An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web

Wolfgang May[1] and 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. The Web of today can be seen as an active and heterogeneous infrastructure of autonomous systems, where reactivity, evolution and propagation of information and changes play a central role. In the same way as the main driving force for XML and the Semantic Web idea was the heterogeneity of the underlying data, the heterogeneity of concepts for expressing behavior calls for an appropriate handling on the semantic level. We present an ontology-based approach for specifying behavior in the Semantic Web by Event-Condition-Action (ECA) rules that models rules as well as their event, condition, and action components, and languages as resources. The necessary information about semantics and suitable processors is then associated with the language resources. The approach makes use of the data integration facilities by URIs that allow for a seamless integration of information and services physically located at different places. Additionally, that point of view allows for sharing and reuse of these resources throughout the Semantic Web.

1 Introduction

The current Web does not only consist of HTML pages, but of *nodes*, some of which are still browsing-oriented, but in general also providing behavior (often summarized as *Web services*). With this, the perspective shifts more to the idea of the Web as a network of (autonomous) *information systems*. Current portals usually integrate a fixed set of known sources, often using "hard-coded" integration. A problem when overcoming this restriction is its *heterogeneity*, both in the actual data formats, and also *semantic heterogeneity*. The goal of the *Semantic Web* is to bridge this heterogeneity and provide unified view(s) on the Web. 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 force for XML and the Semantic Web idea was the heterogeneity of the underlying data, the heterogeneity of concepts for expressing *behavior* requires an appropriate handling on the semantic level. Since the contributing nodes are prospectively 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.

Here, *reactivity* and its formalization as *Event-Condition-Action (ECA) rules* provide a suitable model because they provide a modularization into clean concepts with a well-defined information flow. An important advantage of them is that the *content* (event, condition, and action specifications) is separated from the *generic semantics* of the rules themselves. They are easy to understand, and provide 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 develop an ontology-based approach for describing (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, as well as application-specific languages and ontologies for atomic events and actions. Modularity allows for high flexibility wrt. the heterogeneity of the potential sublanguages, while exploiting and supporting their meta-level *homogeneity* on the way to the Semantic Web.

Structure of the Paper. The remainder of the paper is structured as follows: In Section 2, we analyze the notion of *state* in the Semantic Web and the consequences for the design of the ECA framework for describing *behavior* in the Semantic Web. The rule level of the ontology is then presented in Section 3, including the coarse level of an XML Markup. Section 4 deals with the integration of trigger-like ECA rules "below" the semantical level in the homogeneous local environments of nodes. The Semantic Web level is then refined with a more detailed analysis of the event, query, condition, and action concepts in Section 5; leading to a refined XML Markup. The architecture of the realization of the framework in the Semantic Web based on the "actual" *resources* (e.g., language processors that are associated with the language resources) of the RDF ontology is described in Section 6, followed by a short conclusion.

2 Rules in the Semantic Web: Requirements Analysis

2.1 States and Nodes in the Semantic Web

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 and conditions.

The state of a node in the Semantic Web is represented in XML, RDF(S), and/or OWL. Usually, XML serves for the physical level, mapped to an RDF/OWL

ontology for integration throughout the Web. Here, a framework where the behavior base (i.e., the ECA rules) is also part of the Semantic Web and represented in a declarative way (and can be queried, reasoned about, and updated), will in our view prove useful.

2.2 Behavior of Nodes

Cooperative and reactive behavior is then based on events (e.g., an update at a data source where possibly others depend on). The depending resources detect events (either they are delivered explicitly to them, or they poll them via the communication means of the Semantic Web). Then, conditions are checked (either simple data conditions, or e.g. tests if the event is relevant, trustable etc.), which can include queries to one or several nodes. Finally, an action is taken (e.g., updating own information accordingly).
The behavior has to take into account the distributed state of knowledge:

1. behavior can be local to a node, i.e., all events are explicitly detectible at the node, the condition is a query against local data, and the actions are also local ones;
2. conditions can include queries that may involve other nodes (either explicitly addressed, or by evaluating a Semantic Web query);
3. actions can also effect other nodes; again either by sending an explicit message to a certain node, or as an *intensional* update against "the Web". Such updates are expressed and sent as messages, and appropriate nodes will react upon their receipt by updating their local database;
4. there are also relevant events that are only detectible at other nodes, or intensional events "on the Web"; also, event combinations (from possibly different sources) have to be taken into account.

With these extensions, together with the "Semantic" property of the rules, the ECA concept 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 query languages, the action languages or the event languages/event algebras have to be integrated in a common framework. In contrast to "classical" ECA rules, our approach makes a more succinct separation between event, condition, and action component, which are possibly (i) given in separate languages, and (ii) possibly evaluated/executed in different places. Since not every node will provide ECA capabilities, there will also be nodes that provide "ECA services" (extending the concepts of *publish-subscribe* and *continuous query* services), being able to execute ECA rules (submitted by any Semantic Web participants) that use arbitrary sublanguages (see Section 6).

The requirement (4) also calls for application-dependent handling and detection of atomic events. In general, each application ontology must also specify how derived events can be detected based on actual events (e.g., "account X goes below zero" based on "a debit to account X", or "flight LH123 on 10.8.2005 is fully booked" from "person P is booked for seat 5C of flight LH123 on 10.8.2005").

3 Ontology of Rules and Languages

In usual Active Databases in the 1990s, an ECA language consisted of an event language, a condition language, and an action language. In the Semantic Web, there is a heterogeneous world of such languages which has to be covered. The target of the development and definition of languages for (ECA) rules, events, conditions and actions in the Semantic Web should be a semantic approach, i.e., based on an (extendible) ontology for these notions that allows for *interoperability* and also turns the instances of these concepts into objects of the Semantic Web itself. Thus, in the Semantic Web, we do not have a unique ECA language that consists of three such languages, but there is the semantic concept of an *ECA rule* as shown as an UML diagram in Figure 2. The event, query/test, and action components are described in appropriate languages, and ECA rules can use and combine such languages flexibly. The model is accompanied by an XML ECA rule (markup) language, called ECA-ML.

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, *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 probabyl (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 probably obtaining additional information by a query and then evaluating a *boolean* test, and, if “yes”, then actually *do* something – as shown in Figure 1.

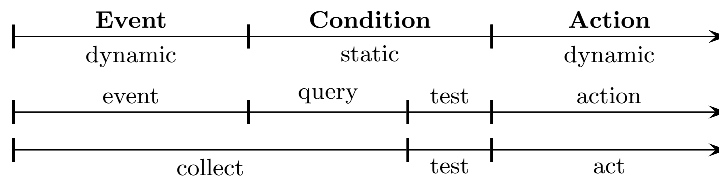


Fig. 1. Components and Phases of Evaluating an ECA Rule

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, e.g., associating them with a processor,

– there is a well-defined *interface* for communication between the E, Q&T, and A component by variables (e.g., bound to XML or RDF fragments).

Sublanguages and Interoperability. For applying such rules in the Semantic Web, a uniform handling of the event, query, test, and action sublanguages is required. For this, rules and their components must be objects of the Semantic Web, i.e., described in XML or RDF/OWL in a generic *rule ontology* describing the UML model shown in Figure 2.

The modular structure requires a communication of parameters between the rule components. We propose to use rule-wide logical variables for a communication flow according to Figure 1: Variables can be bound in the event component and in the subsequent query component; previously bound variables can also be used here. Variables occurring several times are interpreted as join variables, requiring all occurrences to be bound to the same value. The test and action components use these variables.

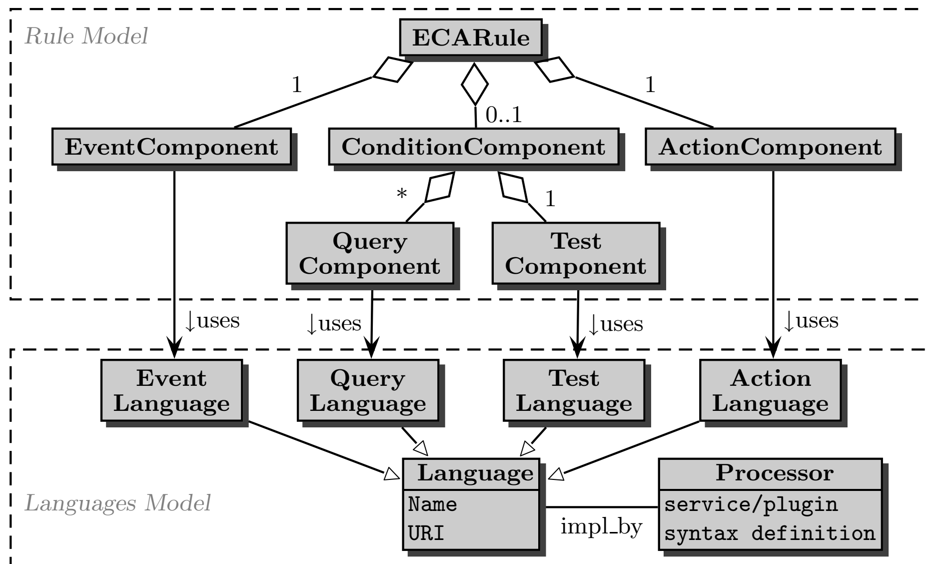


Fig. 2. ECA Rule Components and Corresponding Languages

Based on this ontology, we propose the following markup (ECA-ML):

<**eca:rule** *rule-specific attributes*>
rule-specific contents, e.g., declaration of logical variables
<**eca:event** *identification of the language* >
event specification, probably binding variables; see Section 5.2
<**/eca:event**>
<**eca:query** *identification of the language* > <!-- there may be several queries -->
query specification; using variables, binding others; see Section 5.3
<**/eca:query**>
<**eca:test** *identification of the language* >
condition specification, using variables; see Section 5.3

```
  </eca:test>
  <eca:action identification of the language >
    action specification, using variables, probably binding local ones; see Section 5.3
  </eca:action>
</eca:rule>
```

The actual languages (and appropriate services etc.) are identified by namespaces and their declarations in the <eca:...> elements (see Example 2 later).

A similar markup for ECA rules (without separating the query and test components) has been used in [BCP01] with *fixed* languages (a basic language for atomic events on XML data, XQuery as query+test language and SOAP in the action component). This fixed approach falls short wrt. the language heterogeneity, and especially the use and integration of languages for composite events. The XChange approach [BP05] also uses fixed languages for specifying the event, condition, and action component. 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.

Languages, Rules, and Rule Components as Resources. For the Semantic Web, both the languages, and the instances (i.e., rules, rule components, and also e.g. subevents) are *resources*. This allows to *reuse* rules and to recombine subparts (e.g., events) in several rules.

For the architecture, we propose to use a completely modular concept: the framework for the ECA rules must allow to *plug in* or *connect to* detection engines for events (atomic events such as simple data updates or incoming messages, application-level events and *composite events* such as "if first A happens and then B"), processors for queries, and processors for actions (including updates, intensional updates, and transactions): For each rule, the event, query, test, and action components contain *references* to the corresponding languages as resources, given as a URI, similar to XML's namespaces. These resources in turn are associated with further resources related to the language, e.g., a DTD or XML Schema, an ontology description (e.g., in OWL), and a language processor (e.g., as a Web Service). We come back to this issue in Section 6 where we show that in the end, the processing of each contributing sub-ontology can be associated with separate engines or nodes in the Web, i.e. (anticipating the analysis of the subsequent sections):

- ECA rule processors
- underlying database engines with local, trigger-like rules,
- detection mechanisms for (application-independent) event algebras,
- for each application ontology, detection mechanisms of application-level events,
- query processors,
- services for Web transactions, and
- application-level actions (explicit or intensional actions).

ECA Rules in the Semantic Web are required on several abstraction levels (programming language/data structure level, logical level, and semantic level), and with different scope (local or global). In most of the rules on the higher levels (global, referring to the application ontology), the event, query, test, and action components are subject to heterogeneity. But, there are also mostly simple local rules e.g., for maintaining local consistency, mappings to the underlying data model, and reactions, that work in a homogeneous, local environment, where the "classical" ECA paradigm is sufficient. We first integrate these *trigger-like* rules into the framework and come back to the heterogeneous Semantic Web case in Section 5.

4 Trigger-Like Rules

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 format. While the SQL triggers in relational databases are only able to react on changes of a 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.

Events on XML Data. Work on triggers for XQuery has e.g. been described in [BBCC02] with *Active XQuery* (using the same syntax and switches as SQL, with XQuery in the action component) and in [BPW02,PPW03], emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases. In [ABB$^+$05], we developed the following proposal for atomic events on XML data:

- `ON {DELETE|INSERT|UPDATE} OF` *xsl-pattern*: if a node matching the *xsl-pattern* is deleted/inserted/updated,
- `ON MODIFICATION OF` *xsl-pattern*: if anything in the subtree rooted in a node matching the *xsl-pattern* is modified,
- `ON INSERT INTO` *xsl-pattern*: if a node is inserted (directly) into a node matching the *xsl-pattern*,
- `ON {DELETE|INSERT|UPDATE} [SIBLING] [IMMEDIATELY] {BEFORE|AFTER}` *xsl-pattern*: if a node (optionally: only sibling nodes) is modified (immediately) before or after a node matching the *xsl-pattern*.

All triggers should make relevant values accessible, e.g., `OLD AS ...` and `NEW AS ...` (like in SQL), both referencing the node to which the event happened, additionally `INSERTED AS`, `DELETED AS` referencing the inserted or deleted node.

Similar to the SQL `STATEMENT` and `ROW` triggers, the granularity has to be specified for each trigger:

- `FOR EACH STATEMENT` (as in SQL),

– `FOR EACH NODE`: for each node in the *xsl-pattern*, the rule is triggered only at most once (cumulative, if the node is actually concerned by several matching events) per transaction,
– `FOR EACH MODIFICATION`: each individual modification (possibly for some nodes in the *xsl-pattern* more than one) triggers the rule.

The implementation of such triggers in XML repositories can e.g. be based on the *DOM Level 2/3 Events* or on the triggers of relational storage of XML data.

Events on RDF Data. RDF triples describe properties of a resource. In contrast to XML, there is no subtree structure (which makes it impossible to express "deep" modifications in a simple event), but there is metadata. A proposal for RDF events can be found in RDFTL [PPW03,PPW04]. The following proposal has been developed in [ABB+05]:

– `ON {INSERT|UPDATE|DELETE} OF` *property* `[OF` *class*`]`.

If a property is added to/updated/deleted from a resource (optionally: of the specified class), then the event is raised. Additionally,

– `ON {CREATE|UPDATE|DELETE} OF` *class* is raised if a resource of a given class is created, updated or deleted.

On the RDF/RDFS level, also metadata changes are events:

– `ON NEW CLASS` is raised if a new class is introduced,
– `ON NEW PROPERTY [OF CLASS` *class*`]` is raised, if a new property (optionally: to a specified class) is introduced.

Besides the `OLD` and `NEW` values mentioned for XML, these events should consider as arguments (to bind variables) Subject, Property, Object, Class, Resource, referring to the modified items (as URIs), respectively.
Trigger granularity is `FOR EACH STATEMENT` or `FOR EACH TRIPLE`.

Integrating Triggers into the ECA Ontology: Opaque Rules. In the above trigger-like cases, the languages for specifying the *event*, *condition* and *action* components are the database-level events, and the local query and update languages. For that, from the implementation point of view, the trigger rule as a whole does not require an explicit markup but can be expressed in its native syntax. In our ontology, we embed this as *opaque* rules, see Figure 3 and the following XML markup:

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

Since such opaque rules are ontologically "atomic" objects, their event, condition, and action components cannot be accessed by Semantic Web concepts. Thus,

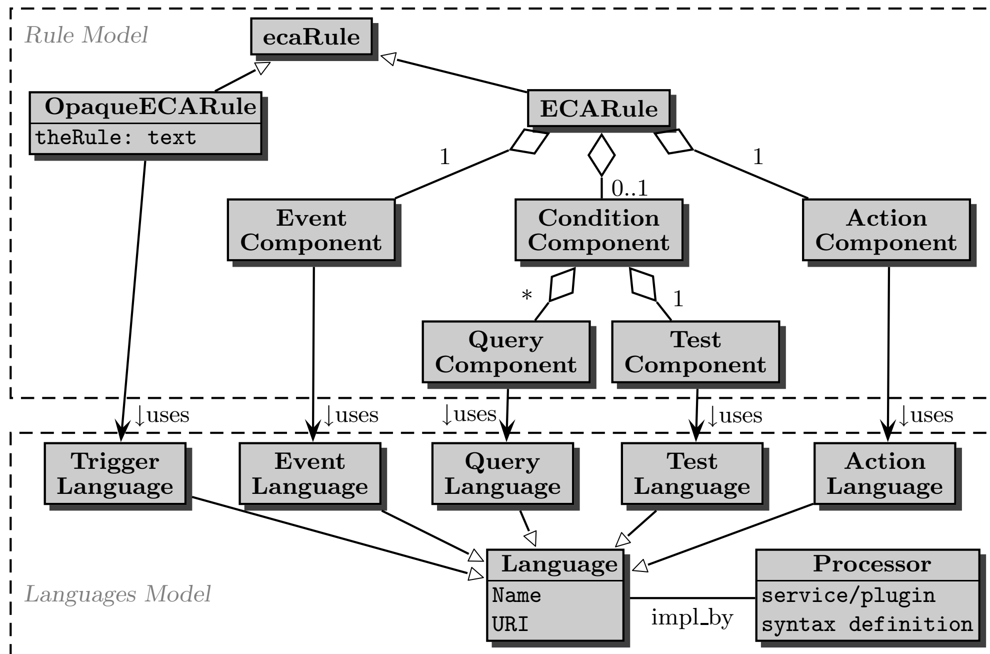


Fig. 3. ECA Rule Components and corresponding Languages II

there is no reuse, and no support for rule analysis. The database-level events on XML or RDF data can also be seen as atomic events in the sense of non-opaque ECA rules; thus, it is also possible to *lift* the triggers to the ontology level and represent them as full-fledged ECA rules.

5 Ontologies for the Rule Components

In the subsequent sections, we develop more detailed ontologies for the event, query, test, and action components, with special emphasis on the event component (the others are similar).

5.1 Ontology of Events

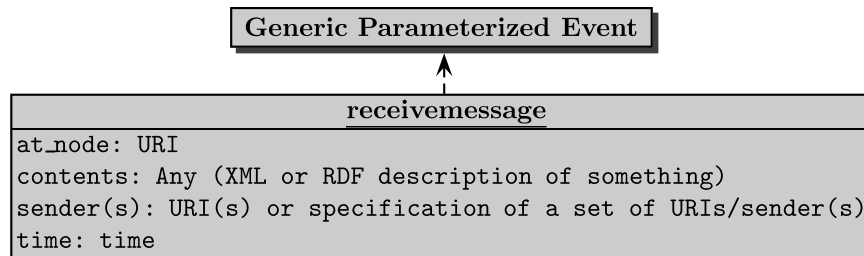
The ontological structure of the event component consists of three subclasses (see explanations below and Figure 4):

- atomic events, that again split into several subtypes:
  - data level events as those discussed in Section 4,
  - events of a given application domain, e.g., in banking, travel organizing, administration; such atomic events are described in terms of the ontologies of the application domain,
  - generic parameterized events that instantiate generic event patterns, e.g., "receive a message about ...".

– *composite* events, e.g.: "A or B", "A and B", or "A and then B".

Atomic Application Level Events. Atomic application-level events are the visible happenings in the application domain. Note that in contrast to the above data-level trigger events on XML or RDF data, 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 debit of 200E from Alice's bank account", "a change of Alice's bank account" (that are 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. Note that application ontologies have to describe the relationship between actions and resulting events. Orthogonal to being derived or not, application-level atomic events can be associated with a certain node (e.g., "if Springer publishes a textbook on the Semantic Web") or can describe happenings on the Web-wide level (e.g., "if a textbook on the Semantic Web is published"). In the latter case, event detection is even more complicated since it must also be searched and derived *where* and *how* the event can be detected. This is not the task of the rule execution, but of application-level reasoning, based on the application ontology. With this, *application-level* rules (i.e. reacting on application-level *global* events) like *business rules* can be described.

Generic Parameterized Events. Generic Parameterized Events are patterns of atomic events that are ontologically independent from the actual application. The most prominent ones are concerned with communication, i.e., receiving and sending messages, or transactional events. Note that also sending of a message can be a relevant event to trigger other rules, e.g., for policies (waiting for an answer for 10 minutes, then sending it again), or "listening" and deriving other events. In general, such events are associated with a certain node.



The specification of such an event can e.g. be used in a rule like "in case that I receive a message from my bank with my account statement that contains a debit of more than 1000E then ...", where the occurrence of a generic event is restricted further wrt. its content. The receipt of the message is an event that can be detected by the communication service of a node, whereas the additional test must be checked on the application level.

Composite Events: Event Algebras. Event algebras, well-known from the Active Database area, serve for specifying *composite* events by defining *terms*

formed by nested application of operators over *atomic* events. Each operator has a semantics that specifies what the composite event means. Detection of a composite event means that its "final" atomic subevent is detected, e.g., as in [CKAK94]:

$$(1)\ (E_1 \nabla E_2)(t) \ :\Leftrightarrow\ E_1(t) \vee E_2(t)\ ,$$
$$(2)\ (E_1 \triangle E_2)(t) \ :\Leftrightarrow\ E_1(t) \wedge E_2(t)\ ,$$
$$(3)\ (E_1; E_2)(t) \ :\Leftrightarrow\ \exists t_1 \leq t : E_1(t_1) \wedge E_2(t).$$

Event algebras contain not only the aforementioned straightforward basic connectives, but also additional operators. A bunch of event algebras have been defined that provide also e.g. "negated events" in the style that "when E_1 happened, and then E_3 but not E_2 in between, then do something", "periodic" and "cumulative" events, e.g., in the SNOOP event algebra [CKAK94] of the "Sentinel" active database system. Some preliminary work on composite events in the Web is presented in [BKK04], but that only considers composition of events of modification of XML data.

Example 1 (Cumulative Event, [CKAK94]). A "cumulative aperiodic event"

$$A^*(E_1, E_2, E_3)(t) \ :\Leftrightarrow\ \exists t_1 \leq t : E_1(t_1) \wedge E_3(t)$$

occurs with E_3 and then requires the execution of a given set of actions corresponding to the occurrences of E_2 in the meantime. Thus, it is not a simple event, but more an active rule, stating a temporal implication of the form "if E_1 occurs, then for each occurrence of an instance of E_2, collect its parameters, and when E_3 occurs, report all collected parameters (in order to do something)".

A cumulative periodic event can be used for "after the end of a month, send an account statement with all entries of this month":

$E(Acct) :=$
$A^*(\mathit{first_of_month}(m), (\mathit{debit(Acct,Am)} \nabla \mathit{deposit(Acct,Am)}), \mathit{first_of_month}(m{+}1))$

where the event occurs with *first_of_next_month* and carries a *list* of the debit and deposit actions.

5.2 Ontology of the Event Component.

With this ontology, the event component may consist of a combination of one ore more event algebras, using atomic events of one or more applications, and possibly atomic data-level events from several data models, and some generic parameterized events (see Figure 4).

An important matter here is that all components of an event specification can be associated with the appropriate components of the language using *identifiers.* This identification is provided for the XML Markup level by *namespaces* and their URIs, and for the RDF level directly by URIs (see also Section 6).

XML Markup for the Event Component. The eca:event elements contain elements according to an event algebra language (identified by its namespace), and embedded into this, eca:atomic-event elements are the "leaves" of the event language level. Inside of eca:atomic-event elements, the namespaces of the applications are used for the actual atomic event patterns. Whenever an atomic event

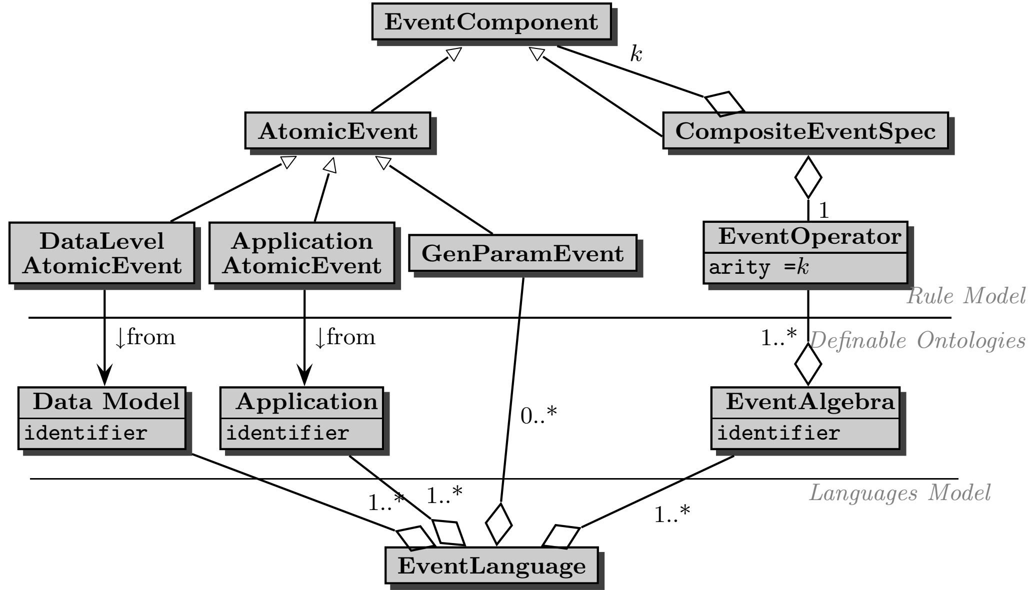


Fig. 4. Event Component Ontology

matches such a pattern, it (i.e., its XML or RDF representation) is bound to the temporary variable $event. eca:bind-variable elements inside eca:atomic-event elements allow for binding rule variables by using $event.

Example 2. Consider the event of Example 1. Atomic events are (i) temporal events that are assumed to be provided/signalled by some service, e.g. as <temporal:first-of-month month="5" year="2005"/>, and (ii) events of the banking application, provided as e.g.,

<banking:deposit account="1234"> <amount>200</amount> </banking:deposit> .

```
<eca:rule>
  <eca:variable name="account" select="arguments[1]"/>
  <eca:variable name="list"/>
  <eca:variable name="month"/>
  <eca:event  xmlns:snoop="uri of the snoop event algebra"
             xmlns:banking="uri of the banking ontology"
             xmlns:temporal="uri of some web service" >
    <snoop:cumulative-event cumulative-result="list"/>
      <snoop:cumulative-start> <eca:atomic-event>
        <temporal:first-of-month>
          <eca:bind-variable name="month" select="$event/@month"/>
        </temporal:first-of-month>
      </eca:atomic-event> </snoop:cumulative-start>
      <snoop:cumulative-collect> <snoop:disjunctive>
```

```
      <eca:atomic-event> <banking:debit account="$account"/>
        <eca:bind-variable name="list" select="$event"/>
      </eca:atomic-event>
      <eca:atomic-event> <banking:deposit account="$account"/>
        <eca:bind-variable name="list" select="$event"/>
      </eca:atomic-event>
    </snoop:disjunctive> </snoop:cumulative-collect>
    <snoop:cumulative-end> <eca:atomic-event>
      <temporal:first-of-month month="$month+1"/>
    </eca:atomic-event> </snoop:cumulative-end>
  </snoop:cumulative-event>
</eca:event>
  :
</eca:rule>
```

Note that the cumulative event defines the variable list to be cumulative, i.e., for each <eca:bind-variable name="list" select="$event"/>, the event (as XML element) is *appended*.

5.3 Query, Test, and Action Ontologies

The ontologies for the query, test, and action components follow a similar design.

Queries. Queries can be queries against individual nodes, or against "the Web". Here, existing languages like XPath, XQuery, or RDQL that are commonly supported can be used. Such languages that are based on a kind of basic expressions and algebraic operators use a classical tree markup. Since query languages are in general supported in the Web nodes themselves, there is in general no need for specific services; often, the query component is *opaque*.

Tests. Tests in general use boolean operators and quantifiers which are already covered in Markup Languages like, e.g., FOL-RuleML [BDG$^+$] for formulas in first-order logic. Instead of first-order atoms, also "atoms" of other data models can be used, employing identifiers in the same way as for the event component. Since all relevant information is gathered in the event and query components, the test is evaluated locally.

Actions. The action component is similar to the event component: we distinguish atomic actions on the database level (updates expressed in DOM, XUpdate, XQuery+Updates, or in an RDF update language), generic actions (sending messages with some content), and execution of composite actions, even as *transactions on the Web*. Actions here also include intensional updates on the semantic level (that must be translated into actual updates at certain nodes). The actual processing of transactions and intensional updates is independent from this framework. Similar to the definition of composite events, composite actions and transactions can be defined e.g. in the style of CCS [Mil83], augmented with transactional commands.

6 Rules, Components, Languages and Processors as Resources

Rules on the semantic level, i.e., RDF or OWL, lift ECA functionality wrt. two (independent) aspects: first, the events, queries, and actions refer to the RDF/ontology level. An even higher level regards rules themselves as objects of the Semantic Web: rules are specified in RDF/OWL using the above rule ontology and event, query, test, and action subontologies.

Reuse: Rules and Rule Components as Resources. The above ontology directly leads to a resource-based approach: every rule, rule component, event, subevent etc. becomes a resource. Every rule is then interpreted as a network of RDF resources of the contributing ontologies (ECA, event algebras, applications etc.). By this, e.g. collections of (sub)events as well as complete (application-specific) rule bases can be designed, published by associating them with a URI, and reused. Figure 5 shows the rule and the event component given in Example 2, combining two application-independent language ontologies:

- the ECA ontology (gray, doublelined), and the SNOOP ontology of the event algebra (incorporating the semantics of the SNOOP operators; gray),

and two application-dependent ontologies:

- the banking application-level ontology: there, the semantics of the atomic events defined in this ontology must be available (diagonally crosshatched).
- the temporal ontology: there, information about temporal events is available (crosshatched).

6.1 Information behind the External Resources

The external resources (SNOOP, banking, and temporal domain) must be associated with further resources of the respective ontologies that are used when actually processing rules:

- Sublanguage ontologies (here, SNOOP): URI or a service for processing the language, e.g., a Web Service where the composite event specification can be registered, and that, when informed about relevant events, runs the detection and informs the client about success (transferring the result parameters). (Similar for languages for composite actions.)
- Temporal domain: URI of a service that provides the relevant atomic events.
- Application ontologies (here, banking domain): in most cases, the "client" knows which "server" (e.g., the bank where the account is located) provides the relevant atomic events. For other ontologies such as stocks or travel, the resource that is "responsible" for the ontology could also provide notification services for atomic events. In either case, derived events (that can locally use another event algebra) have to be defined there (since their definition conceptually also belongs to the ontology, this is not surprising). In the same way, a service for execution of atomic actions and the definition of composite actions (using any action language) can be provided.

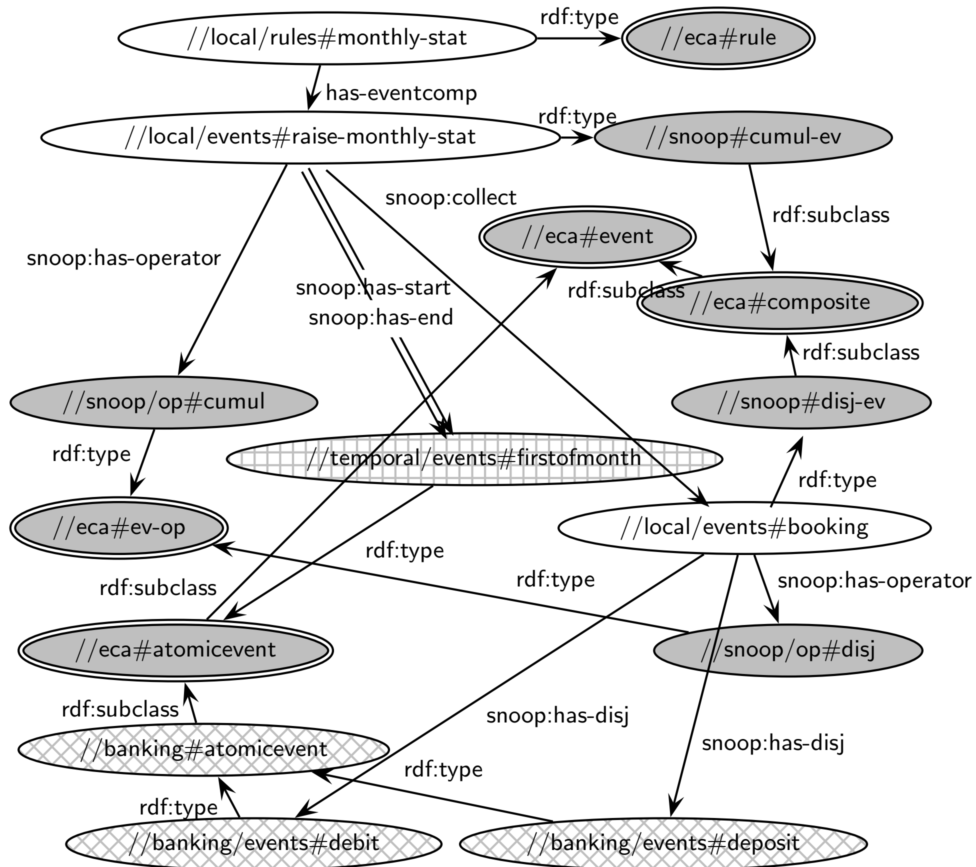


Fig. 5. Example Rule and Event Component as Resources

A Modified Rule using a Derived Event. The banking ontology could define a derived booking event as the disjunction of debit and deposit. The rule could then directly use this derived event. In the ontology diagram, the only difference would be that the booking node is //banking/events#booking and appears diagonally crosshatched (and its semantical information must be kept at the banking resource – but in this case it can also be used in specifications that do not know the SNOOP language).

6.2 Architecture and Processing: Cooperation between Resources

Rules can be evaluated locally at the nodes where they are stored, or they can be registered at a *rule evaluation service*. The rule evaluation engine manages the actual handling of rules based on the language URI references. As described above, every subconcept (i.e., events, queries, tests, and actions) carries the information of the actual language it uses in its xmlns:*namespace* URI attribute (note that this even allows for nested use of operations of *different* event alge-

bras). Assume the case where the language processors are available at these URIs as a Web Services. For event detection (and analogously, execution of composite actions), at least two resources (or services) must cooperate: Event detection splits into the *event algebra* part (that is detected algorithmically by a resource representing a *language* ontology, e.g., SNOOP) and the atomic events of the application ontology. Thus, the algebra processor must be notified about the atomic events. This can be done in several ways:

Straightforward: The "straightforward" way is that the client C organizes the communication between the event generator(s) and the event algebra processor (see Figure 6): C registers rules to be "supervised" at a rule execution service R. For handling the event component, R reads the language URI of the event component, and registers the event component at the appropriate event detection service S (note that a rule service that evaluates rules with events in different languages can employ several event detection mechanisms).

During runtime, the client C forwards all received events to R, that in turn forwards them to all event detection engines where it has registered event specifications for C, amongst them, S. S is "application-unaware" and just implements the semantics of the event combinators for the incoming, non-interpreted events. In case that a (composite) event is eventually detected by S, it is signalled together with its result parameters to R. R takes the variables, and evaluates the query&test (analogously, based on the respective languages), and finally executes the action (or submits the execution order to a suitable service).

(Note that this strategy can be extended towards "selecting" and brokering events according to their namespace in a similar way to the architectures described below.)

Application-centered: The client submits its composite event specification to a service that is aware of all relevant events in the application domain. This service then employs an appropriate event detection service by registering the event specification, and informing it about the atomic events (e.g., "@bank: please trace the following composite event in language L on my account" (and employ a suitable event detection service for L)).

Language-centered: When a rule or an event specification is submitted for registration, this has to be accompanied by information on which resource(s) provide the atomic events (e.g., "@snoop: my bank is at *uri*, please supervise my account and tell me if a composite event *ev* occurs), or the detection service even has to find appropriate event sources (by the namespaces of the atomic events). The detection service then contacts them directly. This proceeding is e.g. appropriate for booking travels where the client is in general not aware of all relevant events (e.g., "@snoop: you know better than me who is well-informed about events relevant for traveling, please detect the event ev_{travel} for me"), as illustrated in Figure 7: A client registers a rule (in the travel domain) at R (Step 1.1). R again submits the event component to the appropriate event detection service S ((1.2), here: snoop). Snoop looks at the namespaces of the atomic events and sees that the travel ontology is relevant. The snoop service contacts a travel event broker (1.3) who keeps it informed (2.2) about atomic events (e.g.,

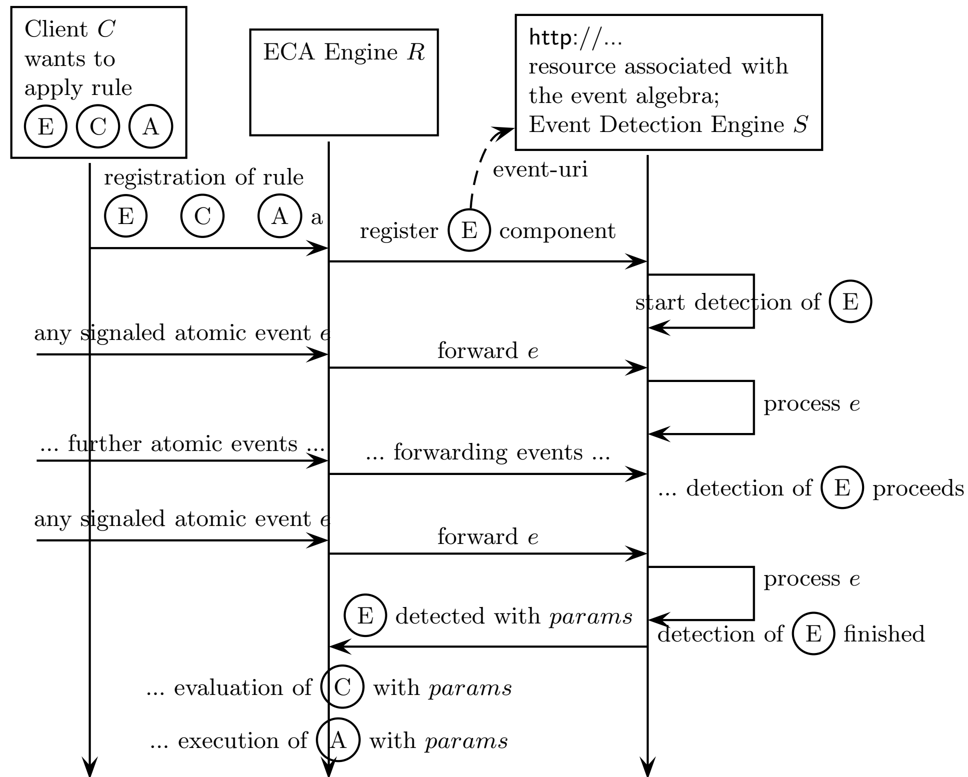


Fig. 6. Straightforward Communication
(UML-style sequence diagram, temporal axis downwards)

happening at Lufthansa (2.1a) and SNCF (2.1b)). Only after detection of the registered composite event, S submits the result to R (3) that then evaluates the Q&C component, and probably executes some actions (4.1, 4.2).

7 Conclusion

We have presented a modular ontology-based framework for ECA rules. The ontology does not only *describe* the domain, but by including the processing resources on the Web also provides the infrastructure for actual implementation of the framework. Different alternatives allow for service-oriented strategies. More detailed aspects are currently investigated, and an implementation has been started, stepwise extending from XML-level ECA rules and services to the above framework. The modularization of the approach allows for dealing with many research issues separately.

Acknowledgement. This research has been funded by the European Commission within the 6th Framework Programme project Rewerse, number 506779 (cf. `http://rewerse.net`).

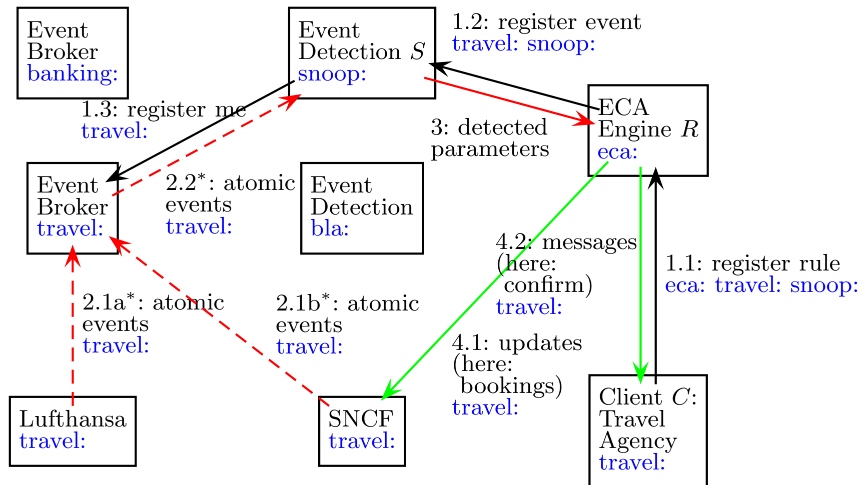


Fig. 7. Language-Centered Communication

References

[ABB+05] J. J. Alferes, M. Berndtsson, F. Bry, M. Eckert, W. May, P. L. Pătrânjan, and M. Schröder. Use Cases in Evolution and Reactivity. Technical Report I5-D2, REWERSE EU FP6 NoE, 2005. Available at `http://www.rewerse.net`.

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

[BCP01] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *Int. WWW Conference*, 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/`.

[BKK04] M. Bernauer, G. Kappel, and G. Kramler. Composite Events for XML. In *Int. WWW Conference*, 2004.

[BP05] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. *20th ACM Symp. Applied Computing*. 2005.

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

[CKAK94] 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*, pp. 606–617, 1994.

[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.