Active Rules in the Semantic Web: Dealing with Language Heterogeneity

Wolfgang May[1], José Júlio Alferes[2], and Ricardo Amador[2]

[1] Institut für Informatik, Universität Göttingen
[2] Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa

Abstract. In the same way as the "static" Semantic Web deals with data model and language heterogeneity and semantics that lead to RDF and OWL, there is language heterogeneity and the need for a semantical account concerning Web dynamics. Thus, generic Rule Markup has to bridge these discrepancies, i.e., allow for *composition* of component languages, retaining their distinguished semantics.

In this paper we analyze the basic concepts for a general language for evolution and reactivity in the Semantic Web. We propose an ontology based on the paradigm of Event-Condition-Action (ECA) rules including an XML markup. In this framework, different languages for events (including languages for composite events), conditions (queries and tests) and actions (including complex actions) can be composed to define high-level rules for describing behavior in the Semantic Web.

1 Introduction

The goal of the *Semantic Web* is to bridge the heterogeneity of data formats, schemas, languages, and ontologies used in the Web to provide unified view(s) on the Web, as an extension to today's *portals*. In this scenario, XML (as a format for storing and exchanging data), RDF (as an abstract data model for states), OWL (as an additional framework for state theories), and XML-based communication (Web Services, SOAP, WSDL) provide the natural underlying concepts. In contrast to the current Web, the *Semantic Web* should be able not only to support querying, but also to propagate knowledge and changes in a semantic way. This *evolution* and *behavior* depends on the cooperation of nodes. In the same way as the main driving forces for XML and the Semantic Web idea were the heterogeneity and the lack of accessing the semantics of the underlying data, the heterogeneity of concepts for expressing behavior requires for an appropriate handling on the semantic level. Since the contributing nodes are based on different concepts such as data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that the *concepts* and the actual *languages* are independent. While for a data model and for querying, a "common" agreed standard evolves with RDF/RDFS, OWL and languages like RDF-QL etc., the concepts for describing and implementing behavior are much more different, due to different needs, and it is −in our opinion− unlikely that there will be a unique language for this throughout the Web.

Here, *reactivity* and its formalization as *Event-Condition-Action (ECA) rules* offer a suitable common model because they provide a modularization into clean concepts with a well-defined information flow. An important advantage of them is that the *content* of a rule (event, condition, and action specifications) is separated from the *generic semantics* of the ECA rules themselves that provides a well-understood formal semantics: when an event (atomic event or composite event) occurs, evaluate a condition, and if the condition is satisfied then execute an action (or a sequence of actions, a program, a transaction, or even start a process). ECA rules provide a generic uniform framework for specifying and implementing communication, local evolution, policies and strategies, and –altogether– global evolution in the Semantic Web.

In the present paper, we describe an ontology-based approach for specifying (reactive) behavior in the Web and evolution of the Web that follows the ECA paradigm. We propose a modular framework for *composing* languages for events, queries, conditions, and actions by separating the ECA semantics from the underlying semantics of events, queries, and actions. This modularity allows for high flexibility wrt. these sublanguages, while exploiting and supporting their meta-level *homogeneity* on the way to the Semantic Web.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general, especially if one wants to reason about evolution, ECA rules (and their components) must be communicated between different nodes, and may themselves be subject to being updated. For that, the ECA rules themselves must be represented as data in the (Semantic) Web. This need calls for an ontology and a (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [RML], but it does not tackle the complexity and language heterogeneity of events, actions, and the generality of rules, as described here.

Related Work – Concepts that have to be covered and integrated by this approach. The importance of being able to update the Web has long been acknowledged, and several language proposals exist (e.g. XUpdate [XML00] and an extension to XQuery in [TIHW01]) for just that. More recently some reactive languages have been proposed that are also capable of dealing-with/reacting-to some forms of events, evaluate conditions, and upon that act by updating data. These are e.g. XML active rules in [BCP01,BBCC02], an ECA language for XML [BPW02], and RDFTL [PPW04], which is an ECA language on RDF data. These languages do not provide for more complex events, and they do not deal with heterogeneity at the level of the language. Relating to our approach, these rules will be used on a low abstraction level.

The recent work on the language XChange [BP05] already aims at specifying more complex events and actions; nevertheless, it still presents a monolithic language for ECA rules in the Web, not dealing with the issue of language heterogeneity.

Structure of the paper. In the next section, we analyse the abstraction levels of behavior (and ECA rules) in the Semantic Web. The modular structuring of

our approach into different language families is presented in Section 3. Section 4 then analyzes the common structure and requirements for the E, Q, T and A languages. Section 5 describes the global semantics of the rules, focussing on the handling of variables for communication between the rule components. Section 6 concludes the paper.

2 Behavior: Abstraction Levels

As described above, the *Semantic Web* can be seen as a network of autonomous (and autonomously evolving) nodes. Each node holds a *local* state consisting of extensional data (facts), metadata (schema, ontology information), optionally a knowledge base (intensional data), and, again optional, a behavior base. In our case, the latter is given by the ECA rules under discussion that specify which actions are to be taken upon which events under which conditions.

In the same way as the "Semantic Web Tower" distinguishes between the data level and the semantic (RDF/OWL) level, behavior can be distinguished wrt. different levels. There is local behavior of Web nodes, partially even "hidden" inside the database, and local rules on the logical level, and *Business Rules* on the application level. The cooperation in the Semantic Web by global *Business Rules* is then based on this local behavior. The proposed comprehensive framework for active rules in the Web integrates all these levels.

Physical Level: Database Triggers.   The base level is provided by rules on the *programming language and data structure level* that react directly on changes of the underlying data. Usually they are implemented inside the database as *triggers*, e.g., in SQL, of the form ON database-update WHEN condition BEGIN pl/sql-fragment END. In the Semantic Web, the data model level is assumed to be in XML (or RDF, see below) format. While the SQL triggers in relational databases are only able to react on changes of a given tuple or an attribute of a tuple, the XML and RDF models call for more expressive event specifications according to the (tree or graph) structure. Work on triggers for XML data or the XQuery language has e.g. been described in [BBCC02,BPW02,PPW03,MAA05].

Logical Level: RDF Triggers.   Triggers for RDF data have been described in [PPW04,MAA05]. Triggering events on the RDF level should usually bind variables Subject, Property, Object, Class, Resource, referring to the modified items (as URIs), respectively in the same form as SQL's OLD and NEW values. In case that data is stored in an RDF database, these triggers can directly be implemented on the physical, storage level. RDF trigger events already use the terminology of the *application ontology* but still the RDF structures of the logical level. Application-level events can be raised by such rules, e.g.,

```
ON INSERT OF has_professor OF department
  % (comes with parameters $subject=dept, $property, and $object=prof )
  % $university is a constant defined in the (local) database
RAISE EVENT (professor_hired($object, $subject))
```

which is then actually an event professor_hired(*prof*, *dept*) of the application ontology on which *business rules* can react.

On the physical and logical levels, in general actions and events coincide (an update to a data item).

Semantic Level: Active Rules. In rules on the semantic level, the events, conditions and actions refer to the ontology level:

ON (professor_hired($prof, $dept, $univ))
WHEN $Books := *select relevant books for people at this dept*
BEGIN *do something* END

Here, there is an important difference between *actions* and *events*: an event is a visible, possibly indirect or derived, consequence of an action. E.g., the action is to "debit 200E from Alice's bank account", and visible events are "a change of Alice's bank account" (that is immediately detectable from the update operation), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update).

More complex rules use also composite events and queries against the Web. Composite events in general consist of subevents at that are oroginally located at different locations.

3 Language Heterogeneity and Structure: Rules, Rule Components and Languages

An ECA concept for supporting interoperability in the Semantic Web needs to be more flexible and adapted to the "global" environment. Since the Semantic Web is a world-wide living organism, nodes "speaking different languages" should be able to interoperate. So, different "local" languages, be it the condition (query) languages, the action languages or the event languages/event algebras have to be integrated in a common framework. There is a more succinct separation between event, condition, and action part, which are possibly (i) given in separate languages, and (ii) possibly evaluated/executed in different places. For this, an (extendible) ontology for rules, events, and actions that allows for *interoperability* is needed, that can be combined with an infrastructure that turns the instances of these concepts into objects of the Semantic Web itself.

In the present paper, we will focus on the language and markup issues; a service-oriented architecture is discussed in [MAA05].

3.1 Components of Active Rules in the Semantic Web

A basic form of active rules are the well-known *database triggers*, e.g., in SQL, of the form ON *database-update* WHEN *condition* BEGIN *pl/sql-fragment* END. In SQL, the *condition* can only use very restricted information about the immediate database update. In case that an action should only be executed under certain conditions which involve a (local) database query, this is done in a procedural way in the *pl/sql-fragment*. This has the drawback of not being declarative;

reasoning about the actual effects would require to analyze the program code of the *pl/sql-fragment*. Additionally, in the distributed environment of the Web, the query is probably (i) not local, and (ii) heterogeneous in the language – queries against different nodes may be expressed in different languages. For our framework, we prefer a *declarative* approach with a *clean, declarative* design as a "Normal Form": Detecting just the dynamic part of a situation (event), then check *if* something has to be done by first obtaining additional information by a query and then evaluating a *boolean* test, and, if "yes", then actually *do* something – as shown in Figure 1.
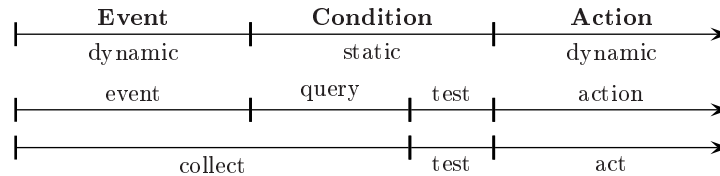

Fig. 1. Components and Phases of Evaluating an ECA Rule

With this further separation of tasks, we obtain the following structure:

- every rule uses an event language, one or more query languages, a test language, and an action language for the respective components,
- each of these languages and their constructs are described by metadata and an ontology of its semantics, and their nature as a language, e.g., associating them with a processor,
- there is a well-defined *interface* for communication between the E, Q&T, and A components by variables.

Sublanguages and Interoperability. For expressing and applying such rules in the Semantic Web, a uniform handling of the event, condition, and action sublanguages is required. Rules and their components are objects of the Semantic Web, i.e., subject to a generic *rule ontology* as shown in the UML model in Figure 2. The ontology part then splits in a structural and application-oriented part, and an infrastructure part about the language itself. In this paper, we restrict ourselves to the issues of the ECA language structure language and the markup itself. From the infrastructure part, we need here only to know that each language is identified by a URI with which information about the specific language (e.g, an XML Schema, an ontology of its constructs, a URL where an interpreter is available) is associated; details about a service-oriented architecture proposal that makes use of this information can be found in [MAA05].

3.2 Markup Proposal: ECA-ML

According to the above-mentioned structure of the rules, we propose the following XML markup. The connection with the language-oriented resources is provided via the namespace URIs:
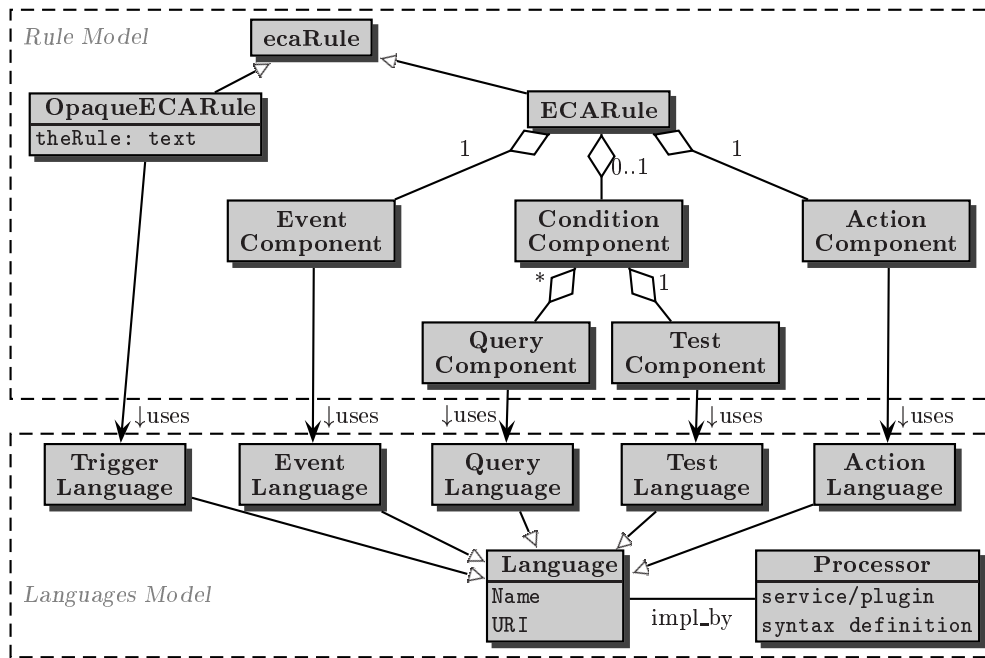
Fig. 2. ECA Rule Components and corresponding Languages II

```
<!ELEMENT rule (event, query*, test?, action)>
```
<**eca:rule** *declaration of namespaces e.g.* xmlns:evlg="http://my.event-language.org" >
  rule-specific contents
  <**eca:event**> *event specification as* <evlg:sequence> ... </evlg:sequence> </**eca:event**>
  <**eca:query**> *query specification* </**eca:query**>
  <**eca:test**> *test specification* </**eca:test**>
  <**eca:action**> *action specification* </**eca:action**>
  <!– *event, condition, and action specification use markup elements*
    of their sublanguage's namespaces; see below –>
</**eca:rule**>

A similar markup for ECA rules has been used in [BCP01] with *fixed* languages (using a basic language for atomic events on XML data, XQuery as condition language and SOAP in the action part). This fixed approach falls short wrt. the language heterogeneity, and especially the use and integration of languages for composite events. In the same way, the XChange approach [BP05] uses a fixed language for specifying the event, condition, and action part. In contrast, the approach proposed here allows for using arbitrary languages. Thus, these other proposals are just *two* possible configurations. Our approach even allows to mix components of both these proposals.

Triggers as Rules. The above database triggers, where a rule is given in an internal syntax, are just wrapped as *opaque* rules: the eca:rule element then

contains only an eca:opaque element with text contents (program code of some
rule language) and attributes lang (text) and ref (URI where an interpreter is
found, similar to the namespace); a similar meachanism to XML's NOTATION can
also be applied:

```
<eca:rule>
  <eca:opaque name="SQL trigger" ref= "uri of the trigger language">
    ON database-update WHEN condition BEGIN action END
  </eca:opaque>
</eca:rule>
```

Since opaque rules are ontologically "atomic" objects, their event, condition, and
action parts cannot be accessed by Semantic Web concepts. Note that there are
canonic mappings between such triggers and their components and the general
ECA ontology, where then the components still end up as opaque native code
segments (see e.g. the analysis of the components structure in Sec. 4; esp. Fig. 5).

3.3 Hierarchical Structure of Languages

The framework defines a hierarchical structure of language families (wrt. embed-
ding of language expressions) as shown in Figure 3: As described until now, there
is an ECA language, and there are (heterogeneous) event, query, test, and action
languages. Rules will combine one (or more) languages of each of the families.
In general, each such language consists of an own, application-independent syn-
tax and semantics (e.g., event algebras, query languages, boolean tests, process
algebras or programming languages) that is then applied to a domain (e.g. trav-
elling, banking, universities, etc.). 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 train schedule changes,
actions of reserving tickets, etc.). Additionally, there are domain-independent
languages that provide primitives (with arguments), like general communication,
e.g. received_message(M) (where M in turn contains domain-specific content), or
transactional languages with an action commit(A) and an event committed(A)
where A is a domain-specific action.

   In the next section, we discuss common aspects of the languages on the
"middle" level (that immediately lead to the tree-style markup of the respective
components – thus, here the XML markup is straightforward). Section 5 then
deals with the interplay between the rule components.

4 Common Structure of Component Languages

The four types of rule components use specialized types of languages that share
the same algebraic language structure, although dealing with different notions:

 – event languages: every expression gives a description of a (possibly composite)
   event. Expressions are built by composers of an event algebra, and the leaves
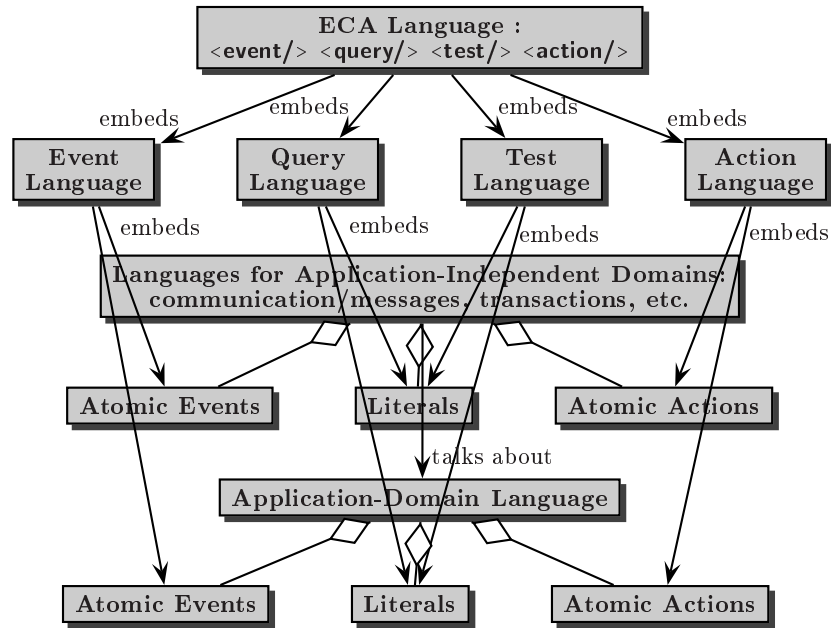   here are atomic events of the underlying application;

Fig. 3. Hierarchy of Languages

- query languages: expressions of an algebraic query-language;
- test languages: they are in fact formulas of some logic over literals (of that logic) and an underlying domain (that determines the predicate and function symbols, or class symbols etc., depending on the logic);
- action languages: every expression describes an (possible composite) activity. Here, algebraic languages (like process algebras) or "classical" programming languages (that nevertheless consist of expressions) can be used. Again, the atomic items are actions of the application domain.

Algebraic Languages. As shown in Figure 4, all component languages consist of an algebraic language defining a set of *composers*, and embedding *atomic* elements (events, literals, actions) that are contributed by *domain languages*, either for specific aplications or application-independent (e.g., messaging). Expressions of the language are then (i) atomic elements, or (ii) composite expressions recursively obtained by applying composers to expressions. Due to their structure, these languages are called *algebraic languages*, e.g. used in *event algebras*, *algebraic query languages*, and *process algebras*. Each composer has a given *cardinality* that denotes the number of expressions (of the same type of language, e.g., events) it can compose, and (optionally) a sequence of parameters (that come from another ontology, e.g., time intervals) that determines its *arity* (see Figures 4 and 5). For instance, "E_1 followed_by E_2 within t" is a binary composer to recognize the occurrence of two events (atomic or not) in a particular order within a time interval, where t is a parameter.
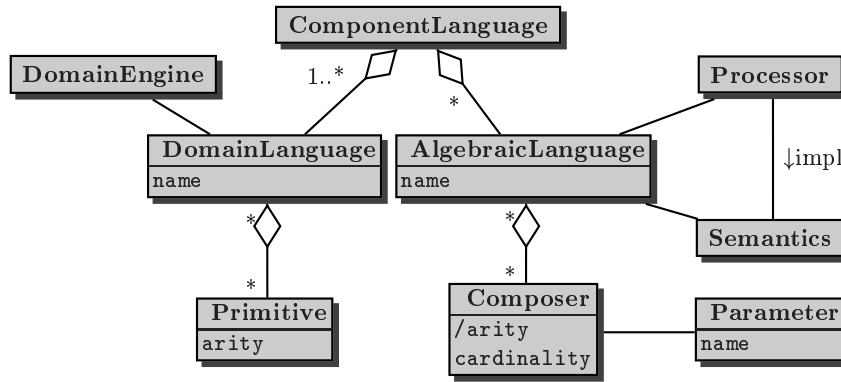
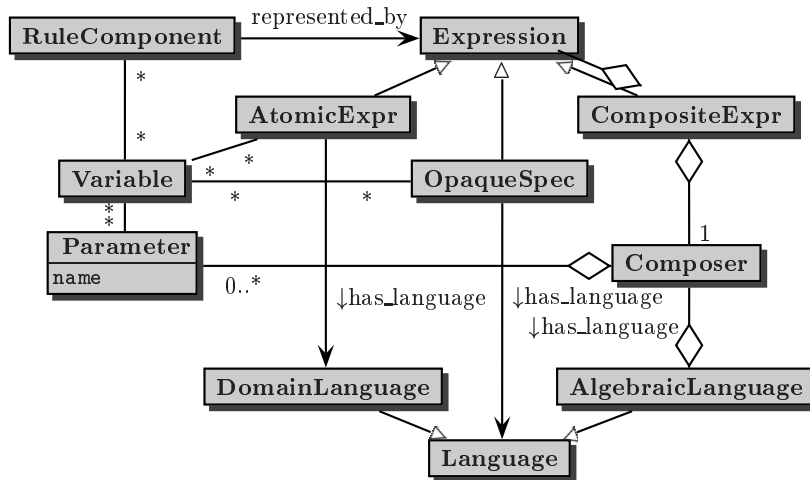**Fig. 4.** Notions of an Algebraic Language

Fig. 5. Syntactical Structure of Expressions of an Algebraic Language

Thus, language expressions are in fact trees which are marked up accordingly. The markup elements are provided by the definition of the individual languages, "residing" in and distinguished by the appropriate namespaces: the expression "structure" inside each component is built from elements of the algebraic language. An expression is either an atomic one (atomic event, literal, action) that belongs to a domain language, or an opaque one that is a code fragment of some event/query/logic/action language, or a composite expression that consists of a composer (that belongs to a language) and several subexpressions (where each recursively also belongs to a language). The leaves of the expression trees are the atomic events, literals, or actions, contributed by the application domains (and residing in the domain's ontology and namespace); they may again have an internal structure in the domain's namespace.

Note that it is also possible to nest composers and expressions from different languages of the same kind (e.g., an event sequence where the first part is described in event algebra A and the second in another algebra B), distinguishing them by the namespaces they use. Thus, languages are not only associated once on the *rule component* level, but this can also be done on the *expression* level.

5 Semantics of Rule Execution

For classical deductive rules in *bottom-up* evaluation, 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). The semantics of ECA rules should be 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.

To support communication between heterogeneous languages at the rule component level, there must be a precise convention between all languages how the different components of a rule can exchange information and interact with each other. In the following, we state some requirements on the contributing sub-languages and provide technical means to integrate these languages with our framework.

5.1 Logical Variables

We propose to use *logical variables* in the same way as in Logic Programming. For each instance of a rule, a variable must *bound* only once. In case that a variable *occurs* more than once, it acts then as a join variable. While in LP 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, or in an "earlier" ($E<Q<T<A$) or at least the same component as where it is used. Usage can be as a join variable in case of the E, Q, or T component, or to execute ("derive") an action in the action component (that corresponds to the rule head). This leads to a definition of safety of ECA rules that is similar to that of LP rules. Variables can be bound to several things: values/literals, references (URIs), XML or RDF fragments, or events (marked up as XML or RDF fragments). Expressions can also use local variables, e.g., in first-order logic conditions e.g., scoped by a quantifier.

Variable Handling on the Rule Level. As in Logic Programming, the semantics of rules is based on sets of tuples of (answer) variable bindings. We propose to use a simple tuple-based representation for interchange of variable bindings:

```
<variable-bindings>
  <tuple>
```

```
    <variable name="name" ref="" />
    <variable name="name"> contents </variable>
       ⋮
  </tuple>
</variable-bindings>
```

Variable Handling in E, Q, T, A Sublanguages. While the semantics
of the ECA *rules* provides the infrastructure for these variables, the markup of
specific languages must provide the actual handling of variables in its expressions.
Currently languages mainly use variables in two ways:

- Languages that bind variables by matching free variables (e.g. query lan-
  guages like Datalog, F-Logic [KL89], XPathLog [May04]). Here, the matches
  can be *literals* (Datalog) or literals and structures (e.g., in F-Logic, XPathLog,
  Xcerpt [BS02]). Similar techniques can also be applied to design languages
  for the event component.
- Functional-style languages: the sublanguages for the query and event com-
  ponents can be designed as functions over a database or an event stream. In
  the XML world, such languages that return a (nameless) data fragment (e.g.
  XQuery; also the expressions of the above-mentioned F-Logic, XPathLog and
  Xcerpt can be used in this way). For event languages, the "result" of an ex-
  pression can be considered the sequence of detected events that "matched"
  the event expression in an event stream (e.g., XChange [BP05]).

Variables: Syntax. We propose constructs for handling variables borrowed
from XSLT: use variables by {$*var-name*} and by <variable name="..."> elements:

- <eca:variable name="*name*">*content*</eca:variable>
  where *content* can be any expression whose value is bound to the variable
  (i.e., an event specification or a query).
- <eca:variable name="*name*" select="*ql-expr*" />
  Such expressions can be used for navigational access/comparison of values,
  or for defining a new variable based on already bound ones in *expr* as a
  shorthand for

  ```
    <eca:variable name="name">
      <eca:query xmlns:ql="ql-url">
        <eca:opaque> expr </eca:opaque>
      </eca:query>
    </ eca:variable>
  ```

  where *ql* is a (simple!) query language, e.g. XPath.

Both constructs can be used on the rule level (e.g., for binding the result of the
event component; see later example), and we recommend also to consider them
when designing component languages.

5.2 Firing ECA Rules: the Event Component

Formally, detection of an event results in an occurrence indication, together with information that has been collected: An ECA rule is fired for each successful detection of the specified event. Thus, initial variable bindings are produced by the event component. The event component consists –as shown above– of an event algebra term whose leaves are atomic events. The pattern is "matched" against the stream of detected events. Inside of <eca:atomic-event> elements, the domain namespaces of the are used for specifying *event patterns* to be matched. In the event component, variables can be bound as described above pattern-based with <eca:variable name= *"var-name"* > ... </eca:variable> , or navigation-based: inside the atomic event itself (as an XML fragment) is available as $event. Then,

<eca:variable name= *"var-name"* select= *"*$event/*path..."* />

can be used to match and access data within the event.

In many approaches (including the SNOOP event algebra [CKAK94]), the "result" of event detection is the sequence of the events that "materialized" the event pattern to be detected. In this case, an appropriate way is to bind this result to a variable (in the present case, using the XML representations of the events). Further variable bindings can then be extracted by subsequent <eca:query> or <eca:variable> elements.

Example 1. Consider the following scenario: "when registration for an exam of subject S is opened, students X register, and registration for S closes, then ... do something". This cumulative event can be specified in SNOOP as

$A^*(reg\_open(Subj), register(Subj, Stud), reg\_close(Subj))$.

The incoming domain-level events are e.g. of the form <uni:register subject="Databases" name="John Doe"/>. The following markup of the event component binds the complete sequence to regseq and the subject to Subj (note that Subj is used as a join variable):

```
<eca:rule ... >
  <eca:variable name="regseq">
    <eca:event xmlns:xmlsnoop="http://xmlsnoop.nop">
      <xmlsnoop:cumulative>
        <xmlsnoop:atomic>
          <uni:reg_open>
          <xmlsnoop:variable name="Subj" select="$event/@Subject" />
          </uni:reg_open>
        </xmlsnoop:atomic>
        <xmlsnoop:atomic> <uni:register subject="$Subj" /> </xmlsnoop:atomic>
        <xmlsnoop:atomic> <uni:reg_close subject="$Subj" /> </xmlsnoop:atomic>
      </xmlsnoop:cumulative>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

Note the namespaces: eca for the rule level, xmlsnoop for the event algebra level (which also supports the variables) and uni for the domain level.

The event component collects the relevant events, and returns them as a sequence, resulting in the following variable bindings:

```
<variable-bindings>
  <tuple>
    <variable name="regseq">
      <reg_open subject="Databases" />
      <register subject="Databases" name="John Doe" />
      <register subject="Databases" name="Scott Tiger" />
        :
      <reg_close subject="Databases" />
    </variable>
    <variable name="Subj">Databases</variable>
  </tuple>
</variable-bindings>
```

5.3 The Query Component

The query component is concerned with *static* information that is obtained and restructured from analyzing the data that has been collected by the event component (in the variable bindings), and based on this data, stating queries against databases and the Web. Whereas the firing of the rule due to a successful detection of an event results in exactly one tuple of variable bindings, the query component is very similar to the evaluation of database queries and rule bodies in Logic Programming: in general, it results in a set of tuples of variable bindings. We follow again the Logic Programming specification that every answer produces a variable binding. For variable binding by matching (as in Datalog, F-Logic, XPathLog, Xcerpt etc.), this is obvious. Since we also allow variable bindings in the functional XSLT style, the semantics is adapted accordingly:

- each answer node of an XPath expression yields a variable binding;
- each node that is *return*ed by an XQuery query yields a variable binding; if the XQuery query is of the form
  <name>{ for ... where ... return ...} *</name>* ,
  then the whole result yields a single variable binding.

Example 2. Consider again Example 1 where the resulting event contained several registrations of students. The names of the students can be extracted as multiple string-valued variables:

```
<eca:rule ... >
  ... same as above, binding variables "Subj" and "regseq" ...
  <eca:variable name="Student">
    <eca:query>
      <eca:opaque lang='xpath'>
```

```
      $regseq//register[@subject=$Subj]/@name/string()
    </eca:opaque>
  </eca:query>
</eca:variable>
  ⋮
</eca:rule>
```

The above query generates the extended variable bindings
$\beta_1 = \{Subj \rightarrow \text{'Databases'}, regseq \rightarrow \text{(as above)}, Student \rightarrow \text{'John Doe'}\}$,
$\beta_2 = \{Subj \rightarrow \text{'Databases'}, regseq \rightarrow \text{(as above)}, Student \rightarrow \text{'Scott Tiger'}\}$.

5.4 The Test Component

In general, the evaluation of conditions is based on a logic over literals with boolean combinators and quantifiers. A Markup Language exists with FOL-RuleML [BDG+]. Instead of first-order atoms, also "atoms" of other data models can be used. Note that XPath expressions are also literals that result in a true/false (true if the result set is non-empty) value. The result of the test component is the set of tuples of variable bindings that satisfy the condition (for further propagation to the action part).

5.5 The Action Component

The action component is the one where *actually* something is done in the ECA rule: for each tuple of variable bindings, the action component is executed. The action component may consist of several <eca:action> elements which contain action specifications, possibly in different action languages, e.g., the CCS process algebra [Mil83]. This can be updates on the database level, explicit message sending (e.g. to Web Service calls), or actions on the ontology level (that must then be implemented appropriately). The semantics is that all actions are executed.

6 Conclusion

We described the concepts and proposed an XML markup for a general ECA-rule framework for the Semantic Web, taking into account the heterogeneity of (existing) languages. By using the namespace/URI mechanism for identifying the languages, also appropriate services can be located. Such an architecture based on this framework is described in [MAA05].

Although the above examples all used "syntactical" languages in XML term markup for the components, also languages using a semantical, e.g., OWL-based representation (which have to be developed) can be used.

There are several issues that are explicitly not dealt with in our approach – because they are encapsulated inside (and "bought with") the concepts to be integrated: The detection of complex events is done and provided by the individual event languages and their engines – the framework provides an environment for

embedding them. In the same way, query evaluation itself is left to the original languages and processors to be embedded into the global approach; also actual execution of actions (and transactions) is left with the individual solutions.

References

[BBCC02] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Intl. Conference on Data Engineering (ICDE)*, pp. 403–418, 2002.

[BCP01] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *WWW Conf. (WWW 2001)*, 2001.

[BDG⁺] H. Boley, M. Dean, B. Grosof, M. Sintek, B. Spencer, S. Tabet, and G. Wagner. FOL RuleML: The First-Order Logic Web Language. `http://www.ruleml.org/fol/`.

[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.

[BPW02] J. Bailey, A. Poulovassilis, and P. T. Wood. An Event-Condition-Action Language for XML. In *WWW Conf.*, 2002.

[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, 2002.

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

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *ACM SIGMOD*, pp. 134–146, 1989.

[MAA05] 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)*, to appear in Springer LNCS, 2005.

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

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, pp. 267–310, 1983.

[PPW03] G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Workshop on Semantic Web and Databases (SWDB'03)*, 2003.

[PPW04] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action Rule Languages for RDF. In *Hellenic Data Management Symposium (HDMS'04)*, 2004.

[RML] Rule Markup Language (RuleML). `http://www.ruleml.org/`.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Halevy, and D. Weld. Updating XML. In *ACM SIGMOD*, pp. 133–154, 2001.

[XML00] XML:DB. XUpdate - XML Update Language. `http://xmldb-org.sourceforge.net/xupdate/`, 2000.