Schema. Thus, it extends RDF Schema by adding more vocabulary to describe properties and relations of concepts more differentiated. OWL is described in detail in [11].

2.2 ECA Rules

2.2.1 Rule Components

Rules that follow the ECA paradigm are in general divided into several components, namely the *event component*, the *condition component* and the *action component*.

The *event component* specifies the event (or the sequence of events) that triggers the execution of the rule. In the Semantic Web, an event is something that can occur at any node and must be propagated and detected separately (in contrast to local databases that represent a closed world).

When the execution of a rule is triggered, the *condition component* can gather additional knowledge and decide if the following actions should be executed. To cleanly separate these tasks it can be split up into a *query part* and a *test part*.

The *action component* finally defines what actions are to be taken. It can use all previously collected information but should not collect any information itself.

The semantics of ECA rule execution can be summarized as

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

Example 2.1 *An example of the financial sector might be the following. Whenever a banking account is debited (event), check the new balance and the credit line (additional knowledge). If*

[2]Uniform Resource Identifier.

the liabilities exceed the credit line (condition), send a mail to the customer and ask him to make an appointment (action).

2.2.2 Component Languages

The ECA framework makes it possible to use arbitrary languages inside each of the rule components to support greater interoperability of the involved nodes. In general, every rule uses an event language, one or more query languages, a test language, and one or more action languages.

Event Languages. Languages used inside the event component must be able to describe the event in a way that allows for its detection.

In case of *atomic events* (that are given as XML fragments), languages can be employed that are able to analyze the structure and content of an XML document. A possible language for the event part is XPath [20] whose expressions can address parts of an XML document and constrain the result to match specified conditions.

Composite events represent expressions over several atomic events. To describe them, the event language must provide composers that define the relation between the atomic events (e.g. AND, OR and AND THEN). A language that is able to describe composite events is SNOOP [4].

Query Languages. There are two main types of languages that can be used inside a query component, namely *functional* and *logical languages.*

Functional query languages provide a set of functions that act upon data and return a set of data items (in case of a database query) or a data fragment (in case of an XML query). To work with the result in a subsequent component, it has to be bound to a variable at the rule level as described in the next section.

Besides the previously mentioned language XPath, the language XQuery [21] represents such a functional language. XQuery is described in more detail in Section 5.4.

In logical query languages, variables are bound by matching *free variables* that act as a kind of placeholder. As these variable bindings represent the result of the query, they can directly be used in a subsequent component. An example for a logical language is Datalog [8] that is able to query relational databases.

Example 2.2 *In logical languages like Datalog, the database to be queried is represented by a set of* predicates *(facts) and* rules*.*

Predicates *are known to be valid and consist of a* head *and a number of* arguments*, for example:*

```
father('John', 'Tim').
mother('Mona', 'Tim').
father('Tim', 'Christine').
father('Tim', 'Lisa').
mother('Nicole', 'Christine').
mother('Nicole', 'Lisa').
```

In contrast, rules *allow for the deduction of new facts. A rule consists of a* head *and a* body *of the form* head :- body*. When the* body *of a rule can be proven, this directly implies the validity of the* head*.*

The two following rules use free variables *to specify that X is a grandfather of Y when he is the father of Z who in turn is a parent (mother or father) of Y:*

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
grandfather(X, Y) :- father(X, Z), mother(Z, Y).
```

It is then possible to query the database:

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

This would bind the variables X, Y and Z in the body to the values

$$X \rightarrow \text{“John”},\ Z \rightarrow \text{“Tim”},\ Y \rightarrow \text{“Christine” and}$$
$$X \rightarrow \text{“John”},\ Z \rightarrow \text{“Tim”},\ Y \rightarrow \text{“Lisa”}$$

which are then used in the head to deduce the facts

```
grandfather('John', 'Christine').
grandfather('John', 'Lisa').
```

Thus, the result of the query would be:

$$X \rightarrow \text{“John”},\ Y \rightarrow \text{“Christine” and}$$
$$X \rightarrow \text{“John”},\ Y \rightarrow \text{“Lisa”}$$

Test Languages. An expression of a language that is used in the test component must result in a logical truth value (i.e. true or false). Results of expressions of functional languages can be interpreted as such. An empty result or a literal “false” indicates a negative result, anything else a positive one. Thus, XPath and XQuery can also be used inside the test component.

Action Languages. In an action component, arbitrary languages can be used that are able to describe the intended action(s). Besides the “classical” programming languages like Java, it is also possible to call Web Services using SOAP or simple HTTP requests.

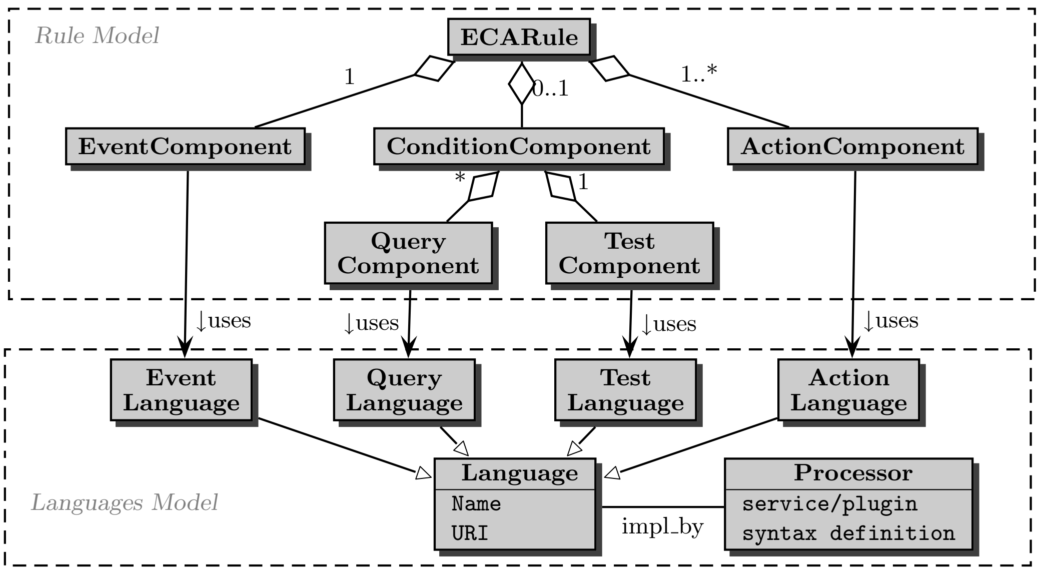


Figure 2.1: ECA Rule Components and Corresponding Languages (from [1])

Language Binding. As every component can use an arbitrary language, it is necessary to provide a uniform way of handling the respective component languages. To achieve this, rules, their components and the languages themselves are turned into objects of the Semantic Web by describing them in XML and RDF/OWL in a generic rule ontology.

As shown in Figure 2.1, every language is associated with an appropriate language processor that implements the language's semantics. The rule evaluation service can then exploit this association and handle all languages inside the rule components the same way. A detailed description of how languages are mapped to language processors is given in Section 3.1.

Example 2.3 *Consider a rule that sends a message to the owner of the car-rental company containing the bookings of the current day. The event part may be written in* SNOOP*, allowing to detect and cumulate these kinds of events. To extract the actual content (e.g. customer name and car type) from the events, some local* XPath *queries are executed. As the owner also wants to know how many cars are still available the next day, the rule may contain another query part written in* XQuery*. As there is no test part in this example, it is simply omitted. The action part may then be written in any language that allows for sending mails.*

Communication between Rule Components. A rule component often depends on information gathered by another component before (e.g., to extract the customer name from the event in the previous example, the query component must be able to access it). In this context, the

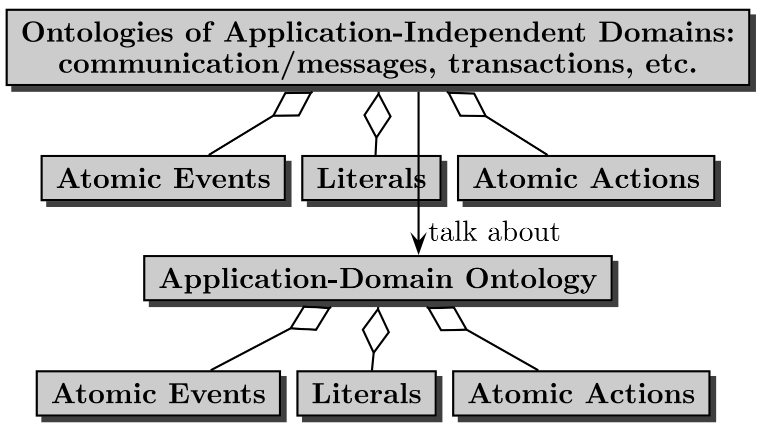

Ontologies of Application-Independent Domains: communication/messages, transactions, etc.
Atomic Events
Literals
Atomic Actions
talk about
Application-Domain Ontology
Atomic Events
Literals
Atomic Actions

Figure 2.2: Kinds and Components of Ontologies (from [1])

use of *logical variables* (see Example 2.2) provides a suitable mechanism for communicating this information between the rule components. Thus, every component may *bind variables* and/or *use variables* that have been bound before. A more extensive description of the use of variables is given in Section 4.1.

2.3 Domain Ontologies

The purpose of domain ontologies is to describe all aspects of a specific domain. Besides the *static* aspects, in the Semantic Web represented by *resources*, this also includes *dynamic* aspects like *events* and *actions*.

As Figure 2.2 illustrates, there are two different kinds of ontologies used in the Semantic Web to define an application.

Application domain ontologies describe the primary domain of the application. An ontology for a travel domain, for example, might describe resources like train and flight schedules, actions like booking a flight and events like “flight booked” or “flight fully booked”.

In contrast, there are application-independent domain ontologies that provide a generic infrastructure and talk about the application domain. For example, the “calendar” domain might define days and months as resources and provide events like “first of month”.

A complete application in the Semantic Web is then described by employing a combination of both types of ontologies.

Inside a domain ontology, the description of an atomic action contains its pre- and postconditions and the specification of an agent that is responsible for its execution. Atomic events

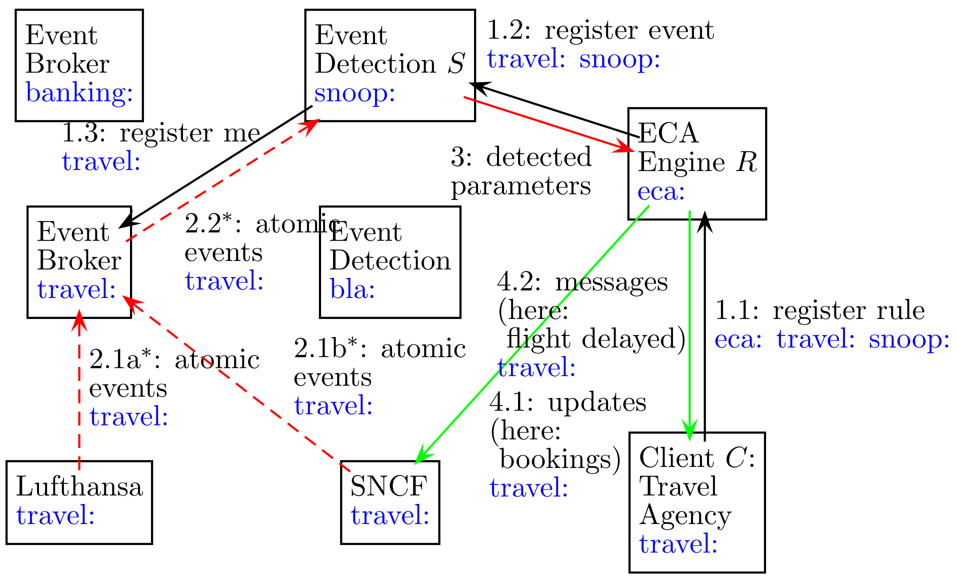

Event Broker banking:
Event Detection S snoop:
1.2: register event travel: snoop:
1.3: register me travel:
3: detected parameters
ECA Engine R eca:
Event Broker travel:
2.2*: atomic events travel:
Event Detection bla:
4.2: messages (here: flight delayed) travel:
1.1: register rule eca: travel: snoop:
2.1a*: atomic events travel:
2.1b*: atomic events travel:
4.1: updates (here: bookings) travel:
Lufthansa travel:
SNCF travel:
Client C: Travel Agency travel:

Figure 2.3: General Framework Architecture (from [1])

directly result from atomic actions[3]. The action "book flight (seat: 67, flightno: LH1234)", for example, might result in the events "booked seat 67 of flight LH1234" and "50% of the seats of flight LH1234 are booked". Thus, the domain ontology also describes the correlation between atomic actions and the atomic events that possibly result from them.

2.4 General Framework Architecture

As the (Semantic) Web consists of a huge number of independent nodes, it is not surprising to find the ECA framework being employed in a completely distributed environment.

The sources of events are represented by services inside the application domain while the clients may be arbitrary nodes in the Semantic Web that want to react to these events by defining rules that specify the intended reactive behavior.

Acting as mediators, there are services that handle the event propagation (e.g. event brokers) and the detection of events. The rule evaluation itself is finally done by a local or remote rule evaluation service — the ECA engine. Thus, the ECA engine can be seen as the main control unit for reactive behavior inside the framework although it does not detect events, handle queries or execute actions itself.

The framework architecture is illustrated in Figure 2.3 based on an example from the "travel" domain.

[3]Note that this is *not* a one-to-one relation, since a single action might result in multiple events.

Example 2.4 *Consider the case when a travel agency wants to be informed, when a flight is delayed and reschedule the passengers to new connecting trains.*

Thus, the travel agency would register a rule at the ECA engine (1.1), containing a composite event of the "travel" domain that is composed of the events "new flight", an arbitrary number of "flight booked" events (that contain the names of the passengers) and a final "flight delayed" event. The event component is given using SNOOP. At the ECA engine, the rule is analyzed and the event component registered at an appropriate event detection engine that is capable of detecting composite events specified in SNOOP (1.2).

To be able to detect the registered event, this engine needs a way of becoming aware of all relevant atomic events. There are several solutions to this problem. The event detection engine might for example contact a dedicated event broker for the "travel" domain and request to be informed about all atomic events from this domain (1.3). The event broker is then used by the different applications inside the "travel" domain to propagate their atomic events (2.1). Thus, it only needs to forward these events to the event detection engine (2.2).

Another possible solution is to instruct the client to find out about the relevant events itself (not shown). When an event occurs, the client forwards it to the rule evaluation service which in turn forwards it to all known event detection engines. This makes it possible to fake events directly at the client, which is an important feature during the development phase of the prototype.

When the composite event is detected, the according event detection engine sends a message to the ECA engine (3) that hereupon triggers the evaluation of the appropriate rule. Since the required information for the rescheduling of the passengers was already gathered in the event component (the "flight booked" events), the ECA engine can directly update the reservations for the connecting trains (4.1) and inform the travel agency about the result (4.2).

In another scenario, the ECA engine might have to contact several other query services (again of the "travel" domain) in order to retrieve the required information.

ECA Engine. The ECA engine is responsible for the evaluation of rules. When it becomes aware that a relevant event was detected, it creates a new instance of the according rule(s), evaluates the specified queries, the test and possibly executes the actions. Depending on the concrete implementation, the ECA engine may furthermore be responsible for the the management of rules[4] (registration and deregistration) and the forwarding of atomic events to all known event detection engines.

[4]This is the case in the prototype developed in this thesis.

Language Processors. As described in Section 2.2.2, every component language is associated with a language processor that is capable of evaluating expressions of the implemented language. During the evaluation of a rule, the ECA engine determines the appropriate language processor for each rule component and communicates the respective component to this processor. Next, the language processor evaluates the component and finally returns the result to the ECA engine that hereupon proceeds with the evaluation of the rule.

An enumeration of the developed query, test and action engines can be found in Section 5.4 and Section 5.6.

3 Rule Markup

3.1 Rule Markup (ECA-ML) and Language Binding

For the definition of rules, the ECA framework provides an XML markup language, ECA-ML, that was presented in [9]. The basic structure of a rule is as follows:

```
<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>
```

The association of the rule components with their specific languages is done at the expression level. There are three kinds of expressions, namely atomic, composite and opaque ones.

- *Atomic expressions* (atomic event, literal or action) belong to a domain language that can be directly identified by the namespace of the expression (cf. Section 2.3).
- *Composite expressions* consist of a composer and several subexpressions which all belong to (possibly different) languages of the same kind. Here the language association is defined by the namespace of the expression's root node.
- The third type - *opaque expressions* - are not in XML markup and thus do not have a namespace definition by themselves. For still being able to determine an appropriate language processor, it is necessary to include the namespace information inside an attribute

of the enclosing element. The language binding of opaque expressions is explained in detail in the next section.

3.2 Opaque Expressions

Opaque expressions provide a way to embed services that do not have an XML markup (e.g. XPath) and/or are not framework-aware. This is especially important during the development of the prototype, as these represent the majority of services inside the current environment.

An opaque expression is marked up the following way:

```
<eca:(event|query|test|action)>
  <eca:opaque attributes>
    opaque code
  </eca:opaque>
</eca:(event|query|test|action)>
```

Since the expression itself is not marked up in XML, there is no namespace declaration the ECA engine could use to find out about the correct language processor. To solve this issue, there are several attributes of the <eca:opaque> element that specify how the according expression is to be evaluated.

- The attribute lang can be used to specify the namespace of the expression language directly. This is useful to specify framework-aware wrappers around language processors whose implemented language does not have an XML markup, e.g. an XQuery engine (see Section 5.4).

- While the previously mentioned XQuery engine represents a generic service (it is not important *which* engine finally evaluates the expression), there are cases, e.g. updates to relational databases, where it *does* matter to which node the request is sent to. By specifying a URI address with the uri attribute, the ECA engine can be forced to send the request to this specific node.

- Many services that are not framework-aware can be queried (or invoked in case of an update) by simple HTTP-GET or the more complex HTTP-POST requests. In order to integrate those services, the attribute method can be given in addition to the previously mentioned uri. Obviously, there are two possible values for method, namely "get" and "post" which instruct the ECA engine to use the respective method.

  For the communication of *input variables* (see Section 4.1) the following logic is applied. At first, the content of the <eca:opaque> element is inspected. If the variable name occurs inside it, it is replaced by the value of the variable. Otherwise, the variable is communicated by appending it as a parameter to the URI in the form “name=value”.

Example 3.1 *Consider the following XML document to be available at the local node, referenced by the URL* http://localhost/customers.xml*:*

```
<customers>
  <customer name="John Doe" mail="john@doe.nop"/>
  <customer name="Lisa Miller" mail="lisa@miller.nop"/>
  <customer name="Jack Miller" mail="jack@miller.nop"/>
</customers>
```

A query component, using an opaque XPath expression, now might query for the customer elements:

```
<eca:query>
  <eca:opaque lang="http://www.w3.org/XPath">
    document('http://localhost/customers.xml')/customers/*
  </eca:opaque>
</eca:query>
```

The value of lang *is mapped to an appropriate language processor to which the query is then sent. In return, the following result is retrieved:*

```
<customer name="John Doe" mail="john@doe.nop"/>
<customer name="Lisa Miller" mail="lisa@miller.nop"/>
<customer name="Jack Miller" mail="jack@miller.nop"/>
```

In order to make this result available to the subsequent rule components, it has to be bound to a variable. How this is expressed in the markup language and how the following components can use the value of this variable (which must then be communicated to the language processor) is the main focus of the next chapter. Furthermore, the previous example raises the question, what should happen to results that consist of multiple answers to a query.

4 Variables and Communication

On the abstract rule level, the individual components of the rule can communicate their collected information by *binding variables*. Any following component may then *use* the values of these variables to narrow down its own query (or execute an action). A detailed description of the variable concept in the ECA framework is given in the first section of this chapter, followed by a presentation of the according markup elements.

When a rule component is actually evaluated, it must be communicated along with its required variable bindings to an appropriate language processor. How this communication takes place and what messages are sent is then explained in the last section.

4.1 Variable Concept

In the ECA framework, a concept of variables is provided that is similar to those found in logical languages. During the evaluation of a rule, variables occur as *free variables* in the scope of the rule and can appear positively or negatively inside the components (see Example 2.2).

If a variable has not already been bound, a *positive occurrence* binds it to a value[1], otherwise it acts as a join variable.

A *negative occurrence* of a variable uses the value it has previously been bound to. Thus, if a variable is used negatively inside a component, it must have been bound before to provide the necessary safety. Although logical variables are sufficient at the abstract rule level there may also be situations during the actual rule evaluation where they need to be used as arguments and results. This leads to a slightly different definition of variables at the operational level which is illustrated in Figure 4.1:

- **used variables** represent all variables that *may* be communicated to the language processor in order to minimize the result as early as possible. They are equivalent to free variables in logical languages.
- **input variables** denote the set of variables that *must* be provided in order to successfully evaluate the expression. They correspond to negative occurrences of free variables.

[1]A variable can be bound to literals, references (URIs), XML or RDF fragments, or events.

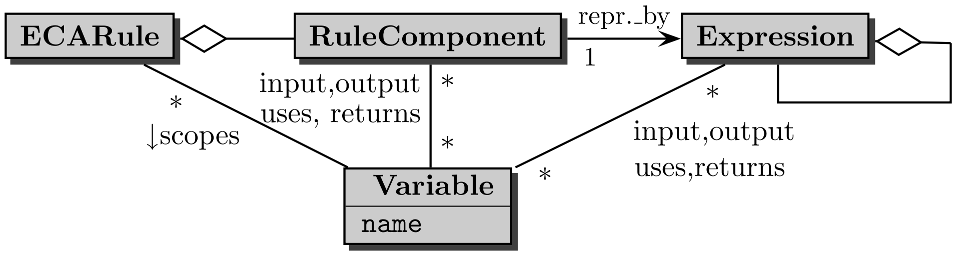


Figure 4.1: Use of Variables in Components (from [6])

- **output variables** are variables that are bound by the language processor in any case. If they have been bound before they *must* be considered by an equi-join.
- **returned variables** are those variables that are communicated to the language processor but not included in the resulting answer. The rule evaluation service has to replenish these variables to ensure correct join semantics.

At the end of the previous chapter, Example 3.1 raised the need for a mechanism of handling multiple answers to a query[2]. To meet this need, the concept of variables is extended to allow for a variable being bound to different values (when specified at once). Variable bindings therefore consist of a set of tuples that in turn contain the actual values of the variables.

Example 4.1 *Consider again Example 3.1. If the resulting elements were bound to the variable "Customer", the resulting variable bindings would look as illustrated in Figure 4.2.*

Variable Bindings

Tuple 1
Customer → <customer name="John Doe" mail="john@doe.nop">

Tuple 2
Customer → <customer name="Lisa Miller" mail="lisa@miller.nop">

Tuple 3
Customer → <customer name="Jack Miller" mail="jack@miller.nop">

Figure 4.2: Multiple Tuples of Variable Bindings

[2] A query on an XML document that returned three elements.

4.2 Markup for Binding and Using Variables

At the rule level, the result of an event or query component can be bound to a variable by enclosing the component with the definition of the variable in the following way:

```
<eca:variable name="name">
  <eca:(event|query)>
    expression
  </eca:(event|query)>
</eca:variable>
```

The attribute name defines the name of the variable and the enclosed element contains an arbitrary expression of the respective event or query language. After the evaluation of the event or query component, the result of the expression is bound at once to a new variable of the specified name. If the evaluation returns multiple results, each result will be bound in a separate tuple as described in the previous section.

Since at first the vast majority of rule components will be given as opaque expressions whose results have to be bound to variables, the following syntax can be used to shorten the component definition:

```
<eca:variable name="name" lang="language" select="expression"/>
```

This variable definition is equivalent to:

```
<eca:variable name="name">
  <eca:query>
    <eca:opaque lang="language">
      expression
    </eca:opaque>
  </eca:query>
</eca:variable>
```

To access information gathered by the previous rule components, every component may make use of an arbitrary number of (previously bound) variables. In order to prevent the necessity of always sending all variable bindings to the language processor, the required variables have to be specified inside the component:

```
<eca:(query|test|action)>
  <eca:input-variable name="..." use="..."/>
  <eca:use-variable name="..."/>
  <eca:output-variable name="..."/>
  <eca:return-variable name="..."/>
  expression
</eca:(query|test|action)>
```

The naming of the elements is analogous to Section 4.1. By specifying the additional attribute use with the <eca:input-variable> element, the variable can temporarily be renamed inside the actual component.

Example 4.2 *Consider the following XML document to be available at the local node, referenced by the URL* http://localhost/customer-cars.xml*:*

```
<customer-cars>
  <car owner="John Doe">Golf</car>
  <car owner="John Doe">Passat</car>
  <car owner="Lisa Miller">Corolla</car>
  <car owner="Jack Miller">Focus</car>
</customer-cars>
```

The following query component then returns the cars that a customer owns:

```
<eca:variable name="OwnCar">
  <eca:query>
    <eca:input-variable name="Person"/>
    <eca:query>
      <eca:opaque lang="http://www.w3.org/XPath">
        document("http://localhost/customer-cars.xml")
        /customer-cars/car[@owner=$Person]/text()
      </eca:opaque>
    </eca:query>
  </eca:query>
</eca:variable>
```

It uses the variable “Person” as input and binds the result of the query to the variable “OwnCar”. Consider “Person” to be bound to the value “John Doe”. The evaluation will then result in the variable bindings

$$\mathsf{OwnCar} \rightarrow \textit{Golf}$$
$$\mathsf{OwnCar} \rightarrow \textit{Passat}$$

that represent the two different answers to the query.

This leads to the question how these answers are communicated and how the language processor is invoked in the first place. Since the variable bindings also must be communicated to the language processor (e.g., a subsequent query component might ask for the availability of equivalent cars), a common format for their exchange has to be specified that is understood by all framework-aware language processors.

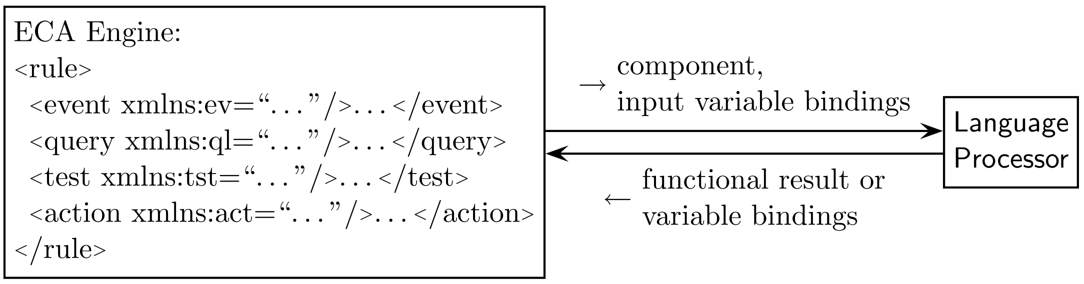

ECA Engine:
<rule>
<event xmlns:ev="..."/>...</event>
<query xmlns:ql="..."/>...</query>
<test xmlns:tst="..."/>...</test>
<action xmlns:act="..."/>...</action>
</rule>
→ component, input variable bindings
← functional result or variable bindings
Language Processor

Figure 4.3: Communication of ECA Engine and Language Processor (adapted from [3])

4.3 Communication between Framework Components

The communication of variable bindings represents a central aspect during the evaluation of a rule. While variables may already be specified before the event component (containing only static values), they are in any case returned by the service that detected the triggering event.

These variable bindings then form the basis of the evaluation of the subsequent query components which extend them by binding further variables. After the evaluation of the test component potentially discarded some of the resulting tuples, the action components are executed for each of the remaining ones.

Since the actual evaluation of a rule component happens at a remote language processor, the variable bindings required for the evaluation must somehow be communicated in addition to the component itself (see Figure 4.3). The language processor then returns the functional result or the resulting variable bindings, depending on the type of language used (functional or logical). In any case, the request message to and the answer message from the language processor must contain an identification of the rule component in order to later correctly process the result. When the ECA engine is notified about the result of the evaluation, it joins the resulting variable bindings with its own.

In the following, a format for the communication of variable bindings is presented and it is shown how this format is integrated into the request and answer messages exchanged with the language processors.

4.3.1 Communication of Variable Bindings

For the communication of variable bindings between the different nodes inside the framework, the following markup is used:

```
<eca:variable-bindings>
  <eca:tuple>
    <eca:variable name="name" ref="URI"/>
    <eca:variable name="name">
      any value
    </eca:variable>
      :
  </eca:tuple>
  <eca:tuple>...</eca:tuple>
  :
  <eca:tuple>...</eca:tuple>
</eca:variable-bindings>
```

Example 4.3 *Consider again Example 4.2 on page 20. The evaluation of the query component presented there resulted in the following variable bindings:*

$$\mathsf{OwnCar} \rightarrow \mathit{Golf}$$
$$\mathsf{OwnCar} \rightarrow \mathit{Passat}$$

Applying the specified exchange format, the variable bindings are marked up the following way:

```
<eca:variable-bindings>
  <eca:tuple>
    <eca:variable name="OwnCar">Golf</eca:variable>
  </eca:tuple>
  <eca:tuple>
    <eca:variable name="OwnCar">Passat</eca:variable>
  </eca:tuple>
</eca:variable-bindings>
```

4.3.2 Request and Answer Messages

Communication between different framework nodes is based on the exchange of <eca:request> and <eca:answers> messages. An <eca:request> message is sent by the ECA engine when it invokes a language processor to evaluate a rule component. The language processor then wraps the result inside an <eca:answers> message and sends it back to the ECA engine.

The structure of both message types is very similar. Their root elements contain four attributes that uniquely identify the rule instance they belong to.

- The attribute ref contains a uri of the form http://localhost:8081/rules#rule2 that identifies the rule and the ECA engine where it is located.
- The rule component is identified by the attribute component. The following rule shows the possible values for component in brackets.

  ```
  <eca:rule>
    <eca:event>event component (event)</eca:event>
    <eca:query>first query component (query[1])</eca:query>
    <eca:query>second query component (query[2])</eca:query>
    ...
    <eca:test>test component (test)</eca:test>
    <eca:action>first action component(action[1])</eca:action>
    <eca:action>second action component (action[2])</eca:action>
    ...
  </eca:rule>
  ```

- In contrast to the first two attributes which provide static information about the rule definition, the attributes timestamp and instance combined identify the correct rule instance. timestamp holds the point in time when the rule instance was created in milliseconds since January 1, 1970. As there may exist multiple rule instances that where created at the same time, the attribute instance uniquely identifies the correct one.

Request Messages. As its first child, an <eca:request> message contains the rule component that is to be evaluated. If the evaluation of the component requires the use of variable bindings, these are inserted immediately after the component itself. The following XML document shows a generic request message.

```
<eca:request ref="..." component="..." timestamp="..." instance="...">
  <eca:[query|test|action]>
    ...
  </eca:[query|test|action]>
  <eca:variable-bindings>
    ...
  </eca:variable-bindings>
</eca:request>
```

Answer Messages. In contrast, an <eca:answers> message is composed of an arbitrary number of <eca:answer> elements - one for each answer to the request. The answer element then contains an optional <eca:result> element (in case of a functional result) and the resulting variable bindings[3]. A generic answers message looks like the following:

```
<eca:answers ref="..." component="..." timestamp="..." instance="...">
  <eca:answer>
    <eca:result>
      ...
    </eca:result>
    <eca:variable-bindings>
      ...
    </eca:variable-bindings>
  </eca:answer>
  .
  .
  <eca:answer>
    ...
  </eca:answer>
</eca:answers>
```

Example 4.4 *An exemplary request message for the query from Example 4.2 that takes the variable "Person" as input looks as follows.*

```
<eca:request ref="..." component="..." timestamp="..." instance="...">
  <eca:query>
    <eca:opaque lang="http://www.w3.org/XPath">
      document("http://localhost/customer-cars.xml")
      /customer-cars/car[@owner=$Person]/text()
    </eca:opaque>
  </eca:query>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="Person">John Doe</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:request>
```

The answer to this request might then be:

```
<eca:answers ref="..." component="..." timestamp="..." instance="...">
  <eca:answer>
    <eca:result>Golf</eca:result>
    <eca:variable-bindings>
      <eca:tuple>
        <eca:variable name="Person">John Doe</eca:variable>
      </eca:tuple>
    </eca:variable-bindings>
```

[3]The resulting variable bindings consist of variables bound during the evaluation and the original input variables to identify the correct tuple this answer belongs to.

```
  </eca:answer>
  <eca:answer>
    <eca:result>Passat</eca:result>
    <eca:variable-bindings>
      <eca:tuple>
        <eca:variable name="Person">John Doe</eca:variable>
      </eca:tuple>
    </eca:variable-bindings>
  </eca:answer>
</eca:answers>
```

Since the language used in this example returns a functional result, the ECA engine has to bind it to a variable as specified in the rule definition. In contrast, the result of a logical language to the same query (without the enclosing variable definition) might look like the following.

```
<eca:answers ref="..." component="..." timestamp="..." instance="...">
  <eca:answer>
    <eca:variable-bindings>
      <eca:tuple>
        <eca:variable name="Person">John Doe</eca:variable>
        <eca:variable name="OwnCar">Golf</eca:variable>
      </eca:tuple>
    </eca:variable-bindings>
  </eca:answer>
  <eca:answer>
    <eca:variable-bindings>
      <eca:tuple>
        <eca:variable name="Person">John Doe</eca:variable>
        <eca:variable name="OwnCar">Passat</eca:variable>
      </eca:tuple>
    </eca:variable-bindings>
  </eca:answer>
</eca:answers>
```

4.3.3 Specification of Variable Handling

During the evaluation of a rule the ECA engine integrates various kinds of language processors. These processors may differ heavily in the method of communication and their capabilities wrt. to the handling of variable bindings. Thus, a mechanism is needed for the ECA engine to find out about all relevant information — the Service Description (SD).

In the first prototype the SD is a simple XML document that contains one element for each feature a language processor may have. Furthermore, as there is currently no way to specify a communication method[4], all framework-aware processors must use the same technology[5].

A generic service description looks as follows:

```
<eca:service language="http://...">
  <eca:multiple-input>(true|false)</eca:multiple-input>
  <eca:send-use>(true|false)</eca:send-use>
</eca:service>
```

The root attribute language specifies the URI of the language that is implemented by the processor. At this time a language processor may only implement a single language.

The first child element, <eca:multiple-input>, indicates if the processor can handle variable bindings that contain more than one tuple. If so, it must replenish the resulting variable bindings with the original input variables to allow for the correct identification of the initial tuple. If the language processor is not capable of handling multiple tuples, the ECA engine itself needs to iterate over them and invoke the processor for each of them.

Example 4.5 *Consider again the variable bindings shown in Example 4.3:*

```
<eca:variable-bindings>
  <eca:tuple>
    <eca:variable name="OwnCar">Golf</eca:variable>
  </eca:tuple>
  <eca:tuple>
    <eca:variable name="OwnCar">Passat</eca:variable>
  </eca:tuple>
</eca:variable-bindings>
```

A subsequent query component might map the given cars to their classes (sizes) according to the following XML document:

```
<classes>
  <class name="B">
    <car>Golf</car>
    <car>Corolla</car>
  </class>
  <class name="C">
    <car>Passat</car>
  </class>
</classes>
```

The query component itself might contain an XPath expression:

[4]For language processors that are not framework-aware this is possible by using the <eca:opaque> element as described in Section 3.2.

[5]The first prototype uses SOAP over HTTP.

```
<eca:variable name="Class">
  <eca:query>
    <eca:input-variable name="OwnCar"/>
    <eca:opaque lang="http://www.w3.org/XPath">
      document("...")/classes/class[car/text()=$OwnCar]/@name
    </eca:opaque>
  </eca:query>
</eca:variable>
```

When the ECA engine invokes a language processor that is capable of handling multiple tuples of variable bindings they can simply be attached to the request message unchanged:

```
<eca:request ref="..." component="..." timestamp="..." instance="...">
  <eca:query>
    <eca:opaque lang="http://www.w3.org/XPath">
      document("...")/classes/class[car/text()=$OwnCar]/@name
    </eca:opaque>
  </eca:query>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="OwnCar">Golf</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="OwnCar">Passat</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:request>
```

If however, the language processor can only handle a single tuple per invocation, the ECA engine has to split the variable bindings and invoke the processor twice:

```
<eca:request ref="..." component="..." timestamp="..." instance="...">
  <eca:query>
    <eca:opaque lang="http://www.w3.org/XPath">
      document("...")/classes/class[car/text()=$OwnCar]/@name
    </eca:opaque>
  </eca:query>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="OwnCar">Golf</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:request>
```

and

```
<eca:request ref="..." component="..." timestamp="..." instance="...">
  <eca:query>
```

```
    <eca:opaque lang="http://www.w3.org/XPath">
      document("...")/classes/class[car/text()=$OwnCar]/@name
    </eca:opaque>
  </eca:query>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="OwnCar">Passat</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:request>
```

The second child element of the service description, <eca:send-use>, defines if the ECA engine should also send *used variables*[6] along with the request. This is especially relevant for logical languages, as it can reduce the result of a query at an early stage by binding otherwise free variables. If these additional variables are supplied, a first join can already happen during the evaluation at the language processor. Otherwise, a much larger result may be returned which then has to be joined at the ECA engine.

Example 4.6 *Consider the following facts that represent cars available for rent and their according location:*

```
available('Golf', 'Paris').
available('Corolla', 'Paris').
available('C6', 'Munich').
available('Passat','Munich').
```

A query in a logical language like Datalog (see Example 2.2 on page 5) could return the variables "AvailableCar" and "City":

```
<eca:query>
  <eca:use-variable name="City"/>
  <eca:opaque lang="prolog">
    ?- available(AvailableCar, City).
  </eca:opaque>
</eca:query>
```

If no value for "City" is provided, this would result in the following answer that contains all cars in all cities:

```
<eca:answer ref="..." component="..." timestamp="..." instance="...">
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="AvailableCar">Golf</eca:variable>
```

[6] See Section 4.1 for an explanation of the different kinds of variables.

```
      <eca:variable name="City">Paris</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="AvailableCar">Corolla</eca:variable>
      <eca:variable name="City">Paris</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="AvailableCar">C6</eca:variable>
      <eca:variable name="City">Munich</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="AvailableCar">Passat</eca:variable>
      <eca:variable name="City">Munich</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:answer>
```

When a value for "City" is provided to the language processor, it can already join the query's result with this value and return only those tuples that match the city name. Thus, a value of "Paris" would result in the following answer:

```
<eca:answer ref="..." component="..." timestamp="..." instance="...">
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="AvailableCar">Golf</eca:variable>
      <eca:variable name="City">Paris</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="AvailableCar">Corolla</eca:variable>
      <eca:variable name="City">Paris</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:answer>
```

Up to now, the different aspects of rule evaluation have only been presented in isolation. The next chapter therefore combines the results gathered so far and describes the complete process in the aggregate on the basis of a working example.

5 Evaluation of ECA Rules

After the theoretical description of ECA-style rules, the ECA framework and the communication between the individual framework components in the previous chapters, this chapter focuses on the process of rule evaluation.

During the evaluation of a rule, several components are working together, namely the ECA engine and a variable set of autonomous language processors. The ECA engine controls *when* to evaluate *which* rule component and keeps the state information during the evaluation. The actual evaluation of a rule component then happens at an appropriate language processor that is capable of processing expressions of the implemented language in consideration of the given variable bindings.

This process is demonstrated in the subsequent examples in this chapter on the basis of the following exemplary ECA rule that offers cars owned by a car-rental company to the customers on the event of a flight booking.

Example 5.1 *Consider the following (abstract) ECA rule that is used in the examples throughout this chapter:*

```
<eca:rule xmlns:eca="http://www.eca.org/eca-ml">
  <eca:variable name="car-rental-url">
    http://localhost:8081/exist/servlet/db/travel/car-rental.xml
  </eca:variable>
  <eca:event>
    <eca:atomic-event><booking person="$Person" to="$To"/></eca:atomic-event>
  </eca:event>
  <eca:variable name="OwnCar">
    <eca:query><!-- query the person's cars --></eca:query>
  </eca:variable>
  <eca:variable name="Class">
    <eca:query><!-- map the cars to the appropriate classes --></eca:query>
  </eca:variable>
  <eca:query>
    <!-- query for cars that are available at the destination. this query returns
         multiple variable bindings and is therefore not bound to a variable itself.  -->
  </eca:query>
  <eca:test><!-- omitted --></eca:test>
  <eca:action><!-- inform the customer about available cars --></eca:action>
</eca:rule>
```

At the time the rule evaluation is triggered, the following facts are known: the name of the person who booked the flight and the destination city. These are used to ask for the cars the customer owns at home which are then mapped to predefined classes. In a further query, all *cars that are available at the destination city are acquired and compared to the cars owned by the customer. Finally, the resulting list is sent to the customer.*

5.1 Rule Registration

Upon the registration of a rule by a client, the ECA engine needs to execute a sequence of actions. At first, the rule may be validated against a given DTD or a XML Schema to ensure its syntactical correctness. The next step is to assign an ID to the rule and make it persistent (e.g. by storing it inside a database). Afterwards, the static variable definitions, that may be given before the event specification of the rule, must be evaluated and bound. Finally, the event part itself is taken and registered at an appropriate event detection engine along with the needed variable bindings. When a rule is deregistered the inverse tasks must be executed.

After the rule has been successfully registered the ECA engine remains in an idle state, waiting for the specified event pattern to be detected.

Since an event detection engine has not yet been implemented, the ECA engine currently only stores the rule without taking any further actions. Notifications of event occurrences are sent by the ECA engine client (see Section 5.8). In the future, the event detection component from ruleCore[1] [14] could be integrated into the framework by providing an "opaque" wrapper that transforms the registration and answer messages.

5.2 Evaluation of the Event Component

During the registration of a rule, the event component is registered at an appropriate event detection engine that is responsible for its further evaluation. The ECA engine hereupon suspends the evaluation of the event component and waits for the specified event pattern to be detected. When a respective answer message arrives that contains the detected event sequence, the ECA engine resumes the evaluation[2].

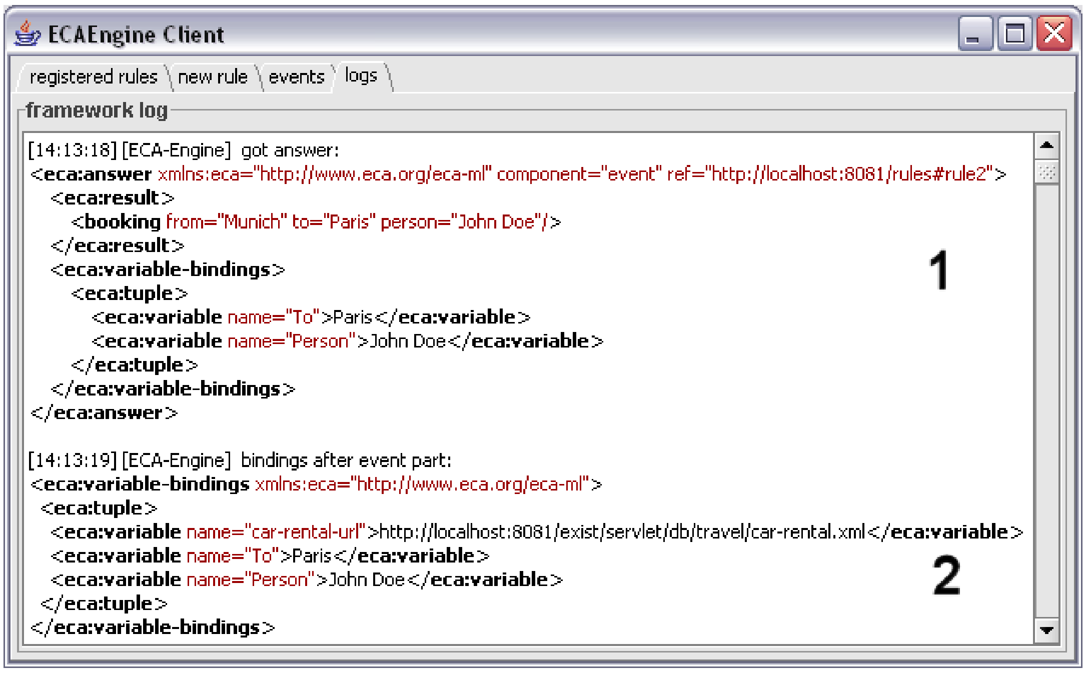
If the detected event contains information that needs to be extracted, the event part of the rule is enclosed by a <eca:variable> element. Thus, the existing variable bindings will be extended with a variable that contains the detected sequence of events.

Example 5.2 *The evaluation of the exemplary rule is triggered when the event*

<booking person="John Doe" from="Munich" to="Paris"/>

[1]ruleCore is a registered trademark of MS Analog Software kb.

[2]Note that an event component can be detected any number of times. Each detection leads to an independent "firing" of the rule.



```
[14:13:18] [ECA-Engine]  got answer:
<eca:answer xmlns:eca="http://www.eca.org/eca-ml" component="event" ref="http://localhost:8081/rules#rule2">
  <eca:result>
    <booking from="Munich" to="Paris" person="John Doe"/>
  </eca:result>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="To">Paris</eca:variable>
      <eca:variable name="Person">John Doe</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:answer>

[14:13:19] [ECA-Engine]  bindings after event part:
<eca:variable-bindings xmlns:eca="http://www.eca.org/eca-ml">
 <eca:tuple>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
</eca:variable-bindings>
```

Figure 5.1: Detection of the Event Component

occurs and is detected by an atomic event detection engine. The ECA engine receives an answer message containing the materialized event along with the bound variables (Figure 5.1[1]) and retrieves the appropriate rule (identified by the ref *attribute of the message).*

At first, the variable car-rental-url *is bound to the given string literal. Next, the event component is evaluated. As it is not enclosed by an* <eca:variable> *element, it does not have to be bound to a variable. Nevertheless, the ECA engine needs to extract the variable bindings that were provided by the event detection engine and join them with its local bindings (Figure 5.1[2]).*

5.3 Evaluation of the Query Components

A query component may be marked up using either an <eca:query> element or an <eca: variable> element containing a select attribute. In the latter case, the ECA engine needs to temporally expand the variable element as described in Section 4.2 to contain a correctly marked up <eca:query> element.

Afterwards, the ECA engine determines the required variable bindings it has to send along with the query by examining the component for <eca:input-variable> elements.

Next, the ECA engine analyzes the namespace of the expression's root element. If the component contains an opaque expression, the necessary information is extracted according to Section 3.2. The namespace is hereupon used to determine an appropriate language processor. Before the ECA engine can send the query to this language processor, it has to find out about the processor's capabilities wrt. the handling of variable bindings. This is accomplished by obtaining and analyzing its service description (see Section 4.3.3).

After consolidating all gathered information, the query is finally send to the language processor for the actual evaluation.

The language processor evaluates the query and returns the result inside an answer message. In case of a functional result, the ECA engine needs to bind it to a variable before it can join the resulting variable bindings with its own. Otherwise, the variable bindings can be joined directly.

In the last step, the ECA engine computes the next rule component and triggers its evaluation.

5.4 Employed Query Engines

This section describes several query engines that were implemented during the development of the ECA engine and shows in detail how the query components of Example 5.1 are actually evaluated.

XPath Engine. This engine interprets expressions of the language XPath [20] which is a query language for XML and is useful for addressing parts of an XML document. XPath forms the basis of several other languages, including XQuery, which is described in the next section.

Example 5.3 *Consider the following simple XML document:*

```
<a>
  <b nr="1">one</b>
  <b nr="2">two</b>
</a>
```

The XPath expression /a/b[@nr="2"] *results in the node*

```
<b nr="2">two</b>
```

XQuery Engine. XQuery is a declarative and functional query language for XML documents that extends XPath. Expressions of this language are composed according to the for, let, where, order by, return paradigm that is similar to SQL's Select, From, Where.

- The for clause allows for iterating over one or more sets of nodes. During each iteration, the current node is bound to the specified variable. Since the result of the for clause represents an ordered sequence of tuples of bound variables, it is called the tuple stream.
- Contrary to the for clause, the let clause binds variables to the result of an expression without iteration. The previously mentioned tuple stream is then extended with these variable bindings.
- If a where clause is present, it is evaluated once for every tuple. Depending on the result, the respective tuple is kept or discarded.
- Usually, every XQuery expression returns its results in document order. Using the order by clause makes it possible to specify a different sort order.
- The return clause finally constructs the result of the expression and is evaluated once for every tuple of variable bindings.

Example 5.4 *Consider the following XML document:*

```
<a>
  <b nr="1">one</b>
  <b nr="2">two</b>
  <b nr="3">three</b>
</a>
```

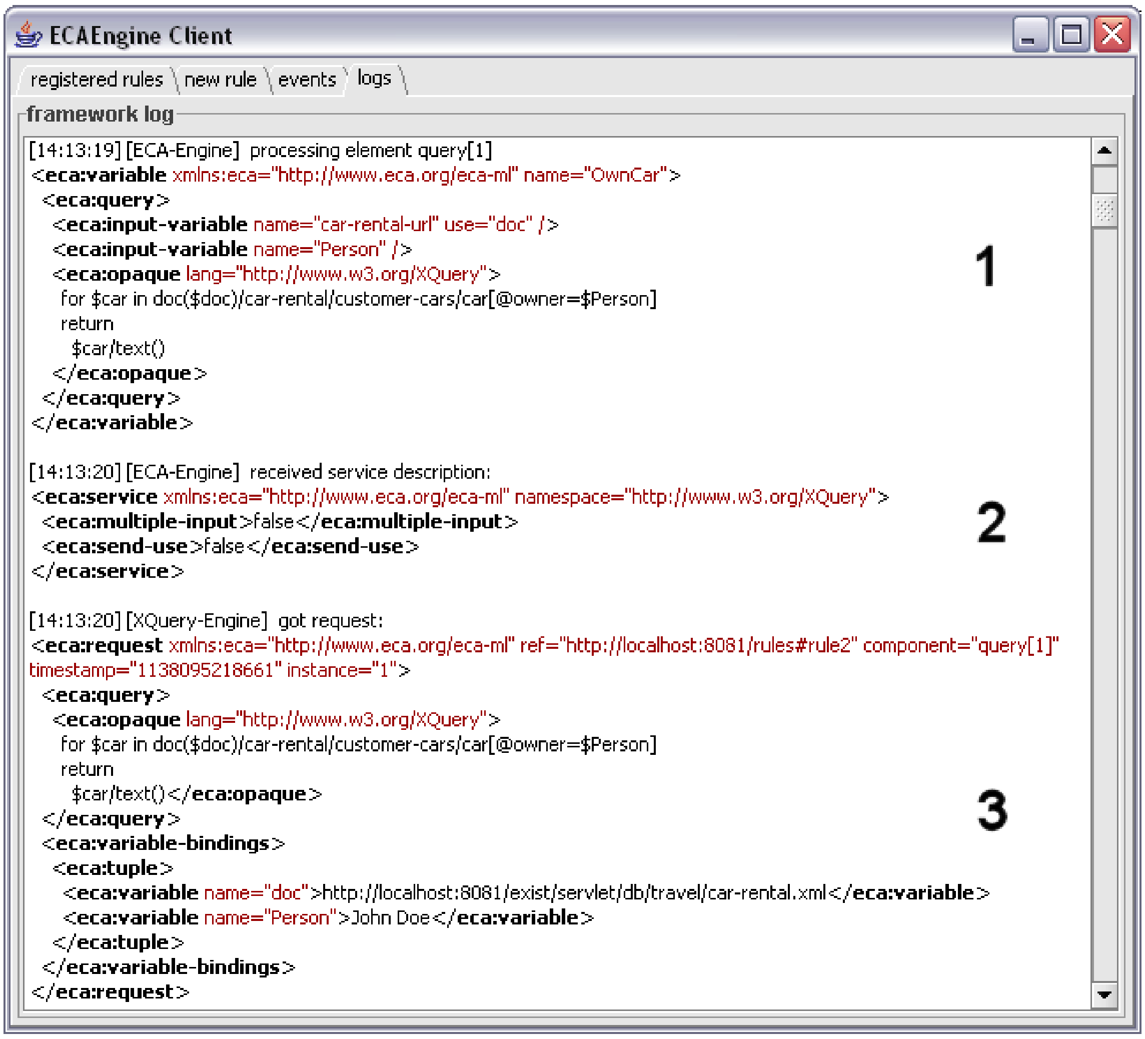
Then, the exemplary XQuery expression

```
<result>
{
  for $b in //b
  where $b/@nr > 1
  return <number>{$b/text()}</number>
}
</result>
```

results in the following XML document:

```
<result>
  <number>two</number>
  <number>three</number>
</result>
```

A more detailed description of XQuery is available at [21].

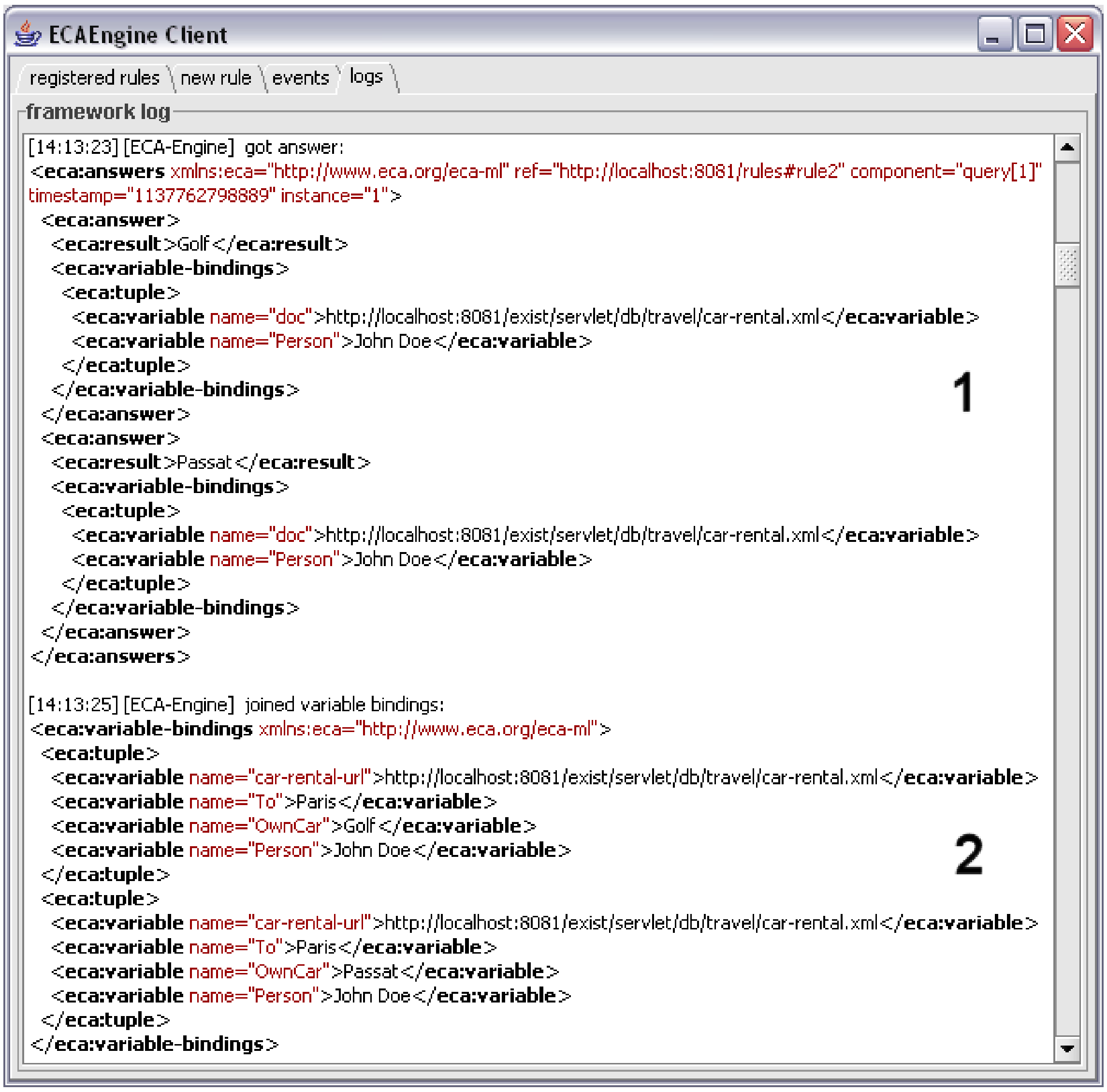


```
[14:13:19] [ECA-Engine]  processing element query[1]
<eca:variable xmlns:eca="http://www.eca.org/eca-ml" name="OwnCar">
 <eca:query>
  <eca:input-variable name="car-rental-url" use="doc" />
  <eca:input-variable name="Person" />
  <eca:opaque lang="http://www.w3.org/XQuery">
   for $car in doc($doc)/car-rental/customer-cars/car[@owner=$Person]
   return
    $car/text()
  </eca:opaque>
 </eca:query>
</eca:variable>

[14:13:20] [ECA-Engine]  received service description:
<eca:service xmlns:eca="http://www.eca.org/eca-ml" namespace="http://www.w3.org/XQuery">
 <eca:multiple-input>false</eca:multiple-input>
 <eca:send-use>false</eca:send-use>
</eca:service>

[14:13:20] [XQuery-Engine]  got request:
<eca:request xmlns:eca="http://www.eca.org/eca-ml" ref="http://localhost:8081/rules#rule2" component="query[1]"
timestamp="1138095218661" instance="1">
 <eca:query>
  <eca:opaque lang="http://www.w3.org/XQuery">
   for $car in doc($doc)/car-rental/customer-cars/car[@owner=$Person]
   return
    $car/text()</eca:opaque>
 </eca:query>
 <eca:variable-bindings>
  <eca:tuple>
   <eca:variable name="doc">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
   <eca:variable name="Person">John Doe</eca:variable>
  </eca:tuple>
 </eca:variable-bindings>
</eca:request>
```

Figure 5.2: Gathering Information about the Language Processor

eXist Engines. The XML database eXist [5] provides a way to natively store XML documents. In order to query the database, XQuery (implying XPath) can be used. In this thesis, two different wrapper engines are developed:

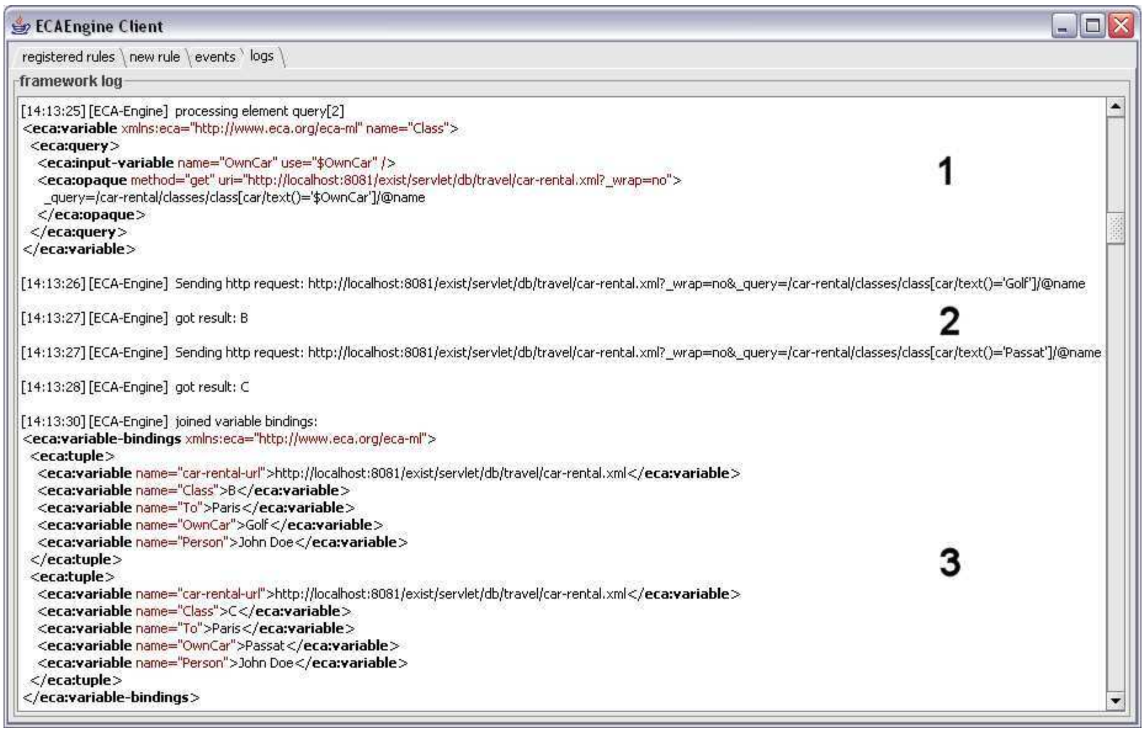
- The *General eXist Engine* is a wrapper around a predefined set of eXist databases that transforms framework-native request messages into correct XQuery expressions by adding the separately given variable bindings using XQuery's let clause. To identify the according eXist database, the content of the doc('/db/...') function is mapped to a predefined URL. After the evaluation, the resulting XML fragment is returned inside an answer message.



```
[14:13:23] [ECA-Engine]  got answer:
<eca:answers xmlns:eca="http://www.eca.org/eca-ml" ref="http://localhost:8081/rules#rule2" component="query[1]"
timestamp="1137762798889" instance="1">
 <eca:answer>
  <eca:result>Golf</eca:result>
  <eca:variable-bindings>
   <eca:tuple>
    <eca:variable name="doc">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
    <eca:variable name="Person">John Doe</eca:variable>
   </eca:tuple>
  </eca:variable-bindings>
 </eca:answer>
 <eca:answer>
  <eca:result>Passat</eca:result>
  <eca:variable-bindings>
   <eca:tuple>
    <eca:variable name="doc">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
    <eca:variable name="Person">John Doe</eca:variable>
   </eca:tuple>
  </eca:variable-bindings>
 </eca:answer>
</eca:answers>

[14:13:25] [ECA-Engine]  joined variable bindings:
<eca:variable-bindings xmlns:eca="http://www.eca.org/eca-ml">
 <eca:tuple>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Golf</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
 <eca:tuple>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Passat</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
</eca:variable-bindings>
```

Figure 5.3: Answer to the First Query Component

- While the *Direct eXist Engine* also extends the given query with the provided variable bindings, it only sends queries to a single predefined eXist database. Furthermore, it returns the resulting XML fragment without modification, i.e. the query *must* return a correctly marked up answer message itself. This allows for the generation of arbitrary answer messages, most notably the faking of an answer returned by a logical language expression (which only contains variable bindings and no functional result).



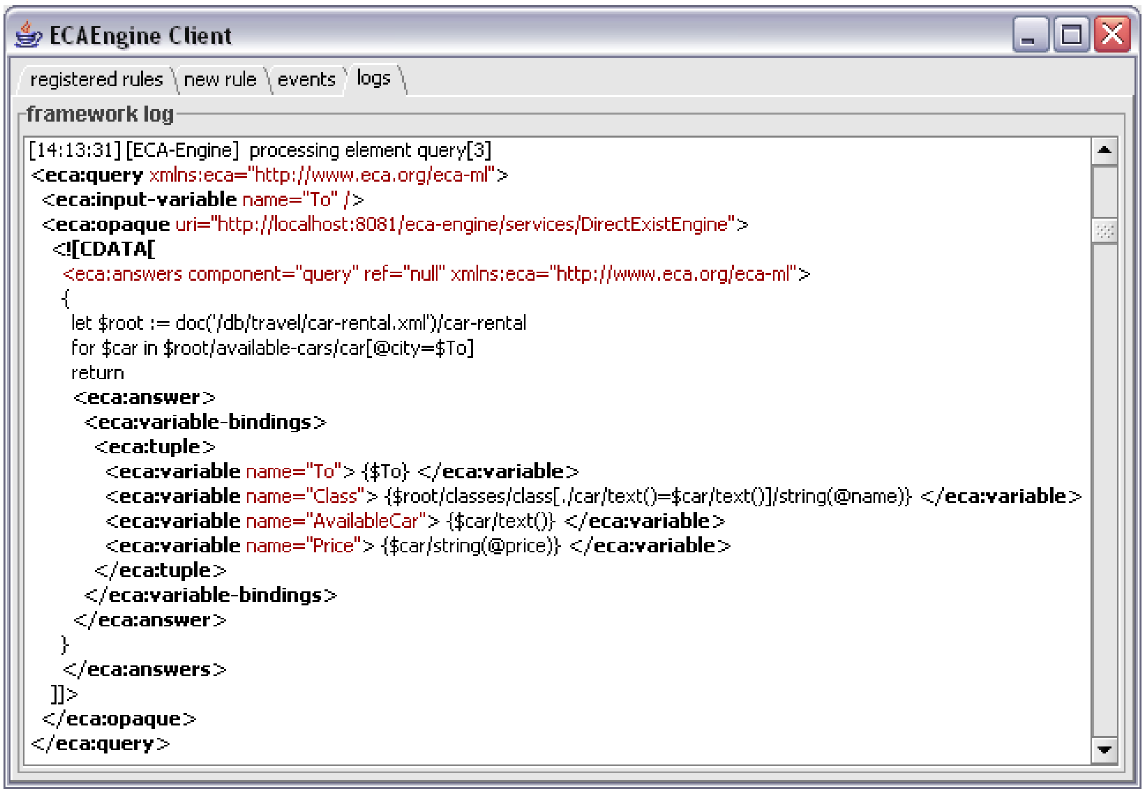
```
[14:13:25] [ECA-Engine]  processing element query[2]
<eca:variable xmlns:eca="http://www.eca.org/eca-ml" name="Class">
 <eca:query>
  <eca:input-variable name="OwnCar" use="$OwnCar" />
  <eca:opaque method="get" uri="http://localhost:8081/exist/servlet/db/travel/car-rental.xml?_wrap=no">
   _query=/car-rental/classes/class[car/text()='$OwnCar']/@name
  </eca:opaque>
 </eca:query>
</eca:variable>

[14:13:26] [ECA-Engine]  Sending http request: http://localhost:8081/exist/servlet/db/travel/car-rental.xml?_wrap=no&_query=/car-rental/classes/class[car/text()='Golf']/@name

[14:13:27] [ECA-Engine]  got result: B

[14:13:27] [ECA-Engine]  Sending http request: http://localhost:8081/exist/servlet/db/travel/car-rental.xml?_wrap=no&_query=/car-rental/classes/class[car/text()='Passat']/@name

[14:13:28] [ECA-Engine]  got result: C

[14:13:30] [ECA-Engine]  joined variable bindings:
<eca:variable-bindings xmlns:eca="http://www.eca.org/eca-ml">
 <eca:tuple>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="Class">B</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Golf</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
 <eca:tuple>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="Class">C</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Passat</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
</eca:variable-bindings>
```

Figure 5.4: Invocation of a Framework-Unaware Service

Example 5.5 *The first query component of the rule (see Figure 5.2[1]) asks for the cars that the customer owns at home using an opaque XQuery expression. The ECA engine retrieves the service description of the language processor (see Figure 5.2[2]) and sends the query including all value combinations for the input variables* car-rental-url *and* Person *(see Figure 5.2[3]).*

The language processor then returns an <eca:answers> *message containing one answer element for each result (see Figure 5.3[1]). At the ECA engine, this result is bound to the variable* OwnCar *and joined with the existing variable bindings (see Figure 5.3[2]). Note that as John Doe owns two cars at home, a* Golf *and a* Passat*, the final variable bindings contain two separate tuples to reflect this fact.*

The next query component maps the cars to classes (sizes) by integrating a framework-unaware service using the GET *method of the* HTTP *protocol (see Figure 5.4[1]). Before invoking this service, the ECA engine combines the* uri *attribute of the* <eca: opaque> *element with its content and replaces the variable* $OwnCar *with its actual value (see Figure 5.4[2]). As there are two tuples of variable bindings, each containing a different value for* OwnCar*, the ECA engine needs to invoke the service twice and assign the result (bound to the variable* Class*) to the correct tuple (see Figure 5.4[3]).*



```
[14:13:31] [ECA-Engine]  processing element query[3]
<eca:query xmlns:eca="http://www.eca.org/eca-ml">
 <eca:input-variable name="To" />
 <eca:opaque uri="http://localhost:8081/eca-engine/services/DirectExistEngine">
  <![CDATA[
   <eca:answers component="query" ref="null" xmlns:eca="http://www.eca.org/eca-ml">
   {
    let $root := doc('/db/travel/car-rental.xml')/car-rental
    for $car in $root/available-cars/car[@city=$To]
    return
    <eca:answer>
     <eca:variable-bindings>
      <eca:tuple>
       <eca:variable name="To"> {$To} </eca:variable>
       <eca:variable name="Class"> {$root/classes/class[./car/text()=$car/text()]/string(@name)} </eca:variable>
       <eca:variable name="AvailableCar"> {$car/text()} </eca:variable>
       <eca:variable name="Price"> {$car/string(@price)} </eca:variable>
      </eca:tuple>
     </eca:variable-bindings>
    </eca:answer>
   }
   </eca:answers>
  ]]>
 </eca:opaque>
</eca:query>
```

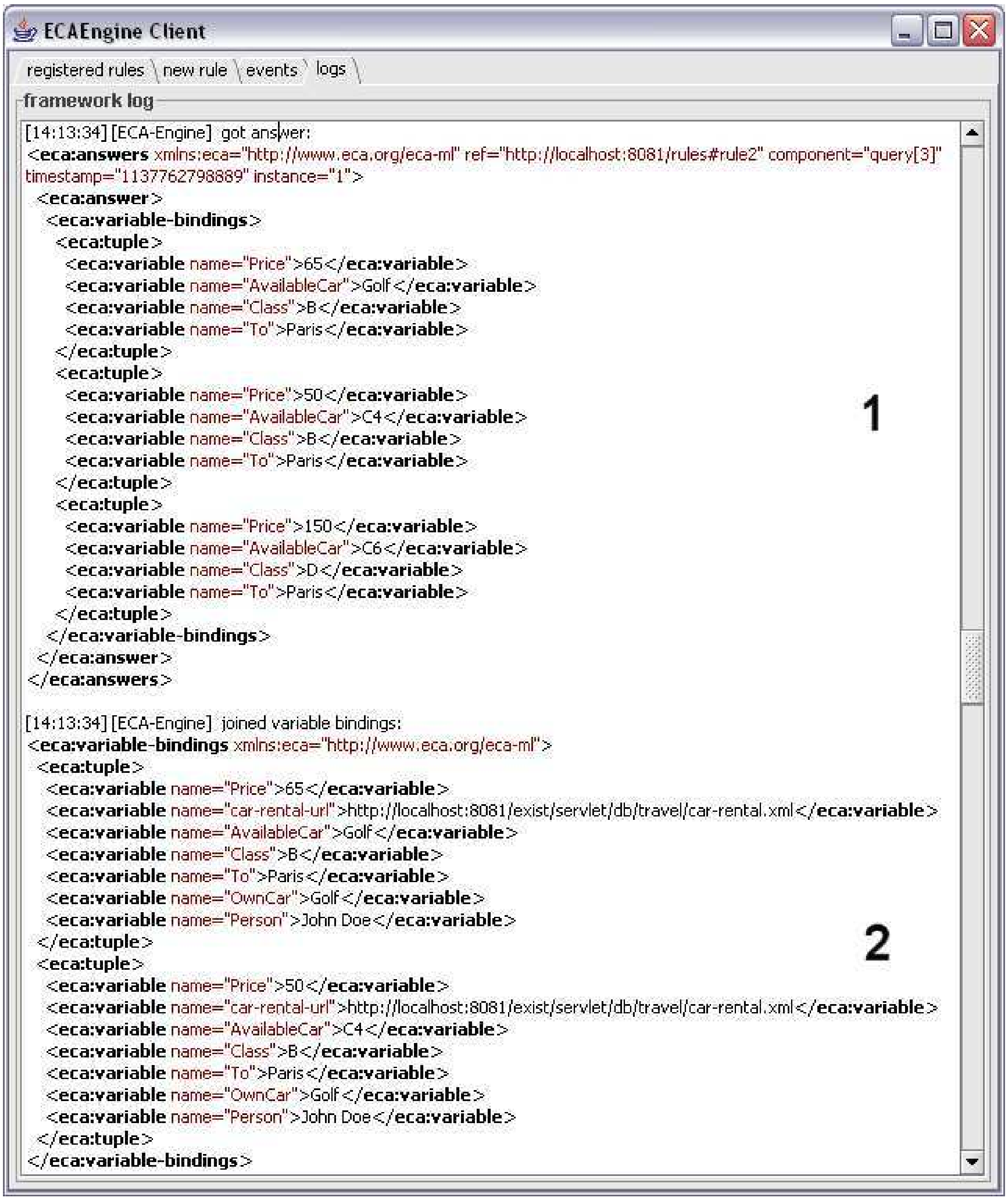
Figure 5.5: Simulation of a Framework-Aware Service using a XQuery Expression

In a further query, a list of all cars that are available at the destination city is retrieved (see Figure 5.5). It is stated against the direct eXist engine *described in Section 5.4 and fakes a framework-aware service by generating an* <eca:answers> *message directly inside the XQuery expression.*

*The classes of the available cars (*B *and* D*) are compared with the classes of the customer's cars (*B *and* C*) as shown in Figure 5.6[1]. In a natural join over the variable* Class*, all tuples containing a car of class* C *or* D *are eliminated and only those with both cars of class* B *remain (see Figure 5.6[2]).*

5.5 Evaluation of the Test and Action Components

When all query components are processed, the evaluation of the test component results in a set of tuples for which the action components will be executed. While the process of evaluation



```
[14:13:34] [ECA-Engine]  got answer:
<eca:answers xmlns:eca="http://www.eca.org/eca-ml" ref="http://localhost:8081/rules#rule2" component="query[3]"
timestamp="1137762798889" instance="1">
 <eca:answer>
  <eca:variable-bindings>
   <eca:tuple>
    <eca:variable name="Price">65</eca:variable>
    <eca:variable name="AvailableCar">Golf</eca:variable>
    <eca:variable name="Class">B</eca:variable>
    <eca:variable name="To">Paris</eca:variable>
   </eca:tuple>
   <eca:tuple>
    <eca:variable name="Price">50</eca:variable>
    <eca:variable name="AvailableCar">C4</eca:variable>
    <eca:variable name="Class">B</eca:variable>
    <eca:variable name="To">Paris</eca:variable>
   </eca:tuple>
   <eca:tuple>
    <eca:variable name="Price">150</eca:variable>
    <eca:variable name="AvailableCar">C6</eca:variable>
    <eca:variable name="Class">D</eca:variable>
    <eca:variable name="To">Paris</eca:variable>
   </eca:tuple>
  </eca:variable-bindings>
 </eca:answer>
</eca:answers>

[14:13:34] [ECA-Engine]  joined variable bindings:
<eca:variable-bindings xmlns:eca="http://www.eca.org/eca-ml">
 <eca:tuple>
  <eca:variable name="Price">65</eca:variable>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="AvailableCar">Golf</eca:variable>
  <eca:variable name="Class">B</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Golf</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
 <eca:tuple>
  <eca:variable name="Price">50</eca:variable>
  <eca:variable name="car-rental-url">http://localhost:8081/exist/servlet/db/travel/car-rental.xml</eca:variable>
  <eca:variable name="AvailableCar">C4</eca:variable>
  <eca:variable name="Class">B</eca:variable>
  <eca:variable name="To">Paris</eca:variable>
  <eca:variable name="OwnCar">Golf</eca:variable>
  <eca:variable name="Person">John Doe</eca:variable>
 </eca:tuple>
</eca:variable-bindings>
```

Figure 5.6: Elimination of Tuples during the Natural Join

is the same as for the query components[3], the handling of the results is more similar to the boolean interpretation of an *XPath* expression. If the result is empty[4] or equals the literal string `false`, the respective tuple of variable bindings is discarded. In any other case, the result is interpreted as `true` and the tuple is kept and later used during the execution of the action components.

When at least one (possibly empty) tuple remains, all action components are executed analogous to the query and test components. With the execution of the last rule component, the evaluation of the rule instance is finished and all state information is discarded.

5.6 Employed Action Engines

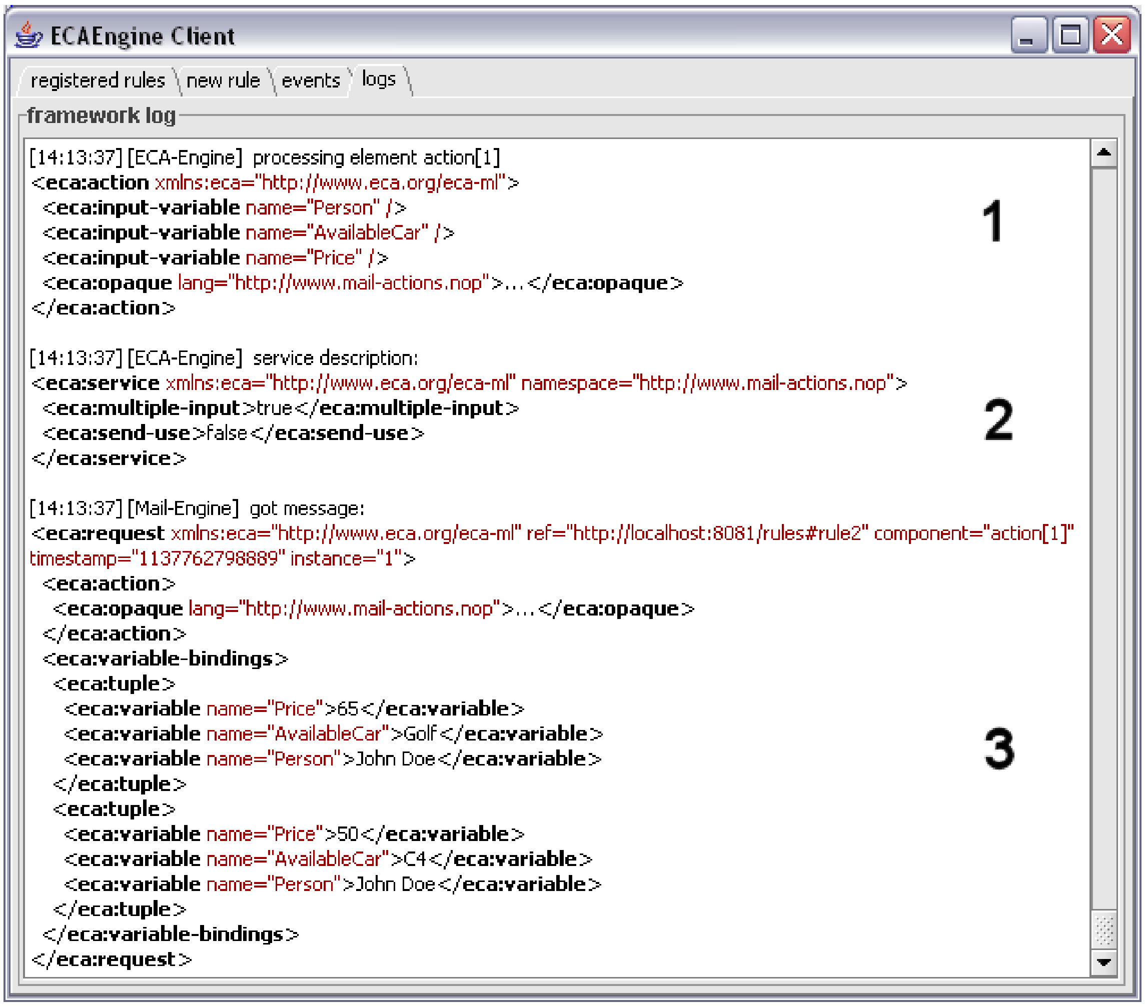
Mail Engine. An exemplary language processor for the action part of a rule is the mail engine. While in a future version of the ECA prototype it will be able to send messages via electronic mail, in this thesis it is only implemented as a simple logging service. When it receives a request message, it forwards it to the framework logging service described in Section 5.8.

Answer Engine. So far, rule execution always started with the detection of an event and resulted in the possible execution of actions. These actions may themselves result in the firing of atomic events (e.g. after a database update). As long as there is no prototypical implementation of an appropriate event detection engine, another mechanism is required that allows for the triggering of rules from inside a rule.

This task is handled by the answer engine that is capable of generating an answers message and forwarding it to the ECA engine.

Example 5.6 *As the exemplary query of this chapter does not contain a query, it is simply omitted and the evaluation of the rule continues with the execution of the first (and only) action component (see Figure 5.7[1]).*

By analyzing the service description of the language processor (see Figure 5.7[2]), the ECA engine learns that it can send the two tuples of variable bindings unchanged. With the sending of this request (see Figure 5.7[3]), the rule evaluation is finished.



```
[14:13:37] [ECA-Engine]  processing element action[1]
<eca:action xmlns:eca="http://www.eca.org/eca-ml">
  <eca:input-variable name="Person" />
  <eca:input-variable name="AvailableCar" />
  <eca:input-variable name="Price" />
  <eca:opaque lang="http://www.mail-actions.nop">...</eca:opaque>
</eca:action>

[14:13:37] [ECA-Engine]  service description:
<eca:service xmlns:eca="http://www.eca.org/eca-ml" namespace="http://www.mail-actions.nop">
  <eca:multiple-input>true</eca:multiple-input>
  <eca:send-use>false</eca:send-use>
</eca:service>

[14:13:37] [Mail-Engine]  got message:
<eca:request xmlns:eca="http://www.eca.org/eca-ml" ref="http://localhost:8081/rules#rule2" component="action[1]"
timestamp="1137762798889" instance="1">
  <eca:action>
    <eca:opaque lang="http://www.mail-actions.nop">...</eca:opaque>
  </eca:action>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="Price">65</eca:variable>
      <eca:variable name="AvailableCar">Golf</eca:variable>
      <eca:variable name="Person">John Doe</eca:variable>
    </eca:tuple>
    <eca:tuple>
      <eca:variable name="Price">50</eca:variable>
      <eca:variable name="AvailableCar">C4</eca:variable>
      <eca:variable name="Person">John Doe</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:request>
```

Figure 5.7: Executing the Action Component

5.7 Internal Functionality of the ECA Engine

While the ECA engine represents a single component of the framework when seen from the outside, the prototype developed in this thesis is divided into two parts. The core ECA engine is responsible for the evaluation of rules at a declarative level.

As illustrated in Figure 5.8, the ECA engine employs a generic service, called *Generic Request Handler (GRH)*, to handle the actual invocation of an appropriate language processor. In

[3]Note that in a production environment the test component will likely be evaluated locally at the ECA engine for optimized performance.
[4]Whitespace at the beginning and the end of the result is ignored.

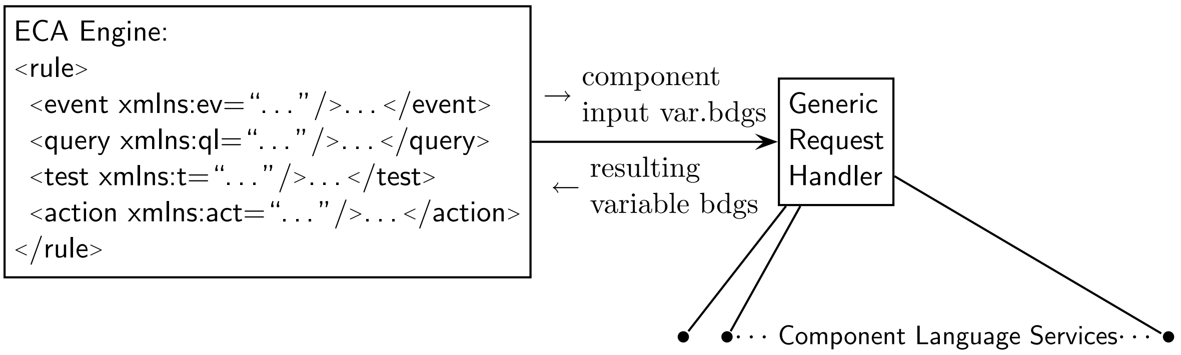

ECA Engine:
<rule>
<event xmlns:ev="..."/>...</event>
<query xmlns:ql="..."/>...</query>
<test xmlns:t="..."/>...</test>
<action xmlns:act="..."/>...</action>
</rule>
→ component input var.bdgs
← resulting variable bdgs
Generic Request Handler
··· Component Language Services···

Figure 5.8: Internal Architecture of the ECA Engine (from [3])

addition to the component, it provides the necessary variable bindings to the GRH and receives the variable bindings resulting from the query.

Thus, from the viewpoint of the core ECA engine, all language processors inside the Web are not only framework-aware but also directly return variable bindings.

To provide this abstraction, the GRH is able to transform the native messages of the framework into messages that are understood by the specific services. Furthermore, it can iterate over sets of variable bindings and replenish the resulting bindings, if applicable.

Example 5.7 *Consider again Figure 5.4. While it only shows that the ECA engine invokes the framework-unaware service twice and joins the results, internally the following things happen:*

1. *The ECA engine sends the component and variable bindings to the GRH.*
2. *The GRH iterates over the tuples and sends the HTTP request once for* Golf *and once for* Passat*.*
3. *The GRH generates the following variable bindings and sends them to the ECA engine:*

```
<eca:variable-bindings>
  <eca:tuple>
    <eca:variable name="Class">C</eca:variable>
    <eca:variable name="OwnCar">Passat</eca:variable>
  </eca:tuple>
  <eca:tuple>
    <eca:variable name="Class">B</eca:variable>
    <eca:variable name="OwnCar">Golf</eca:variable>
  </eca:tuple>
</eca:variable-bindings>
```

4. *The ECA engine joins the resulting bindings with its own.*

5.8 Additional Infrastructure

Besides the framework components described so far, a few additional components are required during the development phase of the ECA prototype.

Message Broker. In the future it will be possible to state queries and actions against "the Web" as a whole without previous knowledge of where the appropriate information system is actually located. The mechanisms of the Semantic Web will then ensure that the query (or action) is sent to the correct node(s). As this functionality is not available yet, it must be simulated by a dedicated message broker that holds a static mapping of namespaces (URIs) to the respective services.

Logging Engine. As the individual components of the framework may be located at different nodes inside the web it is reasonable for the prototype to have a central logging engine. This allows to cumulate all logging messages at one place and make them available to the client for demonstration purposes.

ECA Engine Client. While the framework components mentioned so far represent the center of the ECA prototype developed in this thesis, some important aspects are not yet dealt with:

- An administration tool is needed to be able to register and deregister rules at the ECA engine.
- For demonstration purposes there must be a tool for viewing the framework log.
- As no event detection engine is currently available, the detection of an event must be simulated by sending a faked <eca:answers> message to the ECA engine.

These requirements have to be met by the ECA engine client.

6 Implementation

This chapter presents the actual implementation of the ECA prototype. At first, the employed technologies are described, followed by a general outline of the architecture. Finally, the implementation of each framework component is explained in detail.

6.1 Employed Technologies

The complete protoype of the ECA framework is implemented in Java. The individual framework components exist as web services and use the SOAP protocol for communication[1].

Java and the Spring Framework. The object-oriented programming language Java was invented by Sun and allows for the development of software that is independent of the underlying harware architecture and operation system.

For the easy configuration of the framework components, the Spring framework is used. Spring acts as a container around the individual framework components and provides the necessary dependencies by the use of the *dependency injection* concept. Furthermore, it simplifies the setup and use of web services that are described in the next section.

For more information about Java see [16], the Spring is framework described in [7] and [15].

Web Services with Apache Axis. Since the components of the ECA framework represent arbitrary nodes inside the Semantic Web, a communication method is required that does not depend on a specific operating system or programming language. Thus, the *Simple Object Access Protocol (SOAP)* was chosen as it allows for a complete abstraction of the underlying hard- and software.

A popular implementation of SOAP in Java is Apache Axis which is also used in this implementation. More information on Axis can be found at [2].

[1]Due to the fact, that all messages are already marked up in XML it is likely that SOAP will be replaced by plain HTTP-POST requests in the future.

6.2 Architecture

The ECA prototype consists of two different kinds of classes. On the one hand, there are aspects that several or all of the framework components must deal with, e.g. the handling of variables or the analyzation and generation of messages. On the other hand, each of the components is an individual web service and has its own needs.

To represent this, the classes are divided into common ones and component specific ones. The common classes are kept inside the common package[2], the specific ones in subpackages of engines.

6.3 Common Classes

All classes that are shared by multiple framework components are kept below the subpackage common, most notably classes for the handling of variable bindings and utility classes. This makes it easy to distribute the individual framework components to different web nodes, as they only depend one additional java archive.

6.3.1 Variable Bindings

The classes inside the package common.variables (see Figure 6.1)[3] implement the concept of variable bindings as described in Section 4.1. The central class is called VariableBindings and maintains a list of instances of the class Tuple. It offers methods for joining its own tuples with those of another instance and for finding all existent combinations of input variables inside its own tuples.

The class Tuple maintains a map of variable names to the according instances of subclasses of VariableBinding[4]. A Tuple can be cloned, compared to or joined with another Tuple.

A subclass of VariableBinding then holds the actual value of the logical variable and can also be cloned and compared and may additionally be transformed to its string representation.

6.3.2 Utility Classes

The package common.util (see Figure 6.2) contains - as the name implies - a collection of helper classes that are useful for several framework components. The most generic class is

[2]Note that all packages have the prefix de.uni_goettingen.informatik.dbis.

[3]Note that in all class diagrams throughout this chapter, the declaration of the getter and setter methods for private attributes is omitted.

[4]These are EmptyVariableBinding, StringVariableBinding and XMLVariableBinding for now. This list can later be extended to implement a more complex type system.

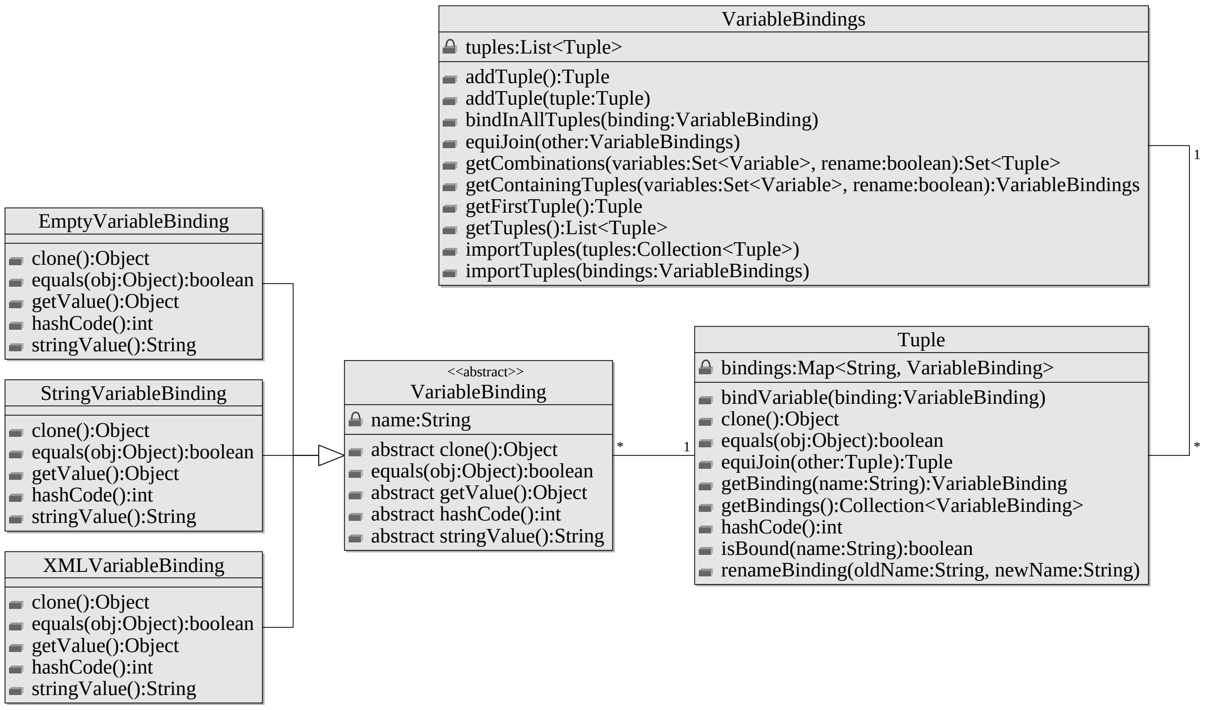

VariableBindings
tuples:List<Tuple>
addTuple():Tuple
addTuple(tuple:Tuple)
bindInAllTuples(binding:VariableBinding)
equiJoin(other:VariableBindings)
getCombinations(variables:Set<Variable>, rename:boolean):Set<Tuple>
getContainingTuples(variables:Set<Variable>, rename:boolean):VariableBindings
getFirstTuple():Tuple
getTuples():List<Tuple>
importTuples(tuples:Collection<Tuple>)
importTuples(bindings:VariableBindings)
1
EmptyVariableBinding
clone():Object
equals(obj:Object):boolean
getValue():Object
hashCode():int
stringValue():String
StringVariableBinding
clone():Object
equals(obj:Object):boolean
getValue():Object
hashCode():int
stringValue():String
XMLVariableBinding
clone():Object
equals(obj:Object):boolean
getValue():Object
hashCode():int
stringValue():String
<<abstract>>
VariableBinding
name:String
abstract clone():Object
equals(obj:Object):boolean
abstract getValue():Object
abstract hashCode():int
abstract stringValue():String
*
1
Tuple
bindings:Map<String, VariableBinding>
bindVariable(binding:VariableBinding)
clone():Object
equals(obj:Object):boolean
equiJoin(other:Tuple):Tuple
getBinding(name:String):VariableBinding
getBindings():Collection<VariableBinding>
hashCode():int
isBound(name:String):boolean
renameBinding(oldName:String, newName:String)
*

Figure 6.1: Class Diagram for Variable Bindings

XMLUtils that is able to serialize and deserialize XML documents to and from their string representation. This feature is for example used by the class VariableBindingsXMLHelper to implement de-/serialization for variable bindings.

Methods for the analyzation and composition of framework messages like <eca:request> and <eca:answers> are provided by the class MessageXMLHelper. It uses the class MessageAttributes that holds all attributes that are required for the unique identification of a rule instance [5].

The class ServiceDescription is also part of this package. It is implemented as a plain object that has one attribute for every feature of a language processor and the respective getter and setter methods. Analogous to the variable bindings, the class ServiceDescriptionXMLHelper implements de-/serialization for the service description[6].

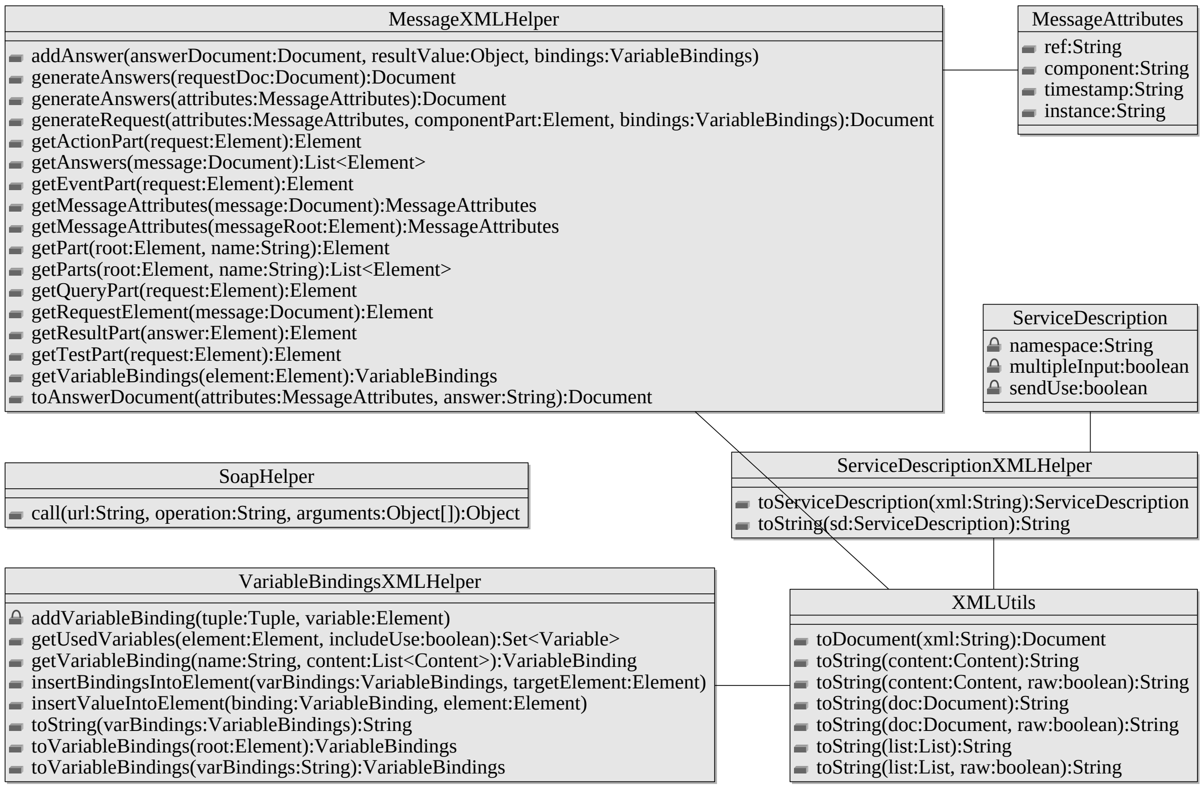

MessageXMLHelper
addAnswer(answerDocument:Document, resultValue:Object, bindings:VariableBindings)
generateAnswers(requestDoc:Document):Document
generateAnswers(attributes:MessageAttributes):Document
generateRequest(attributes:MessageAttributes, componentPart:Element, bindings:VariableBindings):Document
getActionPart(request:Element):Element
getAnswers(message:Document):List<Element>
getEventPart(request:Element):Element
getMessageAttributes(message:Document):MessageAttributes
getMessageAttributes(messageRoot:Element):MessageAttributes
getPart(root:Element, name:String):Element
getParts(root:Element, name:String):List<Element>
getQueryPart(request:Element):Element
getRequestElement(message:Document):Element
getResultPart(answer:Element):Element
getTestPart(request:Element):Element
getVariableBindings(element:Element):VariableBindings
toAnswerDocument(attributes:MessageAttributes, answer:String):Document
MessageAttributes
ref:String
component:String
timestamp:String
instance:String
ServiceDescription
namespace:String
multipleInput:boolean
sendUse:boolean
SoapHelper
call(url:String, operation:String, arguments:Object[]):Object
ServiceDescriptionXMLHelper
toServiceDescription(xml:String):ServiceDescription
toString(sd:ServiceDescription):String
VariableBindingsXMLHelper
addVariableBinding(tuple:Tuple, variable:Element)
getUsedVariables(element:Element, includeUse:boolean):Set<Variable>
getVariableBinding(name:String, content:List<Content>):VariableBinding
insertBindingsIntoElement(varBindings:VariableBindings, targetElement:Element)
insertValueIntoElement(binding:VariableBinding, element:Element)
toString(varBindings:VariableBindings):String
toVariableBindings(root:Element):VariableBindings
toVariableBindings(varBindings:String):VariableBindings
XMLUtils
toDocument(xml:String):Document
toString(content:Content):String
toString(content:Content, raw:boolean):String
toString(doc:Document):String
toString(doc:Document, raw:boolean):String
toString(list:List):String
toString(list:List, raw:boolean):String

Figure 6.2: Class Diagram for Utility Classes

6.4 ECA Engine

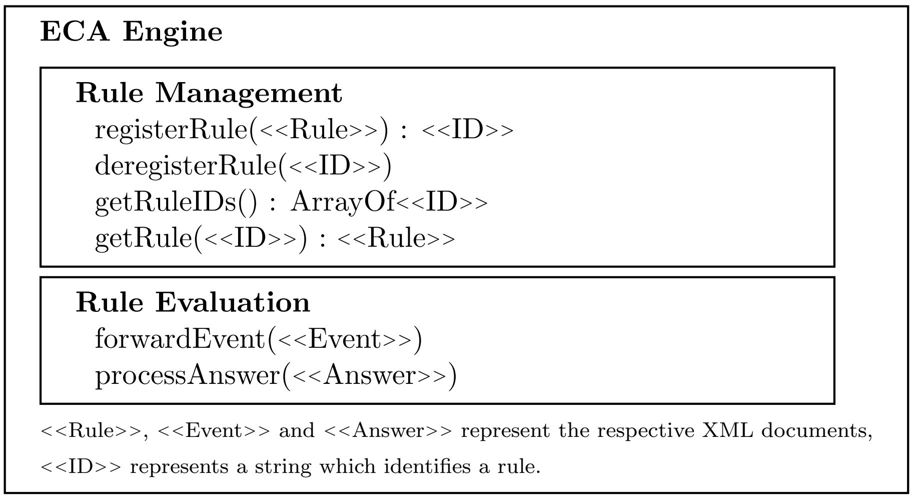
6.4.1 Communication Interface of the ECA Engine

The communication interface of the ECA engine is shown in Figure 6.3. From the perspective of the client, the methods for rule management are most important. When a client registers a rule (in the respective XML markup), the method registerRule returns an identifier that the client may later use to retrieve (getRule) or deregister (deregisterRule) the rule.

During the phase of event detection, the client may need to communicate atomic events to the event detection engine where the event component of the rule was registered[7]. To allow for this communication, the ECA engine provides the method forwardEvent that forwards the given event (an XML fragment) to all known event detection engines.

Finally, when an event was detected, the event detection engine needs to notify the ECA

[5]This is explained in detail in Section 4.3.2.
[6]The format of a SD is explained in Section 4.3.3.
[7]This is only required when the event detection engine is not able to find out about the relevant events itself.



ECA Engine

Rule Management
registerRule(<<Rule>>) : <<ID>>
deregisterRule(<<ID>>)
getRuleIDs() : ArrayOf<<ID>>
getRule(<<ID>>) : <<Rule>>

Rule Evaluation
forwardEvent(<<Event>>)
processAnswer(<<Answer>>)

<<Rule>>, <<Event>> and <<Answer>> represent the respective XML documents,
<<ID>> represents a string which identifies a rule.

Figure 6.3: Communication Interface of the ECA Engine

engine. The method processAnswer therefore provides a way to do this. Furthermore, it is used during the subsequent evaluation of the rule in case the evaluation of a query or test component is happening asynchronously.

Example 6.1 *When a client wants to register a rule at the ECA engine, it may send the following SOAP message to it:*

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:registerRule
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://soapinterop.org/">
      <eca:rule xmlns:eca="http://www.eca.org/eca-ml">
        <eca:event>
          ...
        </eca:event>
        <eca:query>
          ...
        </eca:query>
        ...
        ...
      </eca:rule>
    </ns1:registerRule>
  </soapenv:Body>
</soapenv:Envelope>
```

In return, the ECA engine would provide the id it assigned to the rule:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:registerRuleResponse
```

```
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://soapinterop.org/">
      <registerRuleReturn xsi:type="soapenc:string" xmlns:soapenc="...">
        rule517
      </registerRuleReturn>
    </ns1:registerRuleResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

When the event detection engine notifies the ECA engine about an event occurrence (consider again Example 5.2 on page 32), it sends the following SOAP message:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:handleAnswer
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://soapinterop.org/">
      <eca:answers xmlns:eca="http://www.eca.org/eca-ml" ref="http://..." component="event">
        <eca:answer>
          <eca:result>
            <booking person="John Doe" from="Munich" to="Paris"/>
          </eca:result>
          <eca:variable-bindings>
            <eca:tuple>
              <eca:variable name="Person">
                John Doe
              </eca:variable>
              <eca:variable name="To">
                Paris
              </eca:variable>
            </eca:tuple>
          </eca:variable-bindings>
        </eca:answer>
      </eca:answers>
    </ns1:handleAnswer>
  </soapenv:Body>
</soapenv:Envelope>
```

Obviously, the ECA engine does not return an answer in return.

6.4.2 Architecture of the ECA Engine

The ECA engine is the central part of the framework prototype and is kept in the package engines.eca (see Figure 6.4). Its main class, ECAEngine, provides all necessary methods for rule management and rule evaluation.

Behind the scenes, it does not handle any task itself but uses the RuleManager or another helper class for this purpose. To provide the necessary dependencies at runtime, it passes an instance of HelperContext along with each call. As shown in Figure 6.4, the HelperContext contains references to several objects that are needed for the management of rules and the rule evaluation. These will be explained in the following sections.

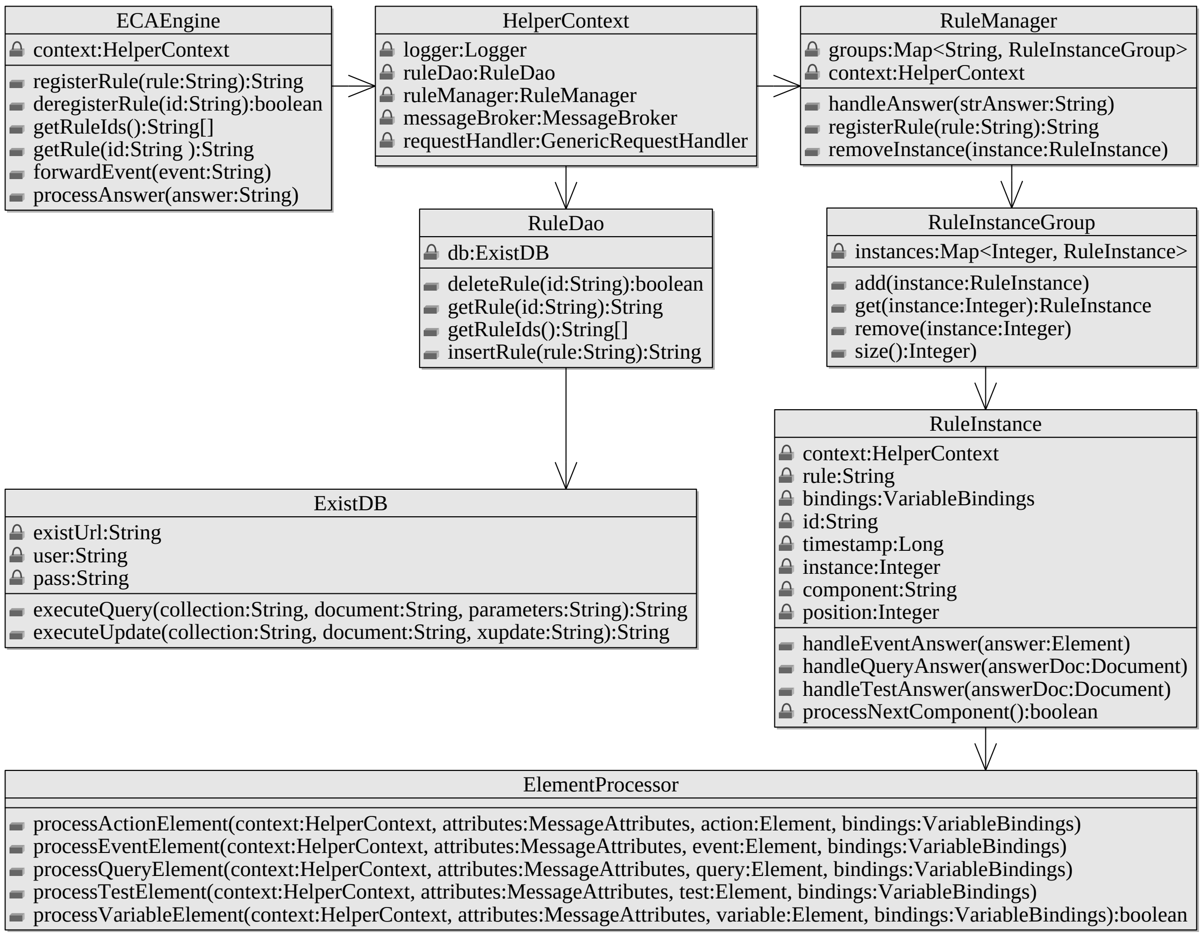

ECAEngine
context:HelperContext
registerRule(rule:String):String
deregisterRule(id:String):boolean
getRuleIds():String[]
getRule(id:String):String
forwardEvent(event:String)
processAnswer(answer:String)
HelperContext
logger:Logger
ruleDao:RuleDao
ruleManager:RuleManager
messageBroker:MessageBroker
requestHandler:GenericRequestHandler
RuleManager
groups:Map<String, RuleInstanceGroup>
context:HelperContext
handleAnswer(strAnswer:String)
registerRule(rule:String):String
removeInstance(instance:RuleInstance)
RuleDao
db:ExistDB
deleteRule(id:String):boolean
getRule(id:String):String
getRuleIds():String[]
insertRule(rule:String):String
RuleInstanceGroup
instances:Map<Integer, RuleInstance>
add(instance:RuleInstance)
get(instance:Integer):RuleInstance
remove(instance:Integer)
size():Integer
RuleInstance
context:HelperContext
rule:String
bindings:VariableBindings
id:String
timestamp:Long
instance:Integer
component:String
position:Integer
handleEventAnswer(answer:Element)
handleQueryAnswer(answerDoc:Document)
handleTestAnswer(answerDoc:Document)
processNextComponent():boolean
ExistDB
existUrl:String
user:String
pass:String
executeQuery(collection:String, document:String, parameters:String):String
executeUpdate(collection:String, document:String, xupdate:String):String
ElementProcessor
processActionElement(context:HelperContext, attributes:MessageAttributes, action:Element, bindings:VariableBindings)
processEventElement(context:HelperContext, attributes:MessageAttributes, event:Element, bindings:VariableBindings)
processQueryElement(context:HelperContext, attributes:MessageAttributes, query:Element, bindings:VariableBindings)
processTestElement(context:HelperContext, attributes:MessageAttributes, test:Element, bindings:VariableBindings)
processVariableElement(context:HelperContext, attributes:MessageAttributes, variable:Element, bindings:VariableBindings):boolean

Figure 6.4: Class Diagram for the ECA Engine

6.4.3 Rule Management

Methods for the registration and deregistration of rules are located at the RuleManager which implements the process described in Section 5.1. When a rule is registered, it is stored inside the native XML database eXist. Thus, the class RuleDao[8] that is responsible for storing the rule uses an instance of ExistDB for connecting with the database and executing queries and updates.

As the methods getRuleIds and getRule require no complex logic, the ECAEngine calls the respective methods of RuleDao directly.

[8]DAO stands for *Data Access Object*.

6.4.4 Rule Evaluation

The process of rule evaluation is described in detail in Chapter 5. At the level of the implementation, rule evaluation starts when the ECAEngine calls the method handleAnswer of the RuleManager with a message that is an answer to the event part of a rule.

If it not already exists, the RuleManager then creates a new instance of the class RuleInstance-Group and stores it along with the actual time point.

Next, a new instance of RuleInstance is created and added to the RuleInstanceGroup[9]. The class RuleInstance controls the evaluation of the rule and stores all necessary information like the actual variable bindings and the position of the actually evaluated rule component.

Finally, the RuleManager calls the method handleEventAnswer at the newly created RuleInstance. This method builds up all static variable bindings that are defined before the rule's event part and binds the event itself, if appropriate. It then calls the private method process-NextElement that is responsible for determining the next rule component to be evaluated.

Depending on the name of the respective element[10], processNextComponent calls the appropriate method of ElementProcessor. The ElementProcessor then determines the required variable bindings, generates an appropriate <eca:request> message and sends it to the generic request handler.

At this point the evaluation of the rule is suspended, as the ECA engine has to wait for the answer to its request. When this is available, it is delgated to the correct RuleInstance which then joins the resulting variable bindings with its own and again determines the next rule component by calling processNextElement.

This process is repeated until the last element of the rule has been evaluated. Afterwards the RuleInstance is removed from the RuleInstanceGroup which itself is removed if it does not contain any more active RuleInstances.

6.5 Generic Request Handler

The generic request handler inside the package engine.generic (see Figure 6.5) is responsible for the identification and invocation of the correct language processor[11] and the preprocessing of the resulting answer.

While the ECA engine and the GRH communicate asynchronously, the communication between the GRH and the language processors is currently implemented in a synchronous man-

[9]If the message contains several <eca:answer> parts, it is split up into the according number of RuleInstance objects.
[10]This can be <eca:variable>, <eca:query>, <eca:test> or <eca:action>.
[11]In case of a non-opaque expression this task is passed on to the message broker.

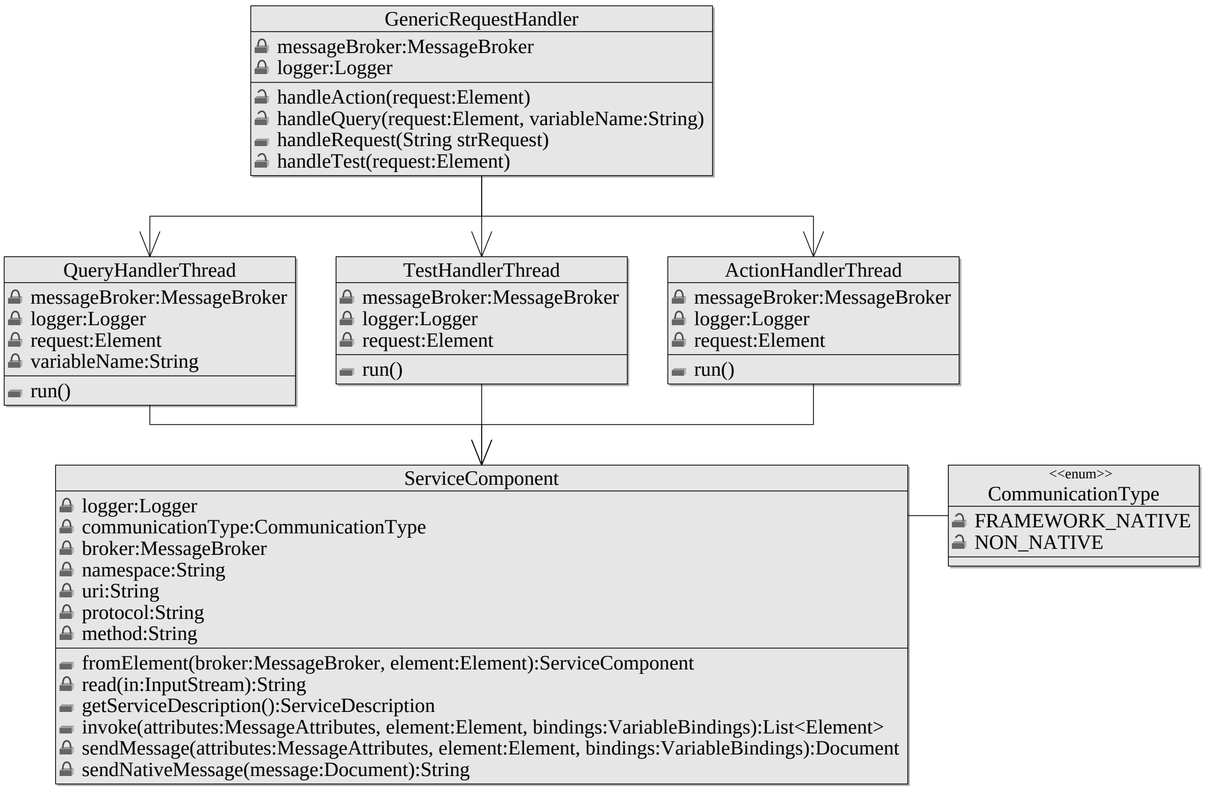
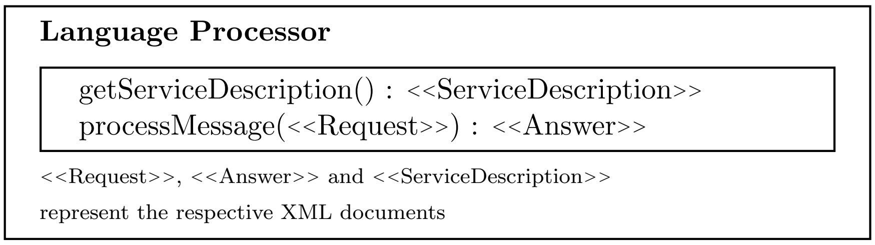

GenericRequestHandler
messageBroker:MessageBroker
logger:Logger
handleAction(request:Element)
handleQuery(request:Element, variableName:String)
handleRequest(String strRequest)
handleTest(request:Element)
QueryHandlerThread
messageBroker:MessageBroker
logger:Logger
request:Element
variableName:String
run()
TestHandlerThread
messageBroker:MessageBroker
logger:Logger
request:Element
run()
ActionHandlerThread
messageBroker:MessageBroker
logger:Logger
request:Element
run()
ServiceComponent
logger:Logger
communicationType:CommunicationType
broker:MessageBroker
namespace:String
uri:String
protocol:String
method:String
fromElement(broker:MessageBroker, element:Element):ServiceComponent
read(in:InputStream):String
getServiceDescription():ServiceDescription
invoke(attributes:MessageAttributes, element:Element, bindings:VariableBindings):List<Element>
sendMessage(attributes:MessageAttributes, element:Element, bindings:VariableBindings):Document
sendNativeMessage(message:Document):String
<<enum>>
CommunicationType
FRAMEWORK_NATIVE
NON_NATIVE

Figure 6.5: Class Diagram for the Generic Request Handler

ner. When the GRH receives a request from the ECA engine, the method handleRequest of the class GenericRequestHandler is called. Based on the component attribute of the request message a new thread is started, running an according instance of either QueryHandlerThread, TestHandlerThread or ActionHandlerThread.

In any case, the rule component and the required variable bindings are extracted from the message. Afterwards, the appropriate language processor is determined and based on its service description, the variable bindings that will be sent to the processor are calculated. The latter is dependent on the evaluated component:

- In case of a *query* component, the set of variables communicated to the language processor may be any subset of all *used* variables that at least contains all *input* variables. The variable bindings finally sent then consist of every existing value combination.
- In the *test* component there are no other *used* variables than *input variables* and thus, the variable bindings can be sent without modification.



Language Processor

getServiceDescription() : <<ServiceDescription>>
processMessage(<<Request>>) : <<Answer>>

<<Request>>, <<Answer>> and <<ServiceDescription>>
represent the respective XML documents

Figure 6.6: Communication Interface of Language Processors

- An *action* component may only define *input* variables. Contrary to the former two, not every value combination is sent to the language processor but every tuple.

At this point, the actual invocation of the language processor takes place which is encapsulated inside the class ServiceComponent. Its invoke method takes the MessageAttributes, the rule component (as an XML element) and the VariableBindings as arguments. Depending on the rule component (opaque vs. non-opaque) it correctly invokes the language processor and returns an accurately marked up <eca:answers> message[12].

The handling of the answer message is then different for the three component types (every class has its own private handleXAnswers method, where X stands for the according component):

- The resulting variable bindings of a *query* component are simply joined with the ones sent by the ECA engine.
- In the contrary, the result of a *test* component has a special meaning. If it is empty of contains the string "false", the according tuple is removed from the variable bindings, otherwise it is kept.
- An *action* component does not yield an answer.

While the evaluation of an *action* component is finished at this point, a *query* or *test* component requires the sending of a <eca:answers> message back to the ECA engine that contains the resulting variable bindings.

6.6 An Exemplary Language Processor: XPath Engine

6.6.1 Communication Interface of a Language Processor

The communication interface of language processors is illustrated in Figure 6.6. Before the ECA engine can invoke a language processor, it has to gather information about the processor's

[12]Note that this is also true for language processors that are not framework-aware.

capabilities wrt. the handling of variable bindings (see Section 4.3.3). It therefore calls the method getServiceDescription that returns the required information (marked up in XML).

Afterwards, it can send a request message to the language processor by calling the method processMessage. In return it will receive the result of the evaluation in form of an answer message (again, both messages are marked up in XML).

In the future, a language processor may have the need of delegating the evaluation of a part of the rule component to another evaluation service. The method processMessage is therefore intentionally not named processRequest, since the language processor will then be able to receive answer messages using this method without violating the naming convention.

Example 6.2 *Consider again Example 5.5 on page 38. When the ECA engine evaluates the first query component, it calls* getServiceDescription *at the language processor (Figure 5.2[2]). Afterwards, the method* processMessage *is used to request the evaluation of the query component (Figure 5.2[3]).*

The respective SOAP message that is send to invoke the language processor looks as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:processMessage
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://soapinterop.org/">
      <eca:request xmlns:eca="http://www.eca.org/eca-ml" ref="..."
        component="query[1]"timestamp="..."instance="...">
        <eca:query>
          <eca:opaque lang="http://www.w3.org/XQuery">
            for $car in doc($doc)/car-rental/customer-cars/car[@owner=$Person]
            return
              $car/text()</eca:opaque>
        </eca:query>
        <eca:variable-bindings>
          <eca:tuple>
            <eca:variable name="doc">
              http://localhost:8081/exist/servlet/db/travel/car-rental.xml
            </eca:variable>
            <eca:variable name="Person">
              John Doe
            </eca:variable>
          </eca:tuple>
        </eca:variable-bindings>
      </eca:request>
    </ns1:processMessage>
  </soapenv:Body>
</soapenv:Envelope>
```

After the evaluation, the language processor returns the following SOAP message:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
  <ns1:processMessageResponse
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="http://soapinterop.org/">
    <eca:answers xmlns:eca="http://www.eca.org/eca-ml" ref="..."
      component="query[1]"timestamp="..."instance="...">
      <eca:answer>
        <eca:result>Golf</eca:result>
        <eca:variable-bindings>
          <eca:tuple>
            <eca:variable name="doc">
              http://localhost:8081/exist/servlet/db/travel/car-rental.xml
            </eca:variable>
            <eca:variable name="Person">
              John Doe
            </eca:variable>
          </eca:tuple>
        </eca:variable-bindings>
      </eca:answer>
      <eca:answer>
        <eca:result>Passat</eca:result>
        <eca:variable-bindings>
          <eca:tuple>
            <eca:variable name="doc">
              http://localhost:8081/exist/servlet/db/travel/car-rental.xml
            </eca:variable>
            <eca:variable name="Person">
              John Doe
            </eca:variable>
          </eca:tuple>
        </eca:variable-bindings>
      </eca:answer>
    </eca:answers>
  </ns1:processMessageResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

After describing the abstract interface of a language processor in this section, the next section shows a possible implementation.

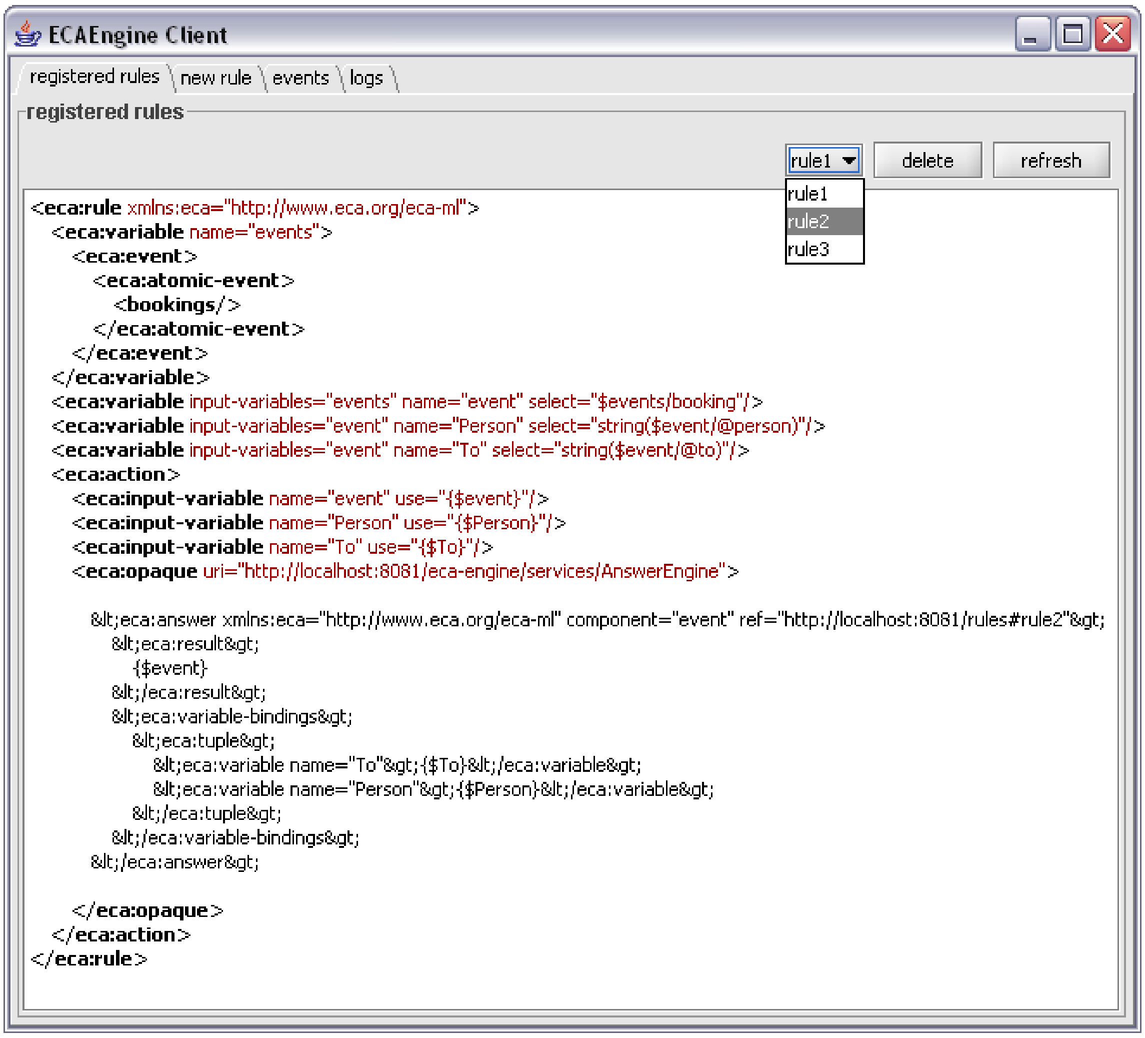
6.6.2 Architecture of the XPath Engine

As shown in the last section, a language processor must implement two methods, getServiceDescription and processMessage, of which the first is simply serializing and returning a ServiceDescription object described in Section 6.3.2.

The processMessage method is now explained in detail. At first, the <eca:request> message is converted to an XML document and analyzed, i.e. the rule component and the required variable bindings are extracted. Before the actual evaluation starts, an empty <eca:answers> message is created, that will later be filled with the results.

Next, the private method query is called for every tuple of variable bindings[13]. This method prepares the given XPath expression and provides the internal evaluator with the required

[13]Note that this service can obviously handle multiple tuples.



```
<eca:rule xmlns:eca="http://www.eca.org/eca-ml">
  <eca:variable name="events">
    <eca:event>
      <eca:atomic-event>
        <bookings/>
      </eca:atomic-event>
    </eca:event>
  </eca:variable>
  <eca:variable input-variables="events" name="event" select="$events/booking"/>
  <eca:variable input-variables="event" name="Person" select="string($event/@person)"/>
  <eca:variable input-variables="event" name="To" select="string($event/@to)"/>
  <eca:action>
    <eca:input-variable name="event" use="{$event}"/>
    <eca:input-variable name="Person" use="{$Person}"/>
    <eca:input-variable name="To" use="{$To}"/>
    <eca:opaque uri="http://localhost:8081/eca-engine/services/AnswerEngine">

      &lt;eca:answer xmlns:eca="http://www.eca.org/eca-ml" component="event" ref="http://localhost:8081/rules#rule2"&gt;
        &lt;eca:result&gt;
          {$event}
        &lt;/eca:result&gt;
        &lt;eca:variable-bindings&gt;
          &lt;eca:tuple&gt;
            &lt;eca:variable name="To"&gt;{$To}&lt;/eca:variable&gt;
            &lt;eca:variable name="Person"&gt;{$Person}&lt;/eca:variable&gt;
          &lt;/eca:tuple&gt;
        &lt;/eca:variable-bindings&gt;
      &lt;/eca:answer&gt;

    </eca:opaque>
  </eca:action>
</eca:rule>
```

Figure 6.7: Managing Rules with the ECA Engine Client

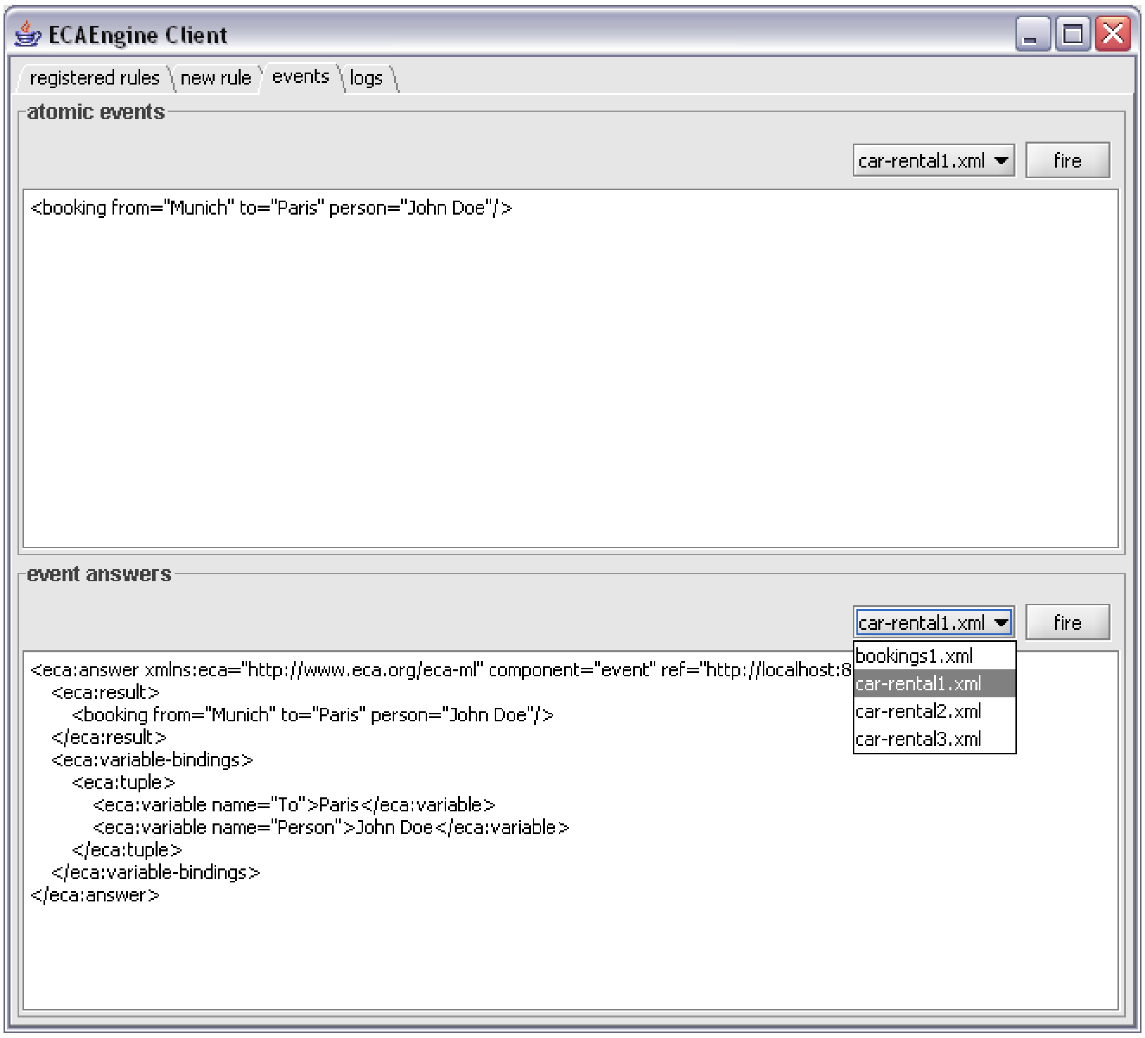
variable bindings. The result of the expression is then returned to the processMessage method which inserts the appropriate number of <eca:answer> elements into the <eca:answers> message.

Finally, the <eca:answers> message is returned to the generic request handler.

6.7 ECA Engine Client

The graphical ECA engine client is implemented in Swing and consists of three logical parts.

1. The classes RegisteredRulesPanel (see Figure 6.7) and NewRulePanel allow for the admin-



```
<booking from="Munich" to="Paris" person="John Doe"/>
```

```
<eca:answer xmlns:eca="http://www.eca.org/eca-ml" component="event" ref="http://localhost:8
  <eca:result>
    <booking from="Munich" to="Paris" person="John Doe"/>
  </eca:result>
  <eca:variable-bindings>
    <eca:tuple>
      <eca:variable name="To">Paris</eca:variable>
      <eca:variable name="Person">John Doe</eca:variable>
    </eca:tuple>
  </eca:variable-bindings>
</eca:answer>
```

Figure 6.8: Sending Answers with the ECA Engine Client

istration of rules at the ECA engine.

2. The class EventPanel (see Figure 6.8) provides an interface for sending <eca:answers> messages to the ECA engine and by doing so, simulating an event detection engine.

3. The class LogPanel presents the output of the framework log to the user. A refresh can happen either in a predefined time interval or at the user's request.

7 Conclusion

In this thesis, an ECA engine for evaluating *Event-Condition-Action* rules was developed, using standard Web technologies (XML, SOAP and Java). This ensures, that the ECA engine can easily be integrated into existing information systems in order to turn them into *active* participants of the Semantic Web. By the use of declarative rules for defining the behavior of these systems, a later reasoning about the same is rendered possible.

During the development of the ECA engine, it became apparent that the internal architecture could be made much simpler by dividing it into two parts. Thus, there is a core part that handles the declarative evaluation of a rule and keeps the current variable bindings. It is accompanied by a generic wrapper around the language processors that hides the complexity of their invocation and returns only the resulting variable bindings to the core ECA engine.

Possible areas of further development are:

- The lifting of the data model from XML to RDF which would allow for working with concepts rather than plain data.
- The implementation of a grouping mechanism for tuples of variable bindings (similar to SQL's GROUP BY).
- The implementation of distributed transactions. This would provide ACID[1] properties to a set of action components.

While the ECA engine already provides services of reactive behavior to its clients, an important aspect of the ECA framework is not yet completely dealt with: the propagation and detection of events. The development of a respective *Event Detection Engine* is currently subject to a project work and will, when finished, complete the prototypical implementation of the basic ECA environment.

[1]Atomicity, Consistency, Isolation, and Durability.

Bibliography

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

[2] Apache Axis: an Implementation of the SOAP Protocol. `http://ws.apache.org/axis`.

[3] Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. Draft for the Workshop "Reactivity on the Web", March 2006.

[4] 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.

[5] eXist: an Open Source Native XML Database. `http://exist-db.org/`.

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

[7] Rob Harrop and Jan Machacek. *Pro Spring.* apress, 2005.

[8] Alfons Kemper and Andre Eickler. *Datenbanksysteme.* Oldenbourg, 4th edition, 2004.

[9] 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.

[10] Wolfgang May, José Júlio Alferes, and Ricardo Amador. An ontology- and resources-based approach to evolution and reactivity in the semantic web. In *Ontologies, Databases and Semantics (ODBASE)*, number 3761 in LNCS, pages 1553–1570. Springer, 2005.

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

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

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

[14] The ruleCore® system — advanced business situation detection. . `http://www.rulecore.com`.

[15] Spring Framework. `http://www.springframework.org/`.

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

[17] W3C – the world wide web consortium. `http://www.w3c.org/`.

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

[19] XML Schema. `http://www.w3.org/XML/Schema`, 1999.

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

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