# An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web

Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert

Institut für Informatik, Universität Göttingen
{behrends, fritzen, may, dschuber}@informatik.uni-goettingen.de

**Abstract.** We describe a generic ECA service for implementing behavior using heterogeneous languages in the Semantic Web. The module details and implements our recent work on an ontology and language concept for a modular approach to ECA rules in the Semantic Web. The ECA level provides generic functionality independent from the actual languages and semantics of event detection, queries, and actions.

## 1 Introduction

In [MAA05b], we presented an ontology-based approach for specifying (reactive) behavior of the Web and evolution of the Web that follows the *Event-Condition-Action (ECA)* paradigm. The approach provides a modular framework for *composing* languages for event detection, queries, conditions, and actions by separating the ECA semantics from the underlying semantics of events, queries, and actions. We deal with the heterogeneity of the components (i.e., event, query and action languages) by associating every rule component with a language. The language descriptions (as resource descriptions) provide pointers to appropriate Web Services that implement the respective languages in a service-oriented architecture. An accompanying proposal for a rule markup language has been given in [MAA05a]. In the present paper, we describe a prototypical implementation (in Java) of an ECA engine for this framework: Section 2 shortly reviews the underlying ontology and language model of the general framework for ECA rules. Section 3 describes the global semantics of the rules, focusing on the handling of variables. Section 4 then describes the actual evaluation and communication with the component language services and illustrates it by a running example. Section 5 concludes the paper.

## 2 Language Heterogeneity: Rule Components and Languages

For dealing with the different languages for events, queries and tests, and actions, we prefer a *declarative* approach with a *clean, modular* design as a "Normal Form": First detect just the dynamic part of a situation (event), then optionally obtain additional information by queries (that can be stated using different languages), evaluate a *boolean* test, and, if "yes", then actually *do* something – as

T. Grust et al. (Eds.): EDBT 2006 Workshops, LNCS 4254, pp. 887–898, 2006.

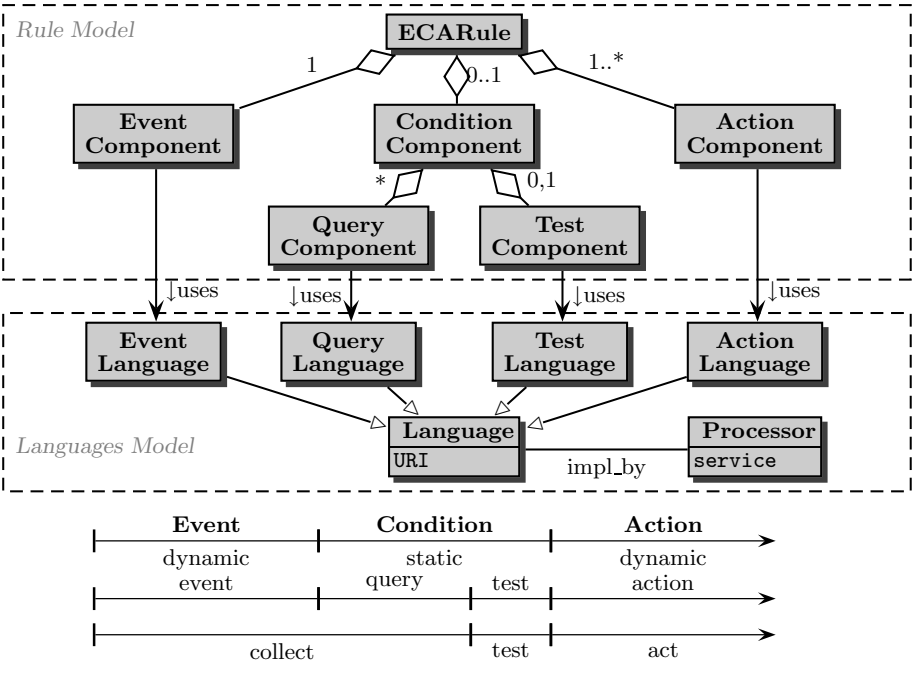


**Fig. 1.** ECA Rule Components and Languages (simplified from [MAA05a])

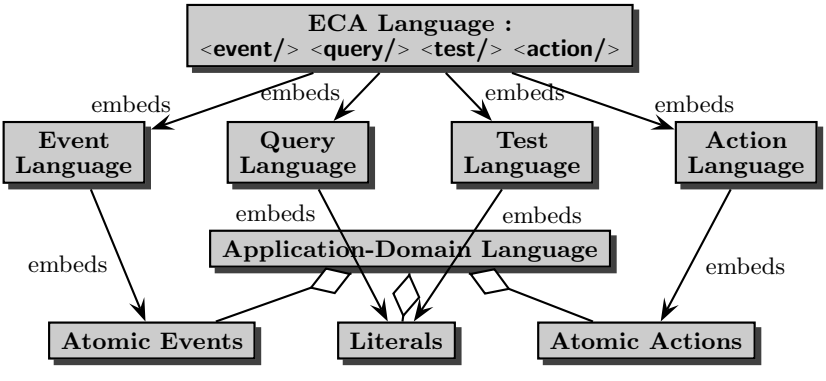


**Fig. 2.** Hierarchy of Languages (simplified from [MAA05a])

shown in Figure 1. Thus, every rule uses an event language, one or more query languages, a test language, and one or more action languages for the respective components. Rules and their components are objects of the Semantic Web, i.e., subject to a generic *rule ontology* as shown in the UML model. Every component is associated with its language (seen as a resource), identified by a URI. With

this URI, further information is associated that allows to address a suitable Web Service that implements the language; details about the service-oriented architecture can be found in [MAA05b].

The framework defines a hierarchical structure of language families (wrt. embedding of language expressions) as shown in Figure 2: the ECA language embeds event, query, test, and action languages. Rules combine one or more languages of each of the families. In general, each such language consists of an application-independent syntax and semantics (e.g., event algebras, query languages, boolean tests, process algebras) which is then applied to a domain (e.g. travelling). The domain ontologies define the static and dynamic notions of the application domain, i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of delayed flights, actions of reserving tickets).

## 3 Semantics of ECA Rules and Variables

For classical deductive rules, there is a *bottom-up* evaluation where the body is evaluated and produces a set of tuples of variable bindings. Then, the rule head is "executed" by *iterating* over all bindings, for *each* binding instantiating the structure described in the head (in some languages also executing actions in the head). We define the semantics of ECA rules as close as possible to this semantics, adapted to the temporal aspect of an event:

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

*Logical variables* are used in the same way as in Logic Programming for communication between the different components of a rule: the semantics of rules is based on sets of tuples of (answer) variable bindings. In case that a variable occurs more than once in a rule, it is handled as a join variable. While in Logic Programming rules, variables must be bound by a positive literal in the body to serve as join variables in the body and to be used in the head, in ECA rules we have four components: A variable must be bound in the rule, in an "earlier" (Event<Query<Test<Action) or at least the same component as where it is used. Usage can be as a join variable in case of the Event, Query, or Test component, or to execute ("derive") an action in the Action component (that corresponds to the rule head). Variables can be bound to values/literals, references (URIs), XML or RDF fragments, or events (marked up as XML or RDF fragments).

While the semantics of the ECA *rules* provides the infrastructure and global semantics, the *component languages* provide the local semantics. For dealing with heterogeneous languages, the ECA level does only minimally constrain the component languages. Communication between the ECA engine and the Event, Query, Test, and Action components is done by exchanging variable bindings. Component languages use variables in two different ways: Logic Programming-style languages match free variables, e.g. query languages like Datalog, F-Logic [KL89], XPathLog [May04], or Xcerpt [BS02]; similar techniques can also be applied to design languages for the event component. Functional-style languages act as functions over a database or an event stream, and some input/environment

variables. In the XML world, such languages return an XML fragment (e.g. XQuery). In most classical approaches for event languages (e.g., as in SNOOP [CKAK94]), the "result" of an expression is often considered to be the sequence of detected events that "matched" the event expression in an event stream. Variables can be bound on the rule level for binding results of functional expressions by borrowing from XSLT as <eca:variable name="*name*">*content*</eca:variable> where *content* can be any expression whose value is then bound to the variable (i.e., an event specification or a query). Similar constructs are recommended to use in the component languages. We will focus here on the ECA level, keeping the component expressions as simple as possible.

## 4 The ECA Engine

### 4.1 Architecture

The architecture is shown in Figure 3. The ECA engine controls the evaluation of a rule, i.e., *when* to evaluate *which* rule component, and keeps the state information during the evaluation. The communication with the autonomous remote component language processors is done via the *Generic Request Handler (GRH)*, using an XML markup for requests and answers. Using the namespace declaration of the components, the GRH determines an appropriate language processor and sends the request and the relevant variable bindings to it in an appropriate form. After receiving the answer, the obtained variable bindings are communicated back to the ECA engine.

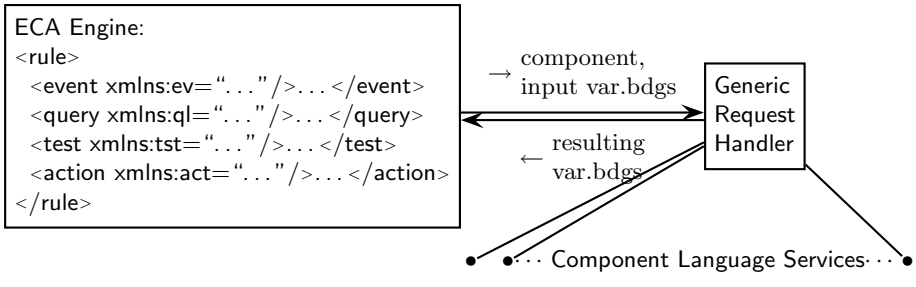


**Fig. 3.** Global Service-Oriented Architecture

This process is described below and illustrated by an exemplary ECA rule of a car-rental company: when a customer books a flight, cars similar in size to his own cars are offered at the given destination (see Figure 4).

### 4.2 Firing ECA Rules: The Event Component

Upon registration of a rule in the ECA engine, its event component is submitted to the GRH. The GRH inspects the namespace of the event language and submits

```
<eca:rule xmlns:eca="http://www.semwebtech.org/06/eca-ml">
  <eca:event><!-- detect a booking by a person --></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 cars that are available at the destination. -->
  </eca:query>
  <eca:action><!-- inform the customer about suitable cars --></eca:action>
</eca:rule>
```

**Fig. 4.** Outline of the Sample Rule

the event component to an appropriate event detection service (see Figure 5). In our simple example, the event component consists only of an atomic event pattern. The event pattern is thus sent directly to an *Atomic Event Matcher* that is aware of relevant events.

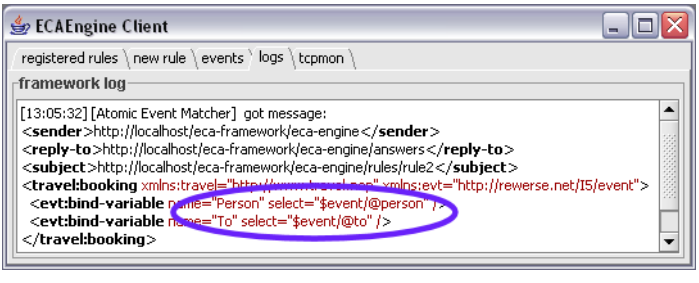


**Fig. 5.** Registration of the Event Component

The event detection service evaluates the event specification against the stream of events. When an (atomic) event that matches the specification, e.g.,

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

occurs, the detection of the event component pattern is signalled from the event detection service to the GRH (containing the identification of the rule, the event sequence that matched the pattern and the collected variable bindings). The GRH forwards it to the ECA engine as shown in Fig. 6(1), using an XML markup for answers and tuples of variable bindings. The arrival of the event detection message marks the starting point of the rule evaluation at the ECA engine. The ECA engine creates one or more instances of the rule with appropriate variable bindings according to the number of answer elements in the message (Fig. 6(2)).

*Languages for Composite Events.* The event component can also use *arbitrary* language for specifying composite events – as far as a service that actually does the event detection is provided. In this case, the event component is of the form

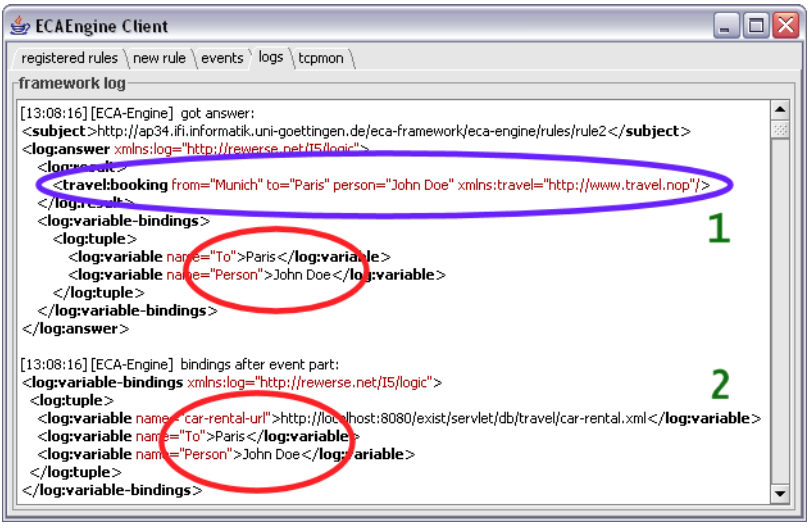


**Fig. 6.** Detection of the Event Component

```
<eca:event xmlns:evt="uri of the event language">
   <evt:operator> nested expression in the event language </evt:operator>
</eca:event>
```

and the component is then registered and processed at an appropriate service associated with the language's URI [MAA05b].

Possible languages here are e.g. an extension of SNOOP [CKAK94] with logical variables where a framework-aware service has been implemented in [Spa06] (with input in XML markup), or XChange [BP05]. Both languages return the event sequence as functional result and bind/use logical (join) variables.

### 4.3 The Query Components

The query components serve for obtaining *static* information from Web resources based on the information contained in the event. The query component is very similar to the evaluation of rule bodies in Logic Programming, extending the set of tuples of variable bindings (and also probably restricting it via join conditions). Since we also allow answers of functional query languages, the semantics is adapted accordingly: when bound to a variable at the rule level, each answer yields a separate variable binding.

In our example, at this time, the following facts are known: the name of the person who booked the flight and the destination city (Fig. 6(2)). The name is used to ask for the cars that this person owns at home. Note that here, the query is stated as an "opaque" XQuery code fragment (against an XML document on the Web) without markup. The query code together with the values of the input variables is communicated to the GRH as shown in Figure 7.

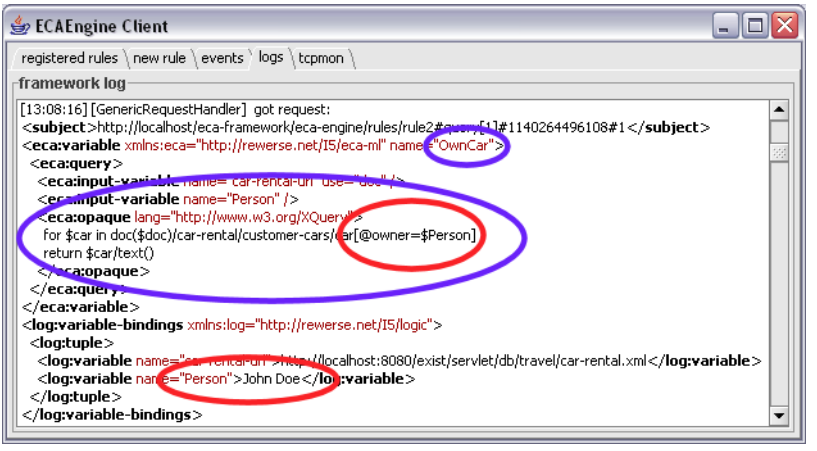


**Fig. 7.** Sending the First Query Component to the GRH: Own Cars

*Languages for the Query Component.* In contrast to the event component, many query languages for SQL, XML and RDF data are around, and many Web nodes already support interfaces for them. Thus, for current applications, *opaque* query components where the query is just given as a string and submitted to such a service (using e.g. HTTP, or calling a saxon-based [saxon] wrapper for XQuery) are expected to be frequently used:

```
<eca:query>
  <eca:opaque (language= "name of the language"|url= "URL of WebService)" >
    query
  </eca:opaque>
</eca:query>
```

### 4.4 The Generic Request Handler

The Generic Request Handler acts as a mediator for dealing with remote services. It inspects the namespace declaration of the components (or the language attribute in case of opaque fragments) for determining an appropriate language processor and forwards the request to it in an appropriate form. For framework-aware services, the incoming requests can just be forwarded. To integrate non-framework-aware services, the GRH uses information about the communication protocol and method in addition to the processor's capabilities wrt. the handling of variable bindings (cf. the second query discussed later).

The first query is forwarded together with the input variable bindings from the GRH to a framework-aware wrapped Saxon [saxon] XQuery processor node. The node evaluates the query and returns one <log:answer> for each result to the GRH as shown in Figure 8(1). For such processors that return a functional result (in an <log:result> element), the query component is surrounded by a

<eca:variable> element (as in our example, see Figure 4). The GRH extends the input bindings with binding the functional result(s) to the given variable. It generates an appropriate <log:answers> message and sends it back to the ECA engine as shown in Figure 8(2) where it is then joined with the existing variable bindings. Note that since John Doe owns two cars at home, a *Golf* and a *Passat*, there are now two tuples of variable bindings (Figure 8(3)).

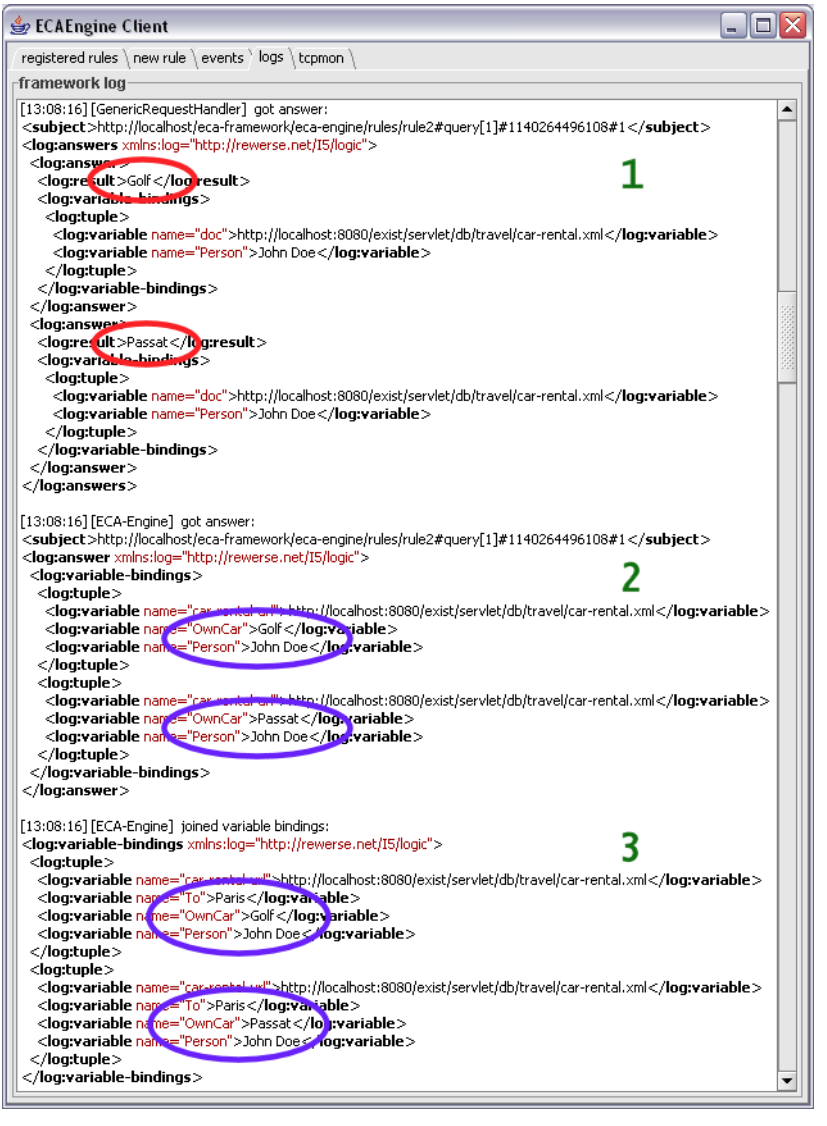


**Fig. 8.** Answer to the First Query Component: Own Cars

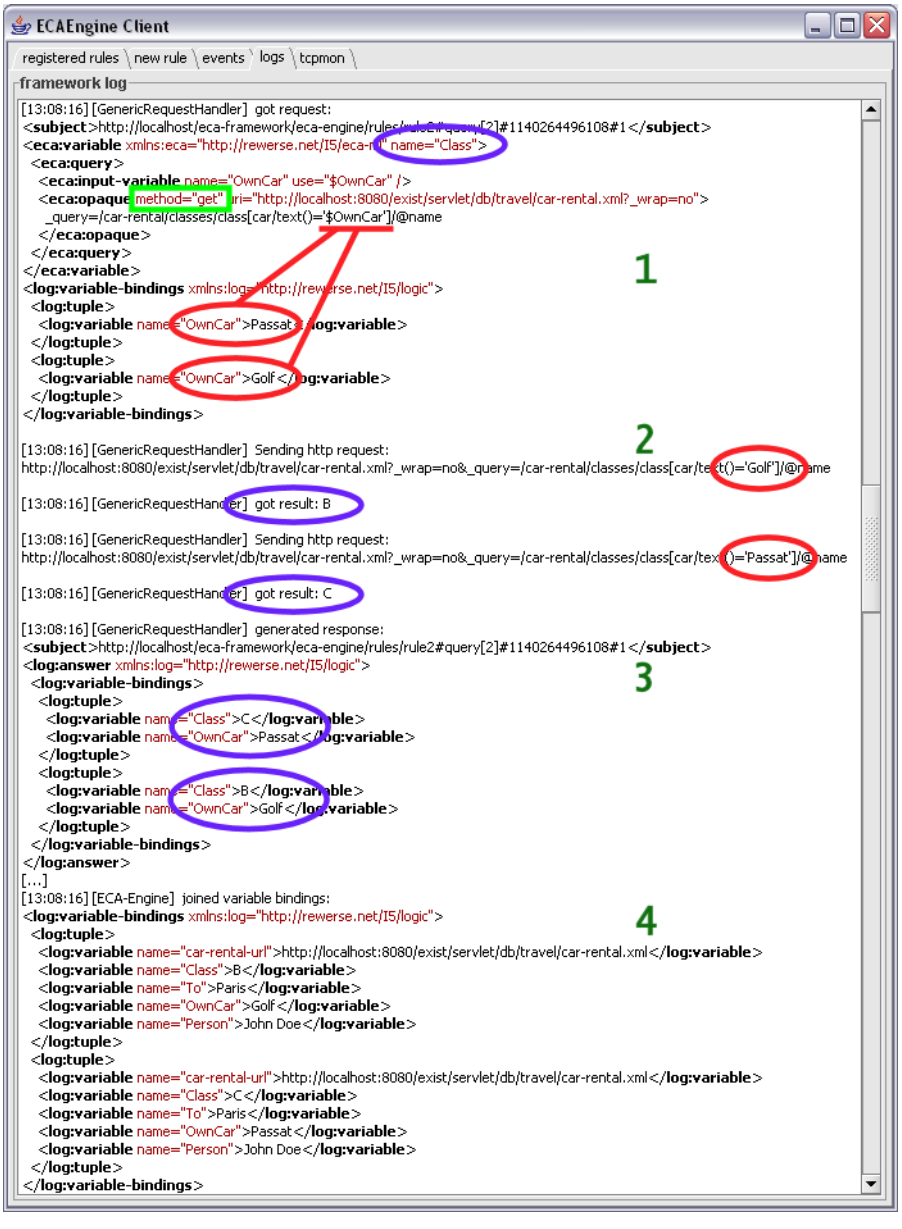


**Fig. 9.** Evaluation of the 2nd Query against a Framework-Unaware Service

In the next query, another database is queried for the classes (sizes) of the respective cars as shown in Figure 9(1). The <eca:opaque> element specifies to contact an unwrapped, framework-unaware XQuery node (an eXist database) via HTTP GET. Variables in the query string are replaced by their values and the

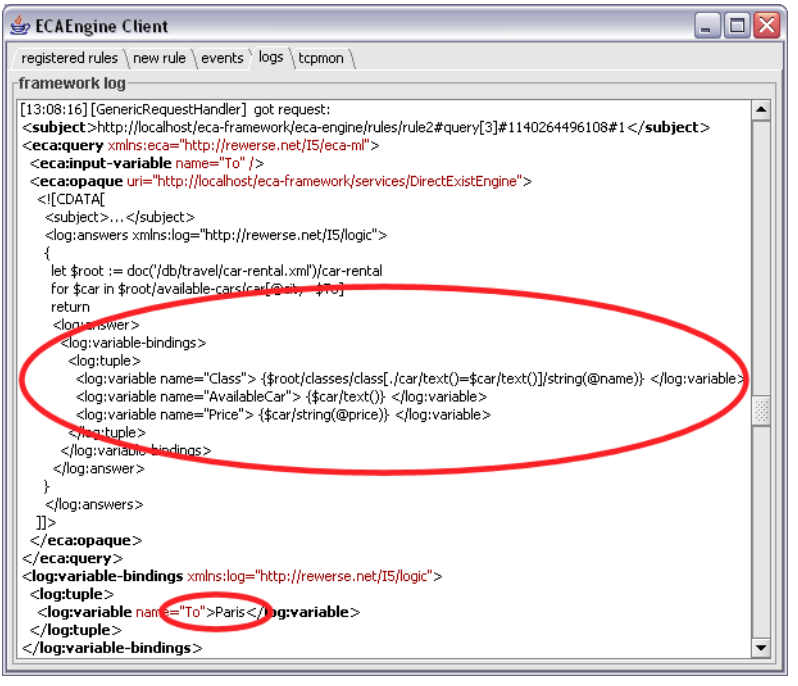


**Fig. 10.** Query Against Available Cars, Generating a <log:answers> Structure

query is submitted. In the example, the GRH executes the request for every tuple of the input variable bindings, once for "Golf" and once for "Passat" (Fig. 9(2)), and binds the results to the variable Class. The resulting variable bindings are then sent as the content of an <log:answers> message back to the ECA engine (Fig. 9(3)) where they are joined with the existing tuples (Fig. 9(4)).

Next, another query retrieves all cars that are available at the destination city (see Figure 10). Here an XQuery engine is addressed directly with a query that generates the required <log:answers> structure to "fake" a framework-aware service. The available cars (of classes B and D) are compared to the classes of the cars owned by the customer (B and C) as shown in Figure 11. The join semantics (natural join over class) eliminates tuples containing a car either of class "C" or of class "D", and only those with class "B" remain.

### 4.5 The Test and Action Components

These two components follow the same principle. The test component (which corresponds to the WHERE clause in SQL and is empty in our example) contains a condition over the bound variables which discards those tuples that do not satisfy the condition. In general, it is evaluated locally, using only simple comparison predicates. The action component then is the one where *actually* something is done: for each tuple of variable bindings, the action component is executed, again via the GRH. This can include commands on the database level, explicit message sending, or actions on the domain ontology level.

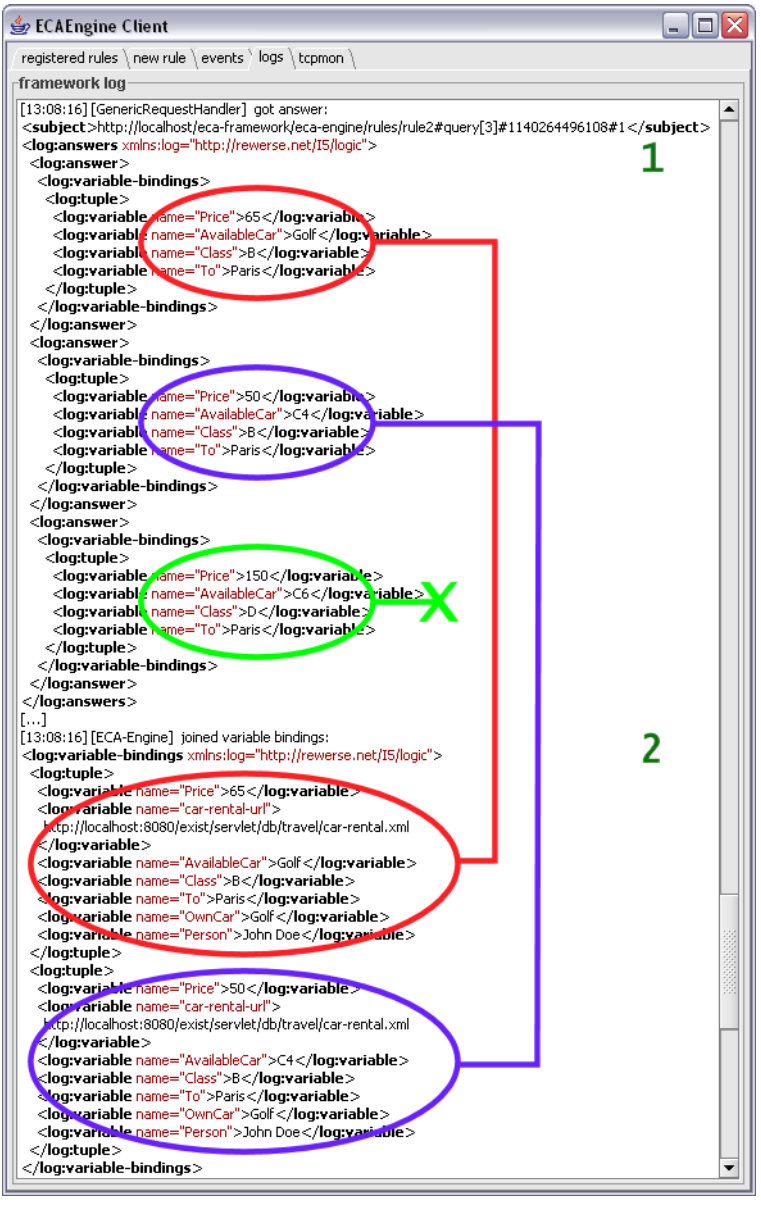


**Fig. 11.** Evaluation of the Available Cars

## 5 Conclusion

The above *ECA engine* and *Generic Request Handler* implement the upper level of the generic ECA framework proposed in [MAA05a, MAA05b]. They can be used for combining arbitrary event detection, query and action languages and

respective engines. A variety of such engines, including sample domain services are currently being developed.

**Acknowledgements.** This research has been funded by the European Commission within the 6th Framework Programme project REWERSE, no. 506779.

## References

[BP05] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *ACM Symp. Applied Computing*. ACM, 2005.

[BS02] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation Unification. In *Intl. Conf. on Logic Programming (ICLP)*, Springer LNCS 2401, pp. 255–270, 2002.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. *VLDB*, 1994.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In J. Clifford, B. Lindsay, and D. Maier, editors, *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 134–146, 1989.

[MAA05a] W. May, J. J. Alferes, and R. Amador. Active Rules in the Semantic Web: Dealing with Language Heterogeneity. In *Rule Markup Languages (RuleML)*, Springer LNCS 3791, pp. 30–44, 2005.

[MAA05b] W. May, J. J. Alferes, and R. Amador. An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web. In *Ontologies, Databases and Semantics (ODBASE)*, Springer LNCS 3761, pp. 1553–1570, 2005.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Progr.*, 4(3):239–287, 2004.

[saxon] Michael Kay. SAXON: an XSLT processor. `http://saxon.sourceforge.net/`.

[Spa06] Sebastian Spautz. Automatenbasierte Detektion von Composite Events gemäss SNOOP in XML-Umgebungen. *Diplomarbeit, TU Clausthal (in german)*, 2006.