

Evolution and Reactivity in the Semantic Web

José Júlio Alferes¹, Michael Eckert², and Wolfgang May³

¹ CENTRIA, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal

² Institut für Informatik, Ludwig-Maximilians-Universität München, Germany

³ Institut für Informatik, Georg-August-Universität Göttingen, Germany

Abstract. Evolution and reactivity in the Semantic Web address the vision and concrete need for an active Web, where data sources evolve autonomously and perceive and react to events. In 2004, when the REVERSE project started, regarding work on Evolution and Reactivity in the Semantic Web there wasn't much more than a vision of such an active Web.

Materialising this vision requires the definition of a model, architecture, and also prototypical implementations capable of dealing with reactivity in the Semantic Web, including an ontology-based description of all concepts. This resulted in a general framework for reactive Event-Condition-Action rules in the Semantic Web over heterogeneous component languages.

Inasmuch as heterogeneity of languages is, in our view, an important aspect to take into consideration for dealing with the heterogeneity of sources and behaviour of the Semantic Web, concrete homogeneous languages targeting the specificity of reactive rules are of course also needed. This is especially the case for languages that can cope with the challenges posed by dealing with composite structures of events, or executing composite actions over Web data.

In this chapter we report on the advances made on this front, namely by describing the above-mentioned general heterogeneous framework, and by describing the concrete homogeneous language XChange.

1.1 Introduction

The Web and the Semantic Web, as we see it, can be understood as a “living organism” combining autonomously evolving data sources, each of them possibly reacting to events it perceives. The dynamic character of such a Web requires declarative languages and mechanisms for specifying the evolution of the data, and for specifying reactive behaviour on the Web.

Rather than a Web of data sources, we envisage a Web of information systems where each such system, besides being capable of gathering information (querying, both on persistent data, as well as on volatile data such as occurring events), can possibly update persistent data, communicate the changes, request

changes of persistent data in other systems, and be able to react to requests from and changes on other systems. As a practical example, consider a set of data (re)sources in the Web of travel agencies, airline companies, train companies, etc. It should be possible to query the resources about timetables, availability of tickets, etc. But in such an evolving Semantic Web, it should also be possible for a train company to report on late trains, and travel agencies (and also individual clients) be able to detect such an event and react upon it, by rescheduling travel plans, notifying clients that in turn could have to cancel hotel reservations and book other hotels, or try alternatives to the late trains, etc.

ECA Rules

Some reactive languages have been proposed that allow for updating Web sources as the above ones, and are also capable of dealing-with/reacting-to some forms of events, evaluate conditions, and upon that act by updating data [12, 11, 4, 61] (see Section 1.2). The common aspect of all of these languages is the use of declarative *Event-Condition-Action* (ECA) rules for specifying reactivity and evolution. Such kind of rules (also known as triggers, active rules, or reactive rules), that have been widely used in other fields (e.g. active databases [62, 70]) have the general form:

on event if condition do action

They are intuitively easy to understand, and provide a well-understood semantics: when an event (atomic or composite) occurs, evaluate a condition, and if the condition (depending on the event, and possibly requiring further data) is satisfied then execute an action (or a sequence of actions, a program, a transaction, or even start a process).

In fact, we fully agree with the arguments exposed in the field of active databases for adopting ECA rules for dealing with evolution and reactivity in the Web (declarativity, modularity, maintainability, etc). Still, the existing languages fall short in various aspects, when aiming at the general view of an evolving Web as described above. In these languages, the events and actions are restricted to updates on the underlying data level; they do not provide for more composite events and actions. In a Semantic Web environment, actions are more than just simple updates to Web data (be it XML or RDF data), but application-level *actions*. As said above, besides that, actions can be notifications to other resources, or update requests of other resources, and they can be composed from simpler actions (like: do this, and then do that).

Moreover, events may in general be more than simple atomic events in Web data, as in the above languages. First, there are atomic events other than physical changes in Web data: events may be received messages, or even “happenings” in the global Web, which may require complex event detection mechanisms (e.g. (once) any train to Munich is delayed ...). Moreover, as in active databases [29, 74], there may be *composite* events. For example, we may want a rule to be triggered when there is a flight cancellation and then the notification of a new reservation whose price is much higher than the previous (e.g. to complain to

the airline company). In our view, a general language for reactivity in the Web should cater for such richer actions and events.

Such a general ECA language with richer actions and events, adapted to Web data, is not yet enough for fully materialising our view of an evolving Semantic Web. In fact, a main goal of the *Semantic Web* since its inception is to provide means for a unified view on the Web, which obviously includes to deal with the heterogeneity of data formats and languages. In this scenario, XML (as a format for exchanging data), RDF (as an abstract data model for states, sometimes stored natively, sometimes mapped to XML or a relational storage), OWL (as an additional framework for theory-based knowledge representation) provide the natural underlying concepts. The Semantic Web does not possess any central structure, neither topologically nor thematically, but is based on peer-to-peer communication between autonomous, and autonomously developing and evolving nodes. This *evolution* and *behaviour* depend on the cooperation of nodes. In the same way as the main driving force for the Semantic Web idea was the heterogeneity of the underlying data, the heterogeneity of concepts for expressing behaviour requires an appropriate handling on the semantic level. When considering evolution, the concepts and languages for describing and implementing behaviour will surely be diverse, albeit due to different needs, and it is unlikely that there will be a unique language for this throughout the Web. Since the contributing nodes are prospectively based on different data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that their *concepts* are independent from the actual data models and languages, and allow for an integrated handling of these.

Our view is that a general framework for evolution and reactivity in the Semantic Web should be based on a general ECA language that allows for the usage of different event languages, condition languages, and action languages. Each of these different (sub)languages should adhere to some minimal requirements (e.g. dealing with variables), but it should be as free as possible.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. For that, the ECA rules themselves must be represented as data in the Semantic Web, based on an ontology of ECA rules and (sub)ontologies for events, conditions and actions, with rules specified in RDF. The ontology does not only cover the rules themselves but, for handling language heterogeneity, the rule components have to be related to actual languages, which in turn can be associated with actual processors. Moreover, for exchange of rules and parts of them, an XML Markup of ECA Rules, that is preferably closely related to the ontology, is needed.

In this chapter, after a brief overview of the state of the art, we present a general framework for evolution and reactivity in the Semantic Web, which caters for the just exposed requirements. The framework also provides a comprehensive set of concrete languages. We continue the chapter with a description of a concrete homogeneous language for reactivity and evolution, XChange.

Both the general framework and the XChange language have been developed (and implemented) in the REVERSE project. This work opened several possi-

bilities of future applications and research areas, that are sketched in the last section of this chapter.

1.2 Starting point and related work

As already mentioned before, the issue of reactivity, and even that of reactivity on the Web, had already been studied before the beginning of this work.

Reactivity in Databases. Reactivity has been extensively studied in the area of databases, e.g., in [62, 70]. In these, notions of composition of events, as advocated above, have been proposed based on *event algebras* with their concise theory and semantics and well-understood detection mechanisms. A prominent representative of such approaches is e.g. the SNOOP algebra of the Sentinel system [28] for transactional rules and rule-driven business workflows. Also more recent approaches like RuleCore [9] use similar concepts more or less explicitly. Our work on composite events in the Semantic Web has its roots on this previous work.

Event Algebra expressions are formed by nesting operators and basic expressions that specify which atomic events are relevant. For this, every event algebra specifies a set of *operators*, e.g., “A and B”, “A or B”, “A and then B”, “not C between A and B”. From a declarative point of view, such an event algebra expression can be true or false over a given sequence of events. From the procedural point of view, a composite event is *detected* at the timepoint where it becomes true wrt. the sequence of events occurred up to that point. Event algebra terms are usually not evaluated like queries against the history (although their semantics is defined like that), but are detected *incrementally* against the stream of incoming events.

Process Definition Languages. Also for the specification of composite actions, work already existed on process algebras and other process definition languages. Well-known process algebras are *CCS (Calculus of Communicating Systems)* [59] or *CSP (Communicating Sequential Processes)* [46]; another prominent recent process specification language is *BPEL (Business Process Execution Language)* [60]. In these approaches, e.g., the following concepts can be specified:

- sequences of actions to be executed (as in simple ECA rules);
- processes that include “receiving” actions, like the corresponding actions a and \bar{a} action in CCS that are used for modeling communication: \bar{a} can only be executed together with a (sending) action a in another process. The semantics of \bar{a} is thus similar to the event part of ECA rules *on a if condition do action* where the occurrence of a “wakes the rule up” and starts execution of the subsequent condition and action;
- guarded (i.e., conditional) execution alternatives;
- families of *communicating, concurrent processes*, and
- starting an iteration or even infinite processes.

Reasoning about Actions. Formalisms for representing and reasoning about actions and effects of actions have also long been studied in Artificial Intelligence. Action languages have been defined to account for just that [49, 56, 5, 40, 41, 42, 43, 44]. Central to this approach of formalizing actions is the concept of a transition system: a transition system is simply a labelled graph where the nodes are states and the arcs are labelled with actions or sets of actions. Usually the states are first-order structures, where the predicates are divided into static and dynamic ones, the latter called *fluents* (cf. [66]). Action programs are sets of sentences that define one such graph by specifying which dynamic predicates change in the environment after the execution of an action. Usual problems here are to predict the consequences of the execution of a sequence of (sets of) actions, or to determine a set of actions implying a desired conclusion in the future (planning).

Most of the above action languages are equipped with appropriate action query languages, that allow for querying such a transition system, going beyond the simple queries of knowing what is true after a given sequence of actions has been executed (allowing e.g. to query about which sets of actions lead to a state where some goal is true, which involves planning).

Web Update Languages. The above work sets up the foundation on which the definition of reactivity and evolution in the Semantic Web has been inspired. Furthermore, *reasoning* about such behaviour has its own specificity that requires a specific solution *after* the mechanisms have been defined. To start, there is the issue of how to update Web data, something that is much more concrete than e.g. the update of states in action languages. For this, as a starting point we could rely on a number of proposals such as XUpdate [72], the XQuery update extension of [69], XML-RL [51], XPathLog [53], and RUL [52].

XUpdate [72] makes use of XPath expressions for selecting nodes to be processed afterwards, in a way similar to XSLT. The XSLT-style syntax of the language makes the programming, and the understanding of complex update programs, very hard.

A proposal to extend XQuery with update capabilities was presented in [69]. In it XQuery is extended with a `FOR ... LET ... WHERE ... UPDATE ...` structure. The new `UPDATE` part contains specifications of update operations (i.e. delete, insert, rename, replace) that are to be executed in sequence. For ordered XML documents, two insertion operations are considered: insertion before a child element, and insertion after a child element. Using a nested `FOR ... WHERE` clause in the `UPDATE` part, one might specify an iterative execution of updates for nodes selected using an XPath expression. Moreover, by nesting update operations, updates can be expressed at multiple levels within a XML structure.

The XML-RL Language [51] incorporates features of object-oriented databases and logic programming. The XML-RL Update Language extends XML-RL with update capabilities. Five kinds of update operations are supported by the XML-RL Update Language, viz. `insert before`, `insert after`, `insert into`, `delete`, and `replace with`. Using the built-in position function, new elements can be inserted at the specified position in the XML document (e.g. insert first,

insert second). Also, complex updates at multiple levels in the document structure can be easily expressed.

XPathLog [53] is a rule-based logic-programming style language for querying, manipulating and integrating XML data. XPathLog can be seen as the migration from F-Logic [48], as a logic-programming style language, for semistructured data to XML. It uses XPath as the underlying selection mechanism and extends it with the Datalog-style variable concept. XPathLog uses rules to specify the manipulation and integration of data from XML resources. As usual for logic-programming style languages, the query and construction parts are strictly separated: XPath expressions in the rule body, extended with variables bindings, serve for selecting nodes of XML documents; the rule head specifies the desired update operations intensionally by another XPath expression with variables, using the bindings gathered in the rule body. As a logic-programming style language, XPathLog updates are insertions.

Reactive Web Languages. Also some reactive languages have been proposed, that do not only allow for updating Web data as the above ones, but are also capable of dealing-with/reacting-to some forms of events, evaluate conditions, and act by updating data. These are e.g. Active XQuery [11], the XML active rules of [12], the Event-Condition-Action (ECA) language for XML defined in [4], and the ECA reactive language RDFTL [61] for RDF data.

Active XQuery [11] expands XQuery with a trigger definition and the execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases (and using the same syntax for CREATE TRIGGER). It adapts the SQL3 notions of BEFORE vs. AFTER triggers and, moreover, the ROW vs. STATEMENT granularity levels to the hierarchical nature of XML data. The core issue here is to extend the notions from “flat” relational tuples to hierarchical XML data.

Another approach to ECA rules reacting on updates of standard XML documents, in the style of SQL3 triggers, is the one of [4]. It defines ECA rules, of the usual form `on ... if ... do`, where events can be of the form `INSERT e` or `DELETE e`, where `e` is an XPath expression that evaluates to a set of nodes; the nodes where the event occurs are bound to a system-defined variable `$delta` where they are available for use in condition and action parts. An extension for a replace operation is sketched. The condition part consists of a boolean combination of XPath expressions. The action part consists of a sequence of actions, where each action represents an insertion or a deletion in XML. For insertion operations, one can specify the position where the new elements are to be inserted using the `BELOW`, `BEFORE`, and `AFTER` constructors. This work has been extended to RDF data (serialised as XML data) in [61].

These approaches are “local”, in that, as in SQL3, work on a local database, are defined inside the database by the database owner, and only consider local events and actions. On the contrary, the XML active rules of [12] establishes an infrastructure for user-defined ECA rules on XML data, where rules to be applied to one given repository can be defined by arbitrary users (using a predefined XML ECA rule markup), and can be submitted to that repository where

they are then executed. The definition of events and conditions is up to the user (in terms of changes and a query to an XML instance). The actions are restricted to sending messages. This approach further implements a subscription system that enables users to be notified upon changes on a specified XML document somewhere on the Web. For this, the approach extends the server where the document is located by a rule processing engine. Users that are interested in being notified upon changes in the document submit suitable rules to this engine that manages all rules corresponding to documents on this server. Thus, evaluation of events and rules is local to the server, and notifications are “pushed” to the remote users. Note that the actions of the rules do not modify the information, but simply send a message.

None of these languages considers composite events in general. There is some preliminary work on composite events in the Web [8], but it only considers composition of events of modification of XML-data in a single document.

Besides being mostly limited to updates, and reaction on updates, on XML (or RDF) data, and with mostly no support for composite events or actions, none of these proposals tackles the issues of heterogeneity of behaviour and languages in the Semantic Web, of dealing with composite events in the Web, and dealing with composite actions, required for materialising our initial vision of an active Semantic Web, where reactivity, evolution and propagation of changes play a central role. Having all of these aspects combined in a single framework is the goal of the work presented in this chapter.

1.3 Conceptualization of ECA Rules and their Components: A General Framework for ECA Rules

The idea of a *General Framework for ECA Rules* aims at covering (i) active concepts w.r.t. the domain ontologies and (ii) heterogeneity of domain-independent conceptualization of activity in a comprehensive way [55].

Active Concepts of Domain Ontologies. The general framework assumes actions and events to be first-class citizens of the domain ontologies. While static notions like classes, properties, and their instances are represented in the state of one or more nodes, events and actions are present as volatile entities. Events and actions are represented by XML (including RDF/XML) fragments that are exchanged (e.g., by HTTP) between nodes. For instance,

```
<travel:CanceledFlight travel:code="LH1234" >
  <travel:reason>bad weather</travel:reason>
</travel:CanceledFlight>
```

is the representation of an event (raised e.g. by an airport).

Every domain ontology – e.g. for banking or traveling – defines *static notions* (classes, relationships) and *dynamic notions*, i.e., the types of possible events and

actions as classes of the respective domain ontologies as shown in UML notation in Fig. 1.

Next, we identify the types of languages used in the rules to deal with state, events, and actions.

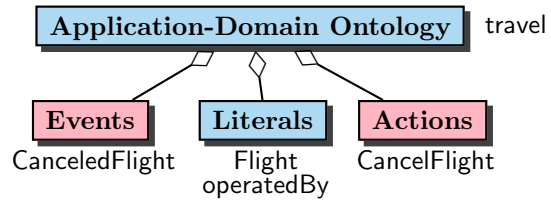


Fig. 1. Components of Domain Ontologies

Domain-Independent Conceptualization of Activity. This aspect deals with the modeling and specification of active rules and rule components, i.e., the description of relevant events, including composite events, conditions, including queries against the state of domain nodes, and actions, including the specification of composite actions up to (even infinite) processes. While the atomic constituents are provided by the domain ontology, any kind of *composition* requires domain-independent notions. As mentioned above, multiple formalisms and languages for composite events, queries, and actions have been proposed. Each of them can be seen as an *ontology* of composite events, processes, etc.

The general framework covers this matter by following a modular approach for rule markup that considers the markup and ontology on the generic rule level separately from the markup and ontologies of the components.

Two different variants of the general idea have been implemented:

MARS – Modular Active Rules for the Semantic Web is an open architecture that allows for combining nearly arbitrary existing languages and services.

For this, MARS includes a meta-level ontology of languages and services in general.

r^3 – Resourceful Reactive Rules follows an integrated design that is based on a toolbox for defining and implementing heterogeneous languages in a homogeneous programming environment.

In the following, we first present the common ideas underlying the general framework, and then point out the different design decisions in MARS and r^3 . Services from both approaches can also interoperate.

1.3.1 The Rule Level

The core of the general framework is a model and architecture for ECA rules that use *heterogeneous* event, query, and action languages. The condition component is divided into queries to obtain additional information (from potentially different sources; the queries can be expressed in different languages) and a test component (that consists only of a boolean combination of generic comparison operators e.g., from XPath):

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

The approach is parametric regarding the component languages. Users write their rules by using component languages of their choice. While the semantics of the ECA rules provides the global semantics, the components are handled by

specific services that implement the respective languages. Fig. 1.2 (from [54]) illustrates the structure of the rules and the corresponding types of languages.

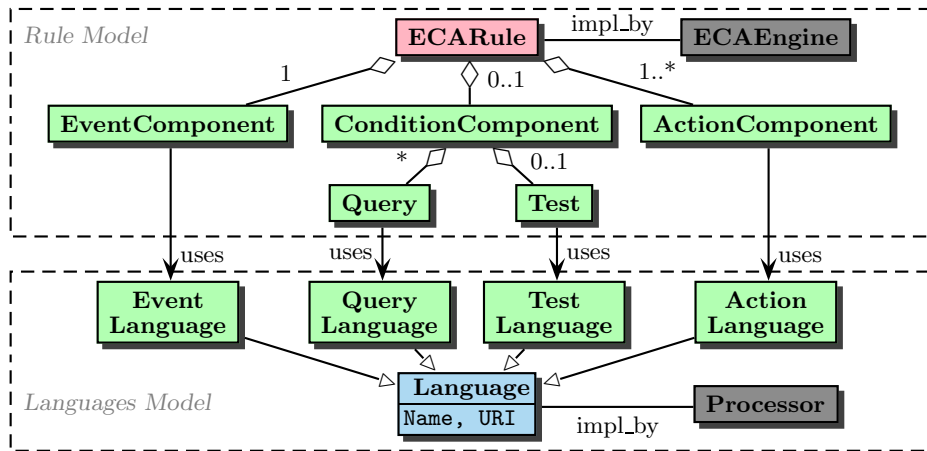


Fig. 1.2. ECA Rule Components and Corresponding Languages (from [54])

Markup. The markup of the rule level, i.e., the ECA-ML markup language, mirrors this structure as shown in Figure 1.3. It allows to embed the components as nested subexpressions in their own markup (using their own namespaces). The conceptual border between the ECA rule level and the particular concepts of the E, C and A components is manifested in language borders between the ECA level language and the languages of the nested components. The language borders are used at execution time to organize the cooperation between appropriate processors. The components are specified as nested subexpressions of the form

```

<eca:Component-Type xmlns:lang="embedded-lang-ns">
  embedded fragment in the embedded language's markup and namespace
</eca:Component-Type>
  
```

in arbitrary formalisms or languages.

As we shall see, analogous conceptual borders are found between the level of *composite expressions* and their *atomic subexpressions*.

Communication and Data Flow Between Components. The data flow throughout the rules, and between the ECA engine and the event, query, test, and action components is provided by *logical variables* in the style of deductive rules, or of production rules. The state of a rule evaluation, and the information sent and returned by service calls is always a *set of tuples of variable bindings*. Thus, all paradigms of query languages, following a functional style (such as XPath/X-Query), a logic style (such as Datalog or SPARQL [64]), or both (F-Logic [48]) can be used. The semantics of the event part (that is actually a “query” against an event stream that is evaluated incrementally) is –from that point of view– very similar to that of queries, and the action part takes variable bindings as

```

<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#" >
  <eca:Event>
    <ns1:el1>nested expression in event specification language markup </ns1:el1>
  </eca:Event>
  <eca:Query" >
    <ns2:el2>nested expression in query language markup </ns2:el2>
  </eca:Query>
  .
  <eca:Query > ... </eca:Query>
  <eca:Test>
    <ns3:el3>test expression over obtained information </ns3:el3>
  </eca:Test>
  <eca:Action>
    <ns4:el4>nested expression in action language markup </ns4:el4>
  </eca:Action>
</eca:Rule>

```

Fig. 1.3. ECA-ML Markup Pattern

input. Given this semantics, the ECA rule combines the evaluation of the components in the style of production rules (evaluated in forward-chaining mode “if body then head”, cf. Figure 1.4):

$$action(X_1, \dots, X_n, \dots, X_k) \leftarrow event(X_1, \dots, X_n), query(X_1, \dots, X_n, \dots, X_k), test(X_1, \dots, X_n, \dots, X_k) .$$

The evaluation of the event component (i.e., the successful detection of a, possibly composite, event) binds variables to values that are then extended in the query component, possibly constrained in the test component, and propagated to the action component.

For the actual data exchange, an XML format has been defined. Alternatively, for local services, internal data structures can be exchanged as references, and also a shared database storage is provided.

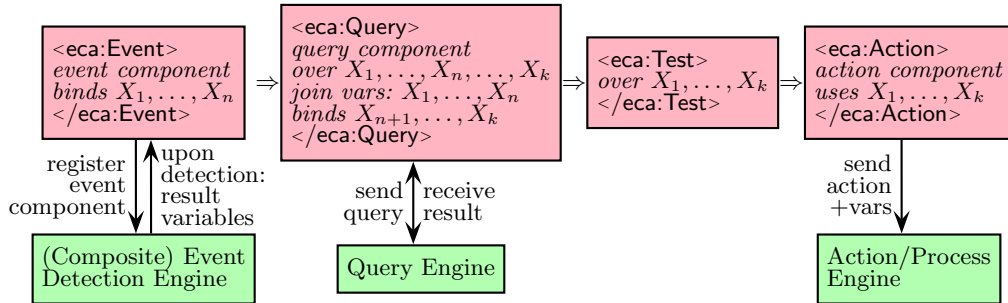


Fig. 1.4. Use of Variables in an ECA Rule

1.3.2 The Event Component

For the event component, two levels of specifications are combined (cf. Figure 1.5): The specification of the (algebraic) structure of the composite event is

given as a *temporal combination* of atomic events, and the specification of the *contributing* atomic events as the leaf expressions is given by small “queries” against the actual events that check if an event matches the specification.

Atomic Event Specifications. As shown at the beginning of this section, actual events are volatile items, considered to be represented as XML fragments. Atomic event specifications (AESs) are used in the rules’ event components for specifying which atomic events are relevant; they form the leaves of the event component tree. Their specification needs to consider the type and contents of atomic events, e.g. reacting to an event “if a flight is canceled due to bad weather conditions then ...”. In case of detecting the composite event “if a flight is first delayed and later canceled then ...”, the flight number must be *extracted* from the event to use it in a *join condition* between *different* atomic events. The following fragment shows an atomic event specification in an XML-QL-style [32] matching formalism, that extracts the flight number and the reason from such an event:

```
<xmq:AtomicEvent
  xmlns:xmq="http://www.semwebtech.org/languages/2006/xmlql#" >
  <travel:CanceledFlight xmlns:travel="..." travel:flight="{flightno}" >
    <travel:reason>{$reason}</travel:reason>
  </travel:CanceledFlight>
</xmq:AtomicEvent>
```

In an atomic event specification, there are always *two* languages involved as shown in Figure 1.5: (i) a domain language (associated with the namespace of the event; above: travel), and (ii) an atomic event description/matching/query language (above: xmq) for *describing* what events should actually be matched.

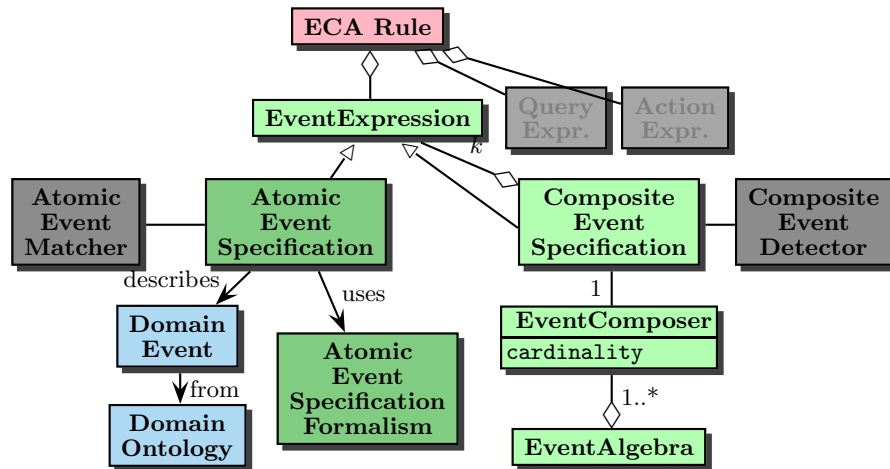


Fig. 1.5. Event Component: Languages (from [7])

Composite Event Specifications. Given an event algebra, the markup of event algebra expressions is straightforward, forming a tree term structure over atomic

event specifications. As a sample composite event language, a variant of the SNOOP event algebra [29] extended with relational data flow has been developed [7].

1.3.3 The Condition Component

The condition component consists of one or more queries to obtain additional information from domain nodes, and a test that is evaluated over the obtained variable bindings.

As query languages, *opaque* components play an important role: most domain nodes are assumed to provide an existing query language according to their data model, e.g., SQL, XPath/XQuery, or SPARQL. While all these languages support variables, they are not in any XML markup¹, but queries are given as *strings* that have to be parsed at the respective nodes. For the ECA rules, these strings are opaque. Opaque embedded fragments are of the form

```
<eca:Opaque eca:language="lang-id" eca:uri="uri" >
  query string
</eca:Opaque>
```

which indicates the language and the URI where the query has to be sent for being answered. Opaque code fragments can also be used in the action part.

Additionally, a query language for RDF and OWL data that has an RDF syntax (whose XML markup is its RDF/XML serialization), called OWLQ, has been developed in the MARS project.

1.3.4 The Action Component

The action component specifies the actual reaction to be taken. This again can be an atomic action, or a composite action, often called *process*. For its specification, process languages or process algebras can be used. Given a process language, the markup on the process level is again straightforward, forming a tree expression structure (note that BPEL [60] is originally defined as an XML language).

Atomic actions are those of the application domains, again represented as XML fragments, belonging to some domain namespace. Such atomic actions are then sent to the appropriate nodes to be executed. The specification of an action to be executed thus consists of the specification to generate an XML fragment which is then submitted to the corresponding domain nodes, or to a domain broker [6] that will in turn submit it to appropriate domain nodes (using the namespace identification). For that, also multiple languages exist.

Conditions, Queries and Events inside Process Specifications. The specification of a process, which e.g. includes branching or waiting for a response, can also require the specification of queries to be executed, and of events to be waited for. For that, we allow event specifications, queries and conditions as regular, executable components of a process:

¹ With exceptions, such as XQueryX as an XML markup for XPath/Query

- “executing” a query means to evaluate the query, to extend the variable bindings, and to continue.
- “executing” a condition means to evaluate it, and to continue for all tuples of variable bindings where the condition evaluates to “true”. For instance, for a *conditional alternative* process $((c : a_1) + (-c : a_2))$, all variable bindings that satisfy c will be continued in the first branch with action a_1 , and the others are continued with the second branch.
- “executing” an event specification means to wait for an occurrence of the respective event.

Figure 1.6 shows the relationship between the process algebra language and the contributions of the event and test component languages, and those of the domain languages. As a sample process language, an enriched variant of CCS [59] has been defined [7, 47] that works on relational states (i.e., a set of tuples of variable bindings) that are manipulated by the atomic actions.

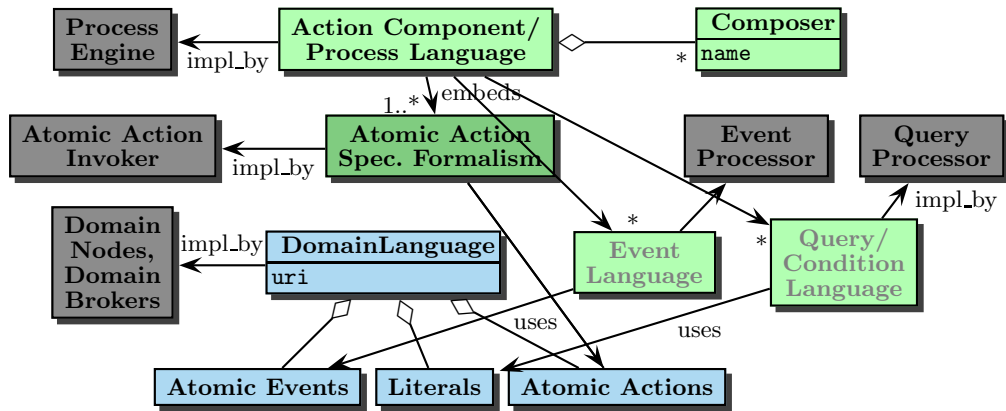


Fig. 1.6. Structure of the Action Component as an Algebraic Language (from [7])

1.3.5 Languages and Language Borders

In the above design, rules and (sub)expressions are represented by XML trees. Nested fragments correspond to subtrees in different languages, corresponding to XML namespaces, as illustrated in Figure 1.7: the rule reacts on a composite event (specified in the extended SNOOP [29, 7] event algebra as a sequence) “if a flight is first delayed and later cancelled”, and binds the flight number and the reason of the cancellation. Two expressions in an XML-QL-style matching formalism contribute the atomic event specifications. Note the occurrence of the travel domain namespace inside the atomic event specification.

Processing of an XML fragment in a given language, or more abstractly, executing some task for a fragment, is organized by using the namespace URI of the fragment’s outermost element. Every processor (e.g., the one responsible for the crosshatched event part in the snoopy namespace) controls the processing

of “his” level (the SNOOP event algebra), and whenever an embedded fragment (e.g., an atomic event specification) has to be processed, the appropriate processor (here, for XML-QL) is invoked.

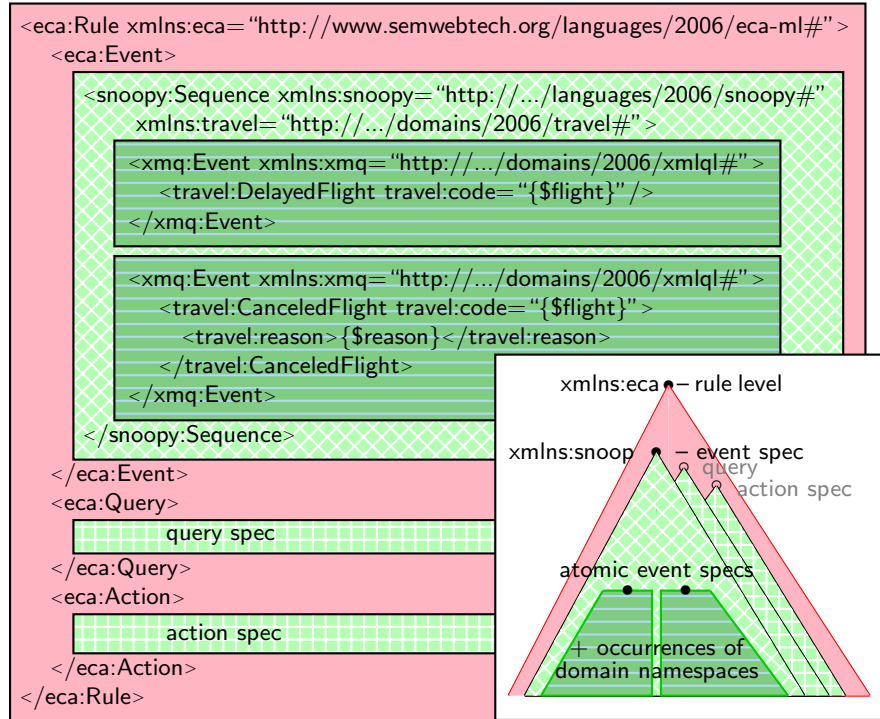


Fig. 1.7. Nesting of Language Subtrees

The aim of the General Framework idea is to allow to embed *arbitrary* languages of appropriate types by only minimal restrictions on the languages. The cornerstones of the framework w.r.t. this issue are the following:

- the approach does only minimally constrain the component languages: the information flow between the ECA engine and the event, query, test, and action components is provided by (i) XML language fragments, and (ii) current *variable bindings* (cf. Fig. 1.4),
- a comprehensive ontology of language types, service types and tasks (as described below),
- an open, service-oriented architecture,
- a *Language and Service Registry (LSR)* that holds information about actual services and how to do the actual communication with them.

The XML and RDF concept of *namespaces* provides a powerful and built-in mechanism to identify the language of an XML fragment: namespaces are the languages’ URIs. The concrete languages are related to actual services, and the

namespace information of a fragment to be processed is used to select and address an appropriate processor.

With that, all necessary information what to do with an embedded fragment of a “foreign language” is contained in (i) the language fragment (via the namespace of its root element), (ii) the local knowledge of the currently processing service (i.e., what it expects the fragment to be, and what it wants do to with it), and (iii) the LSR information.

1.3.6 Languages Types, Service Types, and Tasks

For every kind of language there is a specific type of service that provides a specific set of tasks – these are independent from the concrete language; only the actual implementation by a service is language-dependent.

The *Languages and Services Ontology* is shown in Figure 1.8; sample instances are denoted by dashed boxes. The ontology contains two levels: the level of *language types* and corresponding *service types*, and the concrete *languages* and *services*.

There are the following *language types*, with the corresponding *service types*:

Rule Languages, e.g. the ECA rule language, are handled by rule engines. There, e.g., rules can be registered.

Event Specification Languages (specifications of composite or atomic events): composite event specifications are processed by *Composite Event Detection Engines (CEDs)*; atomic event specifications are processed by *Atomic Event Matchers (AEMs)*. In both cases, event specifications can be registered at such services. Upon occurrence/detection of the event, the registrant will be notified (asynchronously).

Query Languages are handled by *query engines*. Queries can be sent there, and they are answered (synchronously or asynchronously).

Test Languages: tests (i.e., boolean queries) are also handled by query engines, or locally (as they involve rather simple comparisons). Tests can be submitted, and they are answered (synchronously or asynchronously).

Action Languages: Composite and atomic actions are processed by action services. Action specifications can be submitted there for execution (either processes, or atomic actions).

Domain Languages: Every domain defines a language that consists of the names of actions (understood to be executed), classes/properties/predicates (depending on the respective data model), and events (emitted by the domain services) as shown in Figure 1. Domain services support these names and carry out the real “businesses”, e.g., airlines (in the travel domain) or universities. They are able to answer queries, to execute (atomic) actions of the domain, and they emit (atomic) events of the domain. *Domain Brokers* implement a portal functionality for a given domain.

For every kind of language there is intuitively a typical set of tasks (e.g., given a query language *QL*, one expects that a service that implements *QL* provides

the task “answer-query”). In the *Languages and Services Ontology*, the tasks are not associated with the language, but with the corresponding service type (in programming languages terminology, a *service type* is an interface that is implemented by the concrete services; thus the provided tasks can be seen as (part of) its signature).

For a concrete language, e.g., SNOOP, there are one or more concrete services (of the appropriate service type) that implement it (here: snoopy). Each such service has a URI, and has to provide the characteristic tasks of the service type (in programming languages terminology: implement the signature of the interface). For each provided task, there is a *task description* that contains the information how to establish communication (described in detail at the end of this section).

The relationships on the meta level are called *meta-implemented-by* and *meta-provides-task*, while on the concrete level, they are called *implemented-by* and *provides-task*. The reason for not just overloading names is that the RDFS description then allows to distinguish domains and ranges.

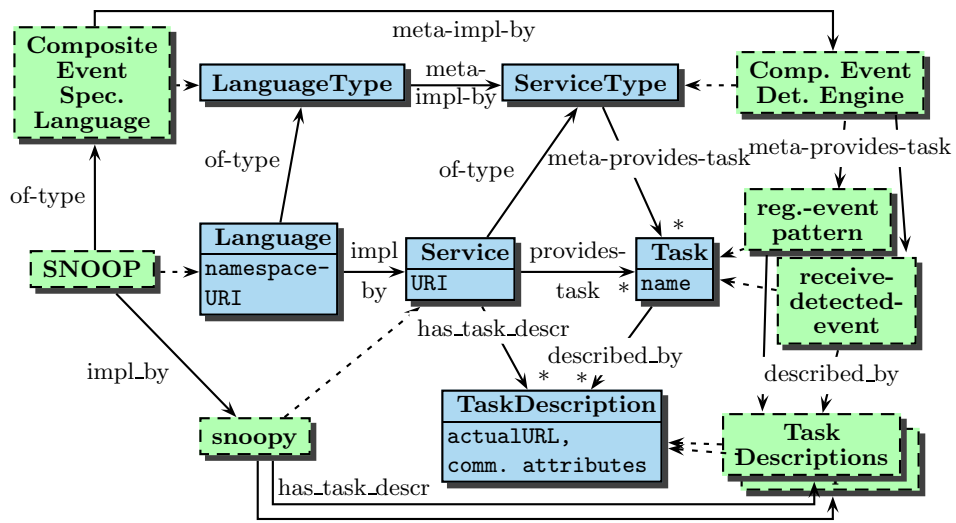
Figure 1.9 shows the most important tasks for each service type; additionally, the actual communication flow is indicated: e.g., rule engines **provide** the task “register-rule”, which in turn **calls** the task “register-event-pattern” that is **provided** by event detectors. When the task “register-event-pattern” of a CED is called, it will in turn **call** the task “register-event-pattern” for the embedded (atomic) subevents at some AEMs (for the respective AESLs). Domain nodes emit events that end up at the task “receive-event” that is **provided** by AEMs. If such an event matches a registered pattern, the AEM will **call** the task “receive-detected-event” that is **provided** by the registrant (which is a CED or a rule engine). A more concrete example using the sample rule from Figure 1.7 will be given in Section 1.3.7.

Information about Concrete Services. The concrete information about available services for the concrete languages is managed by the *Language and Service Registry (LSR)*. For every such service, the LSR contains the URI, and for each provided task, there is a *task description* that contains information for establishing communication (cf. Figure 1.10 for a sample in RDF/XML markup):

- the actual URL (as a service supports multiple tasks, each of them may have an own URL, which is not necessarily related to the service’s URI),
- whether it supports to submit a set of tuples of variables, or only a single tuple at each time,
- information about the required message format:
 - send reply-to address and subject in the message header or in the body,
 - whether it requires a certain wrapper element for the message body,
- whether it will answer synchronously or asynchronously.

All MARS ontologies and an LSR snapshot in XML/RDF syntax can be found at <http://www.dbis.informatik.uni-goettingen.de/MARS/#mars-ontologies>.

For processing a component or subexpression, a *language processor* for the indicated specification language is determined by asking an LSR for a processor



Notation: dashed boxed denote sample instances; dashed lines stand for MOF's <<instance-of>> and denote instanceship.

Fig. 1.8. Ontology of Languages and Services

Language Type:	Rule Languages	Composite Event Languages	Atomic Languages	Query/Test Languages	Process Languages	AtomicAct. Languages	Domain Languages
Service Type:	Rule Engines	Composite Event Detectors	Atomic Query/Test Services	Composite Action Services	Atomic Services	Domain Brokers	Domain Nodes
register-rule	P						
reg.-event-pattern	C	P/C	P/C		C		P
rec-detected-ev. (SendEvent)	P	C/P	C		P		C/P
evaluate-query (answer-query)	C			P/C	C		P/C
rec-query-answer	P			C/P	P		C/P
evaluate-test	C			P/C	C		
rec-test-answer	P			C/P	P		
execute-action	C				P/C	P/C	P/C

P: Provides task ; C: Calls task – asynchronous answers will be sent to another task
 Arrows from C to P of the same service type represent communication between different services of the same type (e.g., nesting of different event algebras).
 The rightmost dashed arrows represents the domain-specific behavior of domain nodes: from executing actions, occurrences of events may be raised.

Fig. 1.9. Services and Tasks

```

<mars:EventAlgebra rdf:about="http://.../languages/2006/snoop#" >
  <mars:is-implemented-by>
    <mars:CompositeEventDetectionEngine xml:base="http://.../services/2007/snoopy/"
      rdf:about="http://www.semwebtech.org/services/2007/snoopy" >
      <has-task-description> <TaskDescription>
        <describes-task rdf:resource="&mars;/ced#register-event-pattern" />
        <provided-at rdf:resource="register" /> <input>element register</input>
        <Reply-To>body</Reply-To> <Subject>body</Subject>
        <variables>*</variables>
      </TaskDescription> </has-task-description>
      :
    </mars:CompositeEventDetectionEngine>
  </mars:is-implemented-by>
</mars:EventAlgebra>

```

Fig. 1.10. MARS LSR: LSR entry with Service Description Fragment for SNOOP

for the *embedded-lang-ns* namespace and the task is submitted to that node according to the information given in the respective task description. The actual process of determining an appropriate service and organizing the communication is operationally performed by a *Generic Request Handler (GRH)*, that is used by all sample services. Details about the actual handling are described in [37].

In the current prototype, the LSR is implemented by a central RDF/XML file. In a fully operational MARS environment, the LSR would be realized as one or more LSR services where language services can register and deregister, and that are connected e.g. in a peer-to-peer way. By that, e.g., different LSRs can list their “friend” services, and only fall back to remote services if no local ones are available.

1.3.7 Architecture and Processing: Cooperation between Resources

Imposed by the structure of the rule and the type of languages, each service plays a certain role when processing the parts it is responsible for. The basic pattern, according to the ECA structure is always the same, as illustrated in Figures 1.11 (global interaction) and 1.12 (more detailed view of the services and tasks that are involved in composite event detection including the prior registration of event specifications).

Consider again the example rule from Figure 1.7:

A client registers the rule (which deals with the *travel* domain) at the ECA engine (Step 1.1). Event processing is done in cooperation of an ECA engine, one or more *Composite Event Detection Engines (CEDs)* that implement the event algebras (in the example: SNOOP), and one or more *Atomic Event Matchers (AEMs)* that implement the *Atomic Event Specification Languages (AESLs)* (in the example: XML-QL). For this, the ECA engine submits the event component to the appropriate CED service (1.2), here, a SNOOP service. The SNOOP service inspects the namespaces of the embedded atomic event specifications and registers the atomic event specifications (for *travel:DelayedFlight* and *travel:CanceledFlight*)

at the XML-QL AEM service (1.3). The AEM inspects the namespaces of the used domains, where in this case the `travel` ontology is relevant. It contacts a travel domain broker (1.4) to be informed about the relevant events (i.e., `DelayedFlight` and `CanceledFlight`).

The domain broker forwards relevant atomic events to the AEM (2.2; e.g., happening at Lufthansa (e.g., 2.1a: `DelayedFlight(LH123)`, 2.1c: `CanceledFlight(LH123)`) and Air France (2.1b: `DelayedFlight(AF456)`)). The AEM matches them against the specifications and in case of a success reports the matched events and the extracted variable bindings to the SNOOP service (3). Only after detection of the registered composite event (after events 2.1a and 2.1c), SNOOP submits the result to the ECA engine (4).

This means the actual “firing” of the rule which then evaluates additional queries and tests (assumed to be empty here). Then, the action component is executed. Assume an action component (which is not given explicitly in Figure 1.7) that uses CCS and contains an atomic action concerning the corresponding airline node and an atomic action that sends a mail (using an SMTP action). The ECA engine inspects the used language namespace (`ccs:`) and forwards it to a CCS service (5.1). The CCS node forwards the action to the Lufthansa node via the domain broker (5.2a,b) and sends a mail (5.3a,b) via an SMTP service.

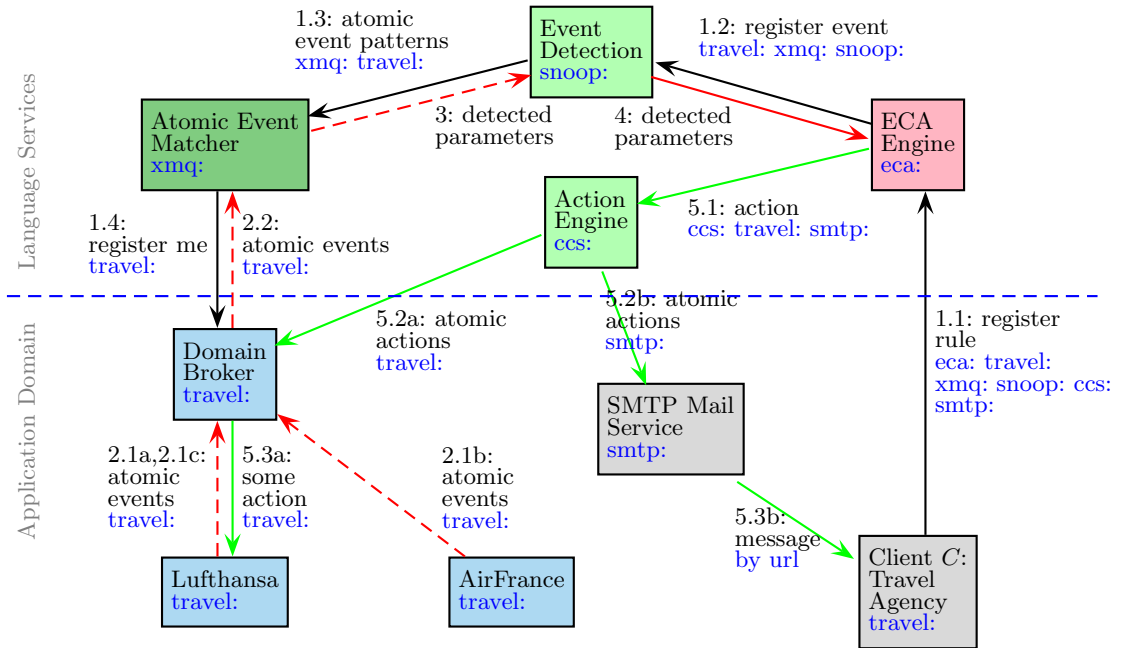


Fig. 1.11. Communication: Event Processing

Embedding of Domain Languages. Domain languages and services are completely compatible with this approach (cf. Figure 1.9). For domain nodes, the tasks are

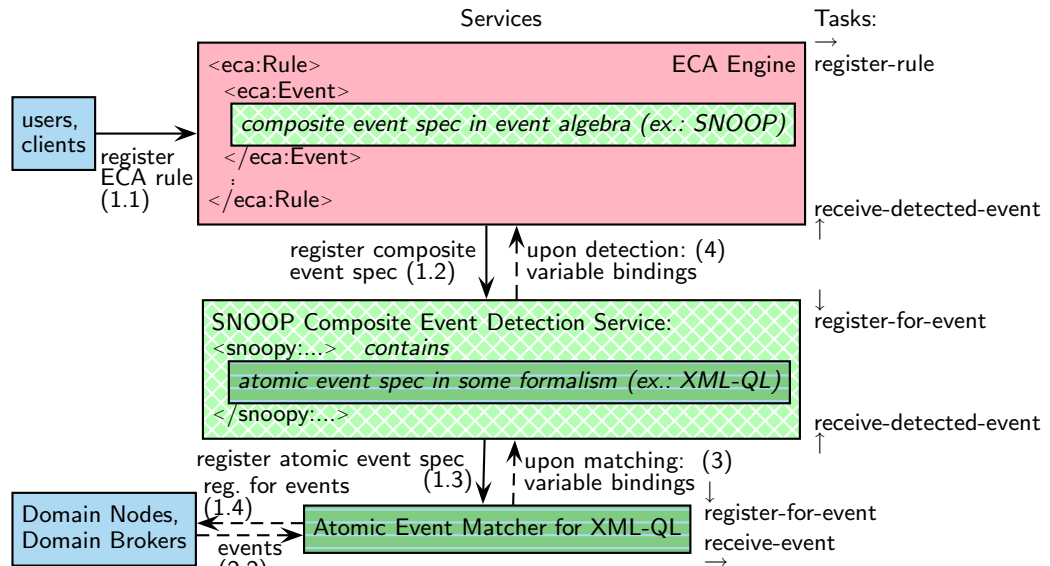


Fig. 1.12. Processing Event Components and Events

register-for-event and execute-action. Domain brokers provide a portal functionality between language services and domain services.

1.3.8 The RDF Level: Language Elements and their Instances as Resources

Rules on the semantic level, i.e., RDF or OWL, lift ECA functionality w.r.t. two (independent) aspects: first, the events, conditions and actions refer to the domain ontology level as described above. On an even higher level, the above rule ontology and event, condition, and action subontologies regard rules themselves as objects of the Semantic Web. Together with the languages and their processors, this leads directly to a resource-based approach: every rule, rule component, event, subevent etc. becomes a resource, which is related to a language which in turn is related to other resources.

Describing rules in RDF provides an important base to be able to reason about rules. This will support several things:

- validation and support for execution,
- actual reasoning about the behaviour of a node, including correctness issues,
- expressing rules in abstract terms instead of w.r.t. concrete languages. The services can then e.g. choose which concrete languages support the expressiveness required by the rule's components.

For the RDF/OWL level, we assume that not only the data itself is in RDF, but also events and actions are given as XML/RDF fragments (using the same URIs for entities and properties as in the static data).

Based on the semantics of the component languages as algebraic structures, a representation in RDF is straightforward for each language. Actually, when designing a language having an RDF and an XML variant (such as developed for SNOOP and CCS), the XML markup is a stripped variant of a certain RDF/XML serialization according to a target DTD of the RDF graph of the rule. The processing of rules given in RDF is actually done via transforming them to the XML syntax which is then executed as described above.

Figures 1.13 and 1.14 show an excerpt of the rule given in Figure 1.7 as RDF: “If a flight is first delayed and then canceled (note: use of a join variable), then ...”. For atomic event matching, it uses the RDF-based OWLQ language.

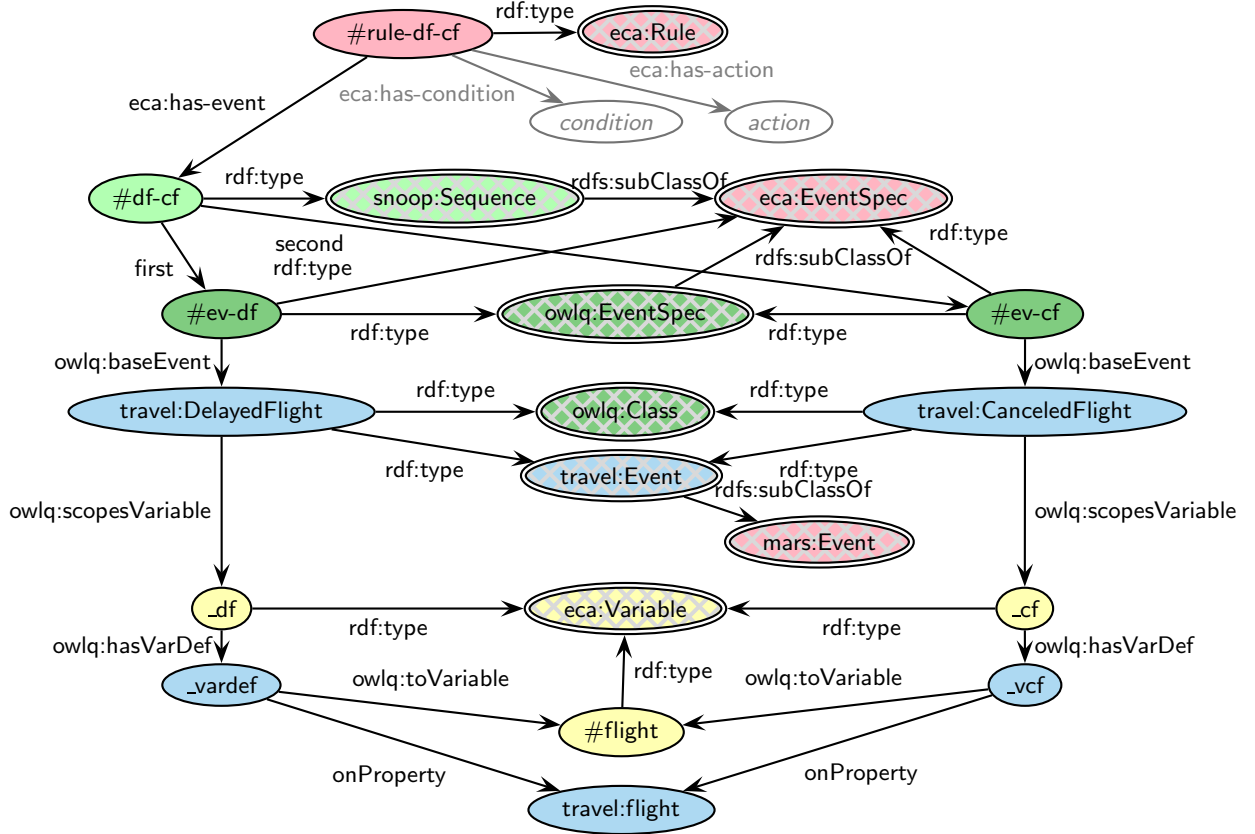


Fig. 1.13. Example Rule and Event Component as Resources

1.3.9 MARS implementation

Modular Active Rules for the Semantic Web – MARS – implements an open, service-oriented architecture exactly as described above. In MARS, every contributing service is completely autonomous. Making a language and a service

```

<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY owlq "http://www.semwebtech.org/languages/2006/owlq#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY travel "http://www.semwebtech.org/domains/2006/travel#"> ]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
  xmlns:snoop="http://www.semwebtech.org/languages/2006/snoopy#"
  xmlns:owlq="http://www.semwebtech.org/languages/2006/owlq#"
  xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
  xml:base="foo:rule">

<eca:Rule rdf:ID="rule-df-cf">
  <eca:uses-variable rdf:resource="#flight"/>
  <eca:has-event>
    <snoop:Sequence>
      <snoop:first>
        <owlq:EventSpec rdf:ID="ev-df">
          <owlq:baseEvent rdf:resource="&travel;DelayedFlight"/>
          <owlq:scopesVariable>
            <owlq:Variable>
              <owlq:hasVariableDefinition>
                <owlq:VariableDefinition>
                  <owlq:onProperty rdf:resource="&travel;flight"/>
                  <owlq:toVariable rdf:resource="#flight"/>
                </owlq:VariableDefinition>
              </owlq:hasVariableDefinition>
            </owlq:Variable>
          </owlq:scopesVariable>
        </owlq:EventSpec>
      </snoop:first>
      <snoop:second>
        <owlq:EventSpec rdf:ID="ev-cf">
          <owlq:baseEvent rdf:resource="&travel;CanceledFlight"/>
          <owlq:scopesVariable>
            <owlq:Variable>
              <owlq:hasVariableDefinition>
                <owlq:VariableDefinition>
                  <owlq:onProperty rdf:resource="&travel;flight"/>
                  <owlq:toVariable rdf:resource="#flight"/>
                </owlq:VariableDefinition>
              </owlq:hasVariableDefinition>
            </owlq:Variable>
          </owlq:scopesVariable>
        </owlq:EventSpec>
      </snoop:second>
    </snoop:Sequence>
  </eca:has-event>
  <!-- ... query and action ... -->
</eca:Rule>

```

Fig. 1.14. Sample RDF Rule

interoperable in MARS just consists of adding an appropriate language entry with a service description to the *Language and Service Registry (LSR)* and using it anywhere in a rule or process. Communication between services is always done via HTTP and the XML serialization of variable bindings.

In the MARS project, several sample languages on the XML and RDF level have been implemented.

XML Level. On the XML level, the focus is on having an XML markup for ECA rules using more or less well-known component languages that have been adapted to relational dataflow. XML is here just used as a markup format for rules and their components and subexpressions:

- Atomic Event Specifications: An XML-QL-style [32] pattern-based query mechanism,
- Composite Event Specifications: the SNOOP event algebra of the Sentinel system [28, 7],
- Queries: XPath and XQuery as *opaque* queries (i.e., non-markuped CDATA contents), XML-QL,
- Atomic Actions: An XML-QL-style pattern-based XML generation mechanism,
- Composite Actions: the *CCS – Calculus of Communicating Systems* process algebra [59, 7, 47].

Process specifications (in CCS) as used in the *Action* part of ECA rules can also be defined and executed standalone.

RDF Level. On the RDF level, all language fragments are represented in RDF/XML. Here, new language proposals are embedded in MARS:

- SNOOP and CCS (in RDF version),
- OWLQ as query language and for atomic event specifications,
- RDF-CL (an OWLQ-style RDF generation language).

Domain Services. There is a prototypical domain broker (cf. [6]) and sample RDF data for demonstrating the rules; an active domain node with a demonstrator application is under development.

Openness. The MARS framework is open for foreign component languages and other sublanguages. Languages that have an XML markup smoothly integrate as shown above. For existing services, it is an easy task to implement a wrapper Web service that provides a suitable interface and to add the respective information to the LSR (the online MARS LSR and the demonstrator contain samples of foreign languages). Languages that do not have an XML markup but any other textual syntax can be integrated using the handling of *opaque* fragments (for details, see [2]), or also via an XML-based wrapper.

An online demonstrator of MARS is available at <http://www.semwebtech.org/mars/frontend/>.

1.3.10 r^3 implementation

Resourceful Reactive Rules – r^3 – is a prototype implementation of the general framework described above that, unlike MARS, follows an integrated design that is based on a toolbox for defining and implementing heterogeneous languages in a homogeneous programming environment.

r^3 is based on an OWL-DL foundational ontology [1], describing reactive rules and their components, and fully relies on the RDF Level. I.e., in r^3 rules, and their components are resources described in RDF according to the OWL-DL foundational ontology. As such, no concrete markup is expected, though a compatibility with the ECA-ML markup described above is provided.

The prototype is actually a network of r^3 engines that cooperate towards the evaluation of ECA rules. As in MARS, the communication between the r^3 engines is done via HTTP and XML serialization of variable bindings. However, making languages and services interoperable is not as simple as in MARS. The entry point is an r^3 main agent, providing operation for loading and removing ECA rules. This main engine then interfaces with r^3 -aware language specific sub-engines, e.g. for detecting events, querying, testing conditions and performing actions. These r^3 sub-engines may either be language services or domain services.

For easing the construction of r^3 engines the prototype comes together with a toolbox, including a development library and a corresponding meta-model to describe component languages. This development library abstracts away communication protocols, bindings of variables, generation of alternative solutions, dealing with RDF models, etc. With this toolbox, building an r^3 engine for a language amounts to describing the language constructs in the meta-model, and implementing each of the constructs, or using already existing implementations. This provides a homogeneous programming environment for building r^3 engines.

In fact several r^3 engines have been built using this toolbox, for different languages and services:

- HTTP, providing functors for put, get, post and delete;
- Prova [50], allowing for querying prova (Prolog-like) rules bases, and for performing actions by using prova programs;
- XPath and XQuery for querying data in the Web;
- Xcerpt [63, 30] (see also Chapter ??), allowing for querying Web data using this language;
- SNOOP for specifying composite events;
- XChange [25] which allows for detecting events with XChange, raising XChange events, and performing XChange actions;
- Protune [10], allowing for query and acting upon policy knowledge bases, as defined in Chapter ??;
- Evolp [3] allowing to query and act in an updateable logic programming knowledge base.

Moreover, a broker has been implemented for allowing r^3 engines to access MARS, and an example domain service for the bio-informatics domains. All of

this, plus the source code of r^3 and the toolbox, installation and usage manual, as well as an online demonstrator, can be found at <http://rewerse.net/I5/r3/>.

1.4 XChange – A concrete Web-based ECA rule language

XChange [25] is a reactive rule language addressing the need for evolution and reactivity on the Web, both local (at a single Web node) and global (distributed over several Web nodes). As motivated in the beginning of this Chapter, it is based on ECA rules of the form “ON *event query* IF *Web query* DO *action*.” When events answering the event query are received and the Web query is successful (i.e., has a non-empty result), the rule’s action is executed.

In contrast to the ECA rule frameworks presented in the previous section, XChange aims at providing a single, homogenous, and elegant language that is tailored for working with Web data and that is easy to learn and use. A guiding idea of XChange is to build upon the existing Web query language Xcerpt [68, 67]. (Xcerpt is discussed in Chapter ?? of this book; the core ideas as relevant for understanding XChange will also be presented shortly here.)

XChange builds on the pattern-based approach of Xcerpt for querying data, and additionally provides for pattern-based updating of Web data [63, 30]. Development of Xcerpt, XChange, and also of the complex event query language XChange^{EQ} [18] follows the vision of a stack of languages for performing common tasks on Web data such as querying, transforming, and updating static data, as well as reacting to changes, propagating updates, and querying complex events. The result is a set of cooperating languages that provide, due to the pattern-based approach that is common to all of them, a homogenous look-and-feel. When a programmer has mastered the basics of querying Web data with Xcerpt’s query terms, she can progress quickly and with smooth transitions to more advanced tasks.

1.4.1 Representing, Querying, and Constructing Web Data

Data terms XML and other Web data is represented in XChange in the term syntax of Xcerpt that is arguably more concise and readable than the original formats. Further, data terms are the basis for query terms and construct terms, and the importance of conciseness and readability of the term syntax will become more pronounced when we introduce them shortly.

Figure 15(a) shows an Xcerpt data term for representing information about flights; its structure and contained information corresponds to the XML document shown in Figure 15(b). A data term is essentially a pre-order linearization of the document tree of an XML document. The element name, or *label*, of the root element is written first, then surrounded by square brackets or curly braces, the linearizations of its children as subterms separated by commas.

The term syntax provides two features that are not found in XML: First, child elements in XML are always ordered. The term syntax allows children to

<pre> flights [flight { number { "UA917" }, from { "FRA" }, to { "IAD" } }, flight { number { "LH3862" }, from { "MUC" }, to { "FCO" } }, flight { number { "LH3863" }, from { "FCO" }, to { "MUC" } }] </pre>	<pre> <flights> <flight> <number>UA917</number> <from>FRA</from> <to>IAD</to> </flight> <flight> <number>LH3862</number> <from>MUC</from> <to>FCO</to> </flight> <flight> <number>LH3863</number> <from>FCO</from> <to>MUC</to> </flight> </flights> </pre>
(a) Data term	(b) XML document

Fig. 1.15. An Xcerpt data term and its corresponding XML document

be specified as either ordered (indicated by square brackets []) or unordered (indicated by curly braces { }). The latter brings no added expressivity to the data format (an unordered collection can always been given some arbitrary order) but is interesting for efficient storage based on reordering elements and for avoiding incorrect queries that attempt to make use of an order that should not exist. In the example of Figure 15(a), the order of the `flight` children of the `flights` element is indicated as relevant, whereas the order of the children of the `flight` elements is not.

Second, the data model of XML is that of a tree. Our terms are more general, supporting rooted graphs, which is necessary to transparently resolve links in XML documents (specified, e.g., with IDREFs [13, 14] or with XLink [31, 17, 16]) and to support graph-based data formats such as RDF. However for understanding XChange in the scope of this article, this feature is not necessary and we therefore refer to [68, 67] for more details.

Query Terms A query term describes a pattern for data terms; when the pattern matches, it yields (a set of) bindings for the variables in the query term. Variable bindings are also called substitutions, and sets thereof (called substitution sets). The syntax of query terms resembles the syntax of data terms and extends it to accommodate variables, incompleteness, and further query constructs.

Variables in query terms are indicated by the keyword `var`. They serve as placeholders for arbitrary content and keep query results in the form of bindings. In the patterns of query terms, single brackets or braces indicate a *complete specification of subterms*. In order for such a pattern to match, there must be a one-to-one matching between subterms (or children) of the data term and the query term. Double brackets or braces in contrast indicate an *incomplete specification (w.r.t. to breadth)*: each subterm in the query term must find a match in the data term, but the data term may contain further subterms. As with data terms, square brackets indicate that the order of subterms is relevant

to the query and curly braces that it is not. *Incompleteness in depth*, that is matching subterms that are not immediate children but descendants at arbitrary depth, is supported with the construct `desc`.

Query terms also cater for restricted variables, negated subterms, optional subterms, label variables, positional variables, regular expression matching, non-structural conditions such as arithmetic comparisons, and more. Examples of query terms used in the ECA rule of Figure 1.16, which will be discussed in detail later, are the first term with root element `xchange:event` (following the keyword `ON`) and the term with root element `flights`.

Construct Terms Construct terms are used to create new data terms using variable bindings obtained by a query. A construct term describes a pattern for the data terms that are to be constructed. The syntax of construct terms resembles the syntax of data terms and extends it to support variables and grouping.

In constructing new data, *variables* in construct terms are simply replaced by the bindings obtained from the query. The result is a new data term. If there are no grouping constructs, then a new data term is generated for each binding of the variables. For more complex restructuring of data, *groupings* can be expressed as subterms in a construct term of the form `all c group by { var V }`, where `c` is another construct term (which may of course contain further grouping constructs). Its effect is to generate a data term from the construct term `c` (as subterm for the overall construct term) for each distinct binding of the variable `V`. The `group by` part can also be left out; the default then is to group by the free variables immediately inside the construct term after `all`. When grouping generates a list, the *order* of the generated subterms can be influenced with an `order by` clause. Grouping constructs can be nested.

Construct terms also cater for aggregation functions (e.g., `max`, `count`), groupings that are restricted to a fixed number of subterms, dealing with optional variables, and construction of graph rather than tree data. An examples of construct terms used in the ECA rule of Figure 1.16, which will be discussed in detail later, is the second term with root element `xchange:event` (following the keywords `DO` and `and`); note that this is a fairly simple construct term that does not make use of grouping constructs.

1.4.2 Event-Condition-Action (ECA) Rules

An XChange program consists of one or more reactive rules of the form `ON event query IF Web query DO action`.², with the intuitive meaning as described above. Both event query and Web query can extract data through variable bindings, which can then be used in the action. As we can see, both event and Web queries

² In the course of the development of XChange, different keywords and orders for the rules have also been used. In particular, rules can also be written as `RAISE event raising action ON event query FROM Web query` or `TRANSACTION update action ON event query FROM Web query`.

```

ON
  xchange:event {{
    flight-cancellation {{
      flight-number { var N },
      passenger {{
        name { "John Q Public" }
      }} }} }}
IF
  in { resource { "http://www.example.com/flights.xml", "xml" },
    flights {{
      flight {{
        number { var N },
        from   { var F },
        to     { var T }
      }} }} }
DO
  and {
    xchange:event [
      xchange:recipient [ "http://sms-gateway.org/us/206-240-1087/" ],

      text-message [
        "Hi, John! Your flight ", var N,
        " from ", var F, " to ", var T, " has been canceled."
      ] ],

    in { resource { "http://shuttle.com/reservation.xml", "xml" },
      reservations {{
        delete shuttle-to-airport {{
          passenger { "John Q Public" },
          airport   { var F },
          flight    { var N }
        }} }} }
  }
END

```

Fig. 1.16. An XChange ECA rule reacting to flight cancellations for passenger “John Q Public”

serve a double purpose of detecting *when* to react and influencing —through binding variables— *how* to react. For querying data, as well as for updating data, XChange embeds and extends the Web query language Xcerpt presented earlier.

Figure 1.16 shows an example of an XChange ECA rule, which will be used for our subsequent explanations. The individual parts of the rules employ Xcerpt and its pattern-based approach. Patterns are used for querying data in both the event and condition part, for constructing new event messages in the action part, and for specifying updates to Web data in the action part.

1.4.3 Events

Event messages Events in XChange are represented and communicated as XML messages. The root element for all events is `xchange:event`, where the prefix `xchange` is bound to the XChange namespace. Events messages also carry some meta-data as children of the root element such as

- `raising-time` (i.e. the time of the event manager of the Web node raising the event),
- `reception-time` (i.e. the time at which a node receives the event),

```

<xc:event xmlns:xc="http://pms.ifi.lmu.de/xchange">
  <xc:sender>      http://airline.com </xc:sender>
  <xc:recipient>   http://passenger.com</xc:recipient>
  <xc:raising-time> 2005-05-29T18:00 </xc:raising-time>
  <xc:reception-time>2005-05-29T18:01 </xc:reception-time>
  <xc:reception-id> 4711 </xc:reception-id>

  <flight-cancellation>
    <flight-number>UA917</flight-number>
    <passenger>John Q Public</passenger>
  </flight-cancellation>
</xc:event>

```

(a) XML syntax

```

xchange:event [
  xchange:sender      ["http://airline.com"],
  xchange:recipient  ["http://passenger.com"],
  xchange:raising-time ["2005-05-29T18:00"],
  xchange:reception-time ["2005-05-29T18:01"],
  xchange:reception-id ["4711"],

  flight-cancellation {
    flight-number { "UA917" },
    passenger { "John Q Public" }
  }
]

```

(b) Data term syntax

Fig. 1.17. Example of an event message

- **sender** (i.e. the URI of the Web node where the event has been raised),
- **recipient** (i.e. the URI of the Web node where the event has been received), and
- **id** (i.e. a unique identifier given at the recipient Web node).

An example event that might represent the cancellation of a flight with number “UA917” for a passenger named “John Q Public” is shown in both XML and term syntax in Figure 1.17.

Simple (“atomic”) event queries The event part of a rule specifies a class of events that the rule reacts upon. This class of events is expressed as an event query. A simple (or atomic) event query is expressed as a single Xcerpt query term.

Event messages usually contain valuable information that will be needed in the condition and action part of a rule. By binding variables in the query term, information can flow from the event part to the other parts of a rule. Hence, event queries can be said to satisfy a dual purpose: (1) they specify classes of events the rule reacts upon and (2) they extract data from events for use in the condition and action part in the form of variable bindings.

An XChange program continually monitors the incoming event messages to check if they match the event part of one of its XChange rules. Each time an event that successfully matches the event query of a rule is received, the condition

part of that rule is evaluated and, depending on the result of that, the action might be executed.

The event part of the ECA rule from Figure 1.16 would match the event message in Figure 1.17. In the condition and action part the variable N would then be bound to the flight number “UA917”.

Event Composition Operators To detect complex events, the original proposal of XChange supported composition operators such as **and** (unordered conjunction of events), **andthen** (ordered sequence of events), **without** (absence of events in a specified time window), etc. [33, 63, 24, 25]. This algebraic approach to query complex events with composition operators has been common in Active Database research [62, 29]; it is however not without problems and has weaknesses in terms of expressiveness and potential misinterpretations of operators [73, 38, 18, 21].

Querying Complex Events with XChange^{EQ} Later work on the complex event query language XChange^{EQ} [18, 20] seeks to replace the original composition operators of XChange with an improved and radically different approach to querying complex events. The problems associated with composition operators can be attributed to a large extent to the operators mixing different aspects of querying (see [35] and [34] for an elaboration). For example in the case of a sequence operator (**andthen**), composition of events and their temporal order are mixed.

XChange^{EQ} is built on the idea that an expressive event query language must cover the following four orthogonal dimensions, and must treat them separately to gain ease-of-use and full expressiveness:

- **Data extraction:** Events contain data that is relevant to whether and how to react. For events that are received as SOAP messages (or in other XML formats), the data can be structured quite complex (semi-structured). The data of events must be extracted and provided (typically as bindings for variables) to construct new events or trigger reactions (e.g., database updates).
- **Event composition:** To support composite events, i.e., events that are made up out of several events, event queries must support composition constructs such as the conjunction and disjunction of events (or more precisely of event queries).
- **Temporal (and causal) relationships:** Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as “events A and B happen within 1 hour, and A happens before B .” For some applications, it is also interesting to look at causal relationships, e.g., to express queries such as “events A and B happen, and A has caused B .”
- **Event accumulation:** Event queries must be able to accumulate events to support non-monotonic features such as negation (absence) of events, aggregation of data, or repetitive events. The reason for this is that the event stream is (in contrast to extensional data in a database) infinite; one therefore has to define a scope (e.g., a time interval) over which events are accumulated

when aggregating data or querying the absence of events. Application examples where event accumulation is required are manifold. A business activity monitoring application might watch out for situations where “a customer’s order has *not* been fulfilled within 2 days” (negation). A stock market application might require notification if “the *average* of the reported stock prices over the last hour raises by 5%” (aggregation).

XChange^{EQ} also adds support for deductive rules on events, relative temporal events (e.g., “five days longer than event *i*”), and enforces a clear separation between time specifications that are used as events (and waited for) or only as restrictions (conditions in the **where**-part).

The research on XChange^{EQ} also puts an emphasis on formal foundations for querying events [19]. Declarative semantics of XChange^{EQ} can be given as a model theory with accompanying fixpoint theory [18]. This is a well-understood approach for traditional (non-event) query and rule languages, and it is shown that with some important adaptations, this approach can be used for event query languages as well. Operational semantics for an efficient incremental evaluation of XChange^{EQ} programs are based on a tailored variant of relational algebra and finite differencing [19, 20]. The notion of temporal relevance is used in the operational semantics to garbage collect events that become irrelevant (to a given query) as time progresses during the evaluation [19, 20].

1.4.4 Conditions

Web queries The condition part of XChange rules queries data from regular Web resources such as XML documents or RDF documents. It is a regular Xcerpt query, i.e., anything could come after the **FROM** part of an Xcerpt rule. Like event queries in the event part, Web queries in the condition part have a two-fold purpose: they (1) specify conditions that determine whether the rule’s action is executed or not and (2) extract data from Web resources for use in the action part in the form of variable bindings.

The condition part in the rule from Figure 1.16 accesses a database of flights like the one from Figure 1.15 located at <http://www.example.com/flights.xml> (the resource is specified with a URI using the keyword **in**). It checks that the number (variable *N*) of the canceled flight exists in the database and extracts the flight’s departure and destination airport (variables *F* and *T*, respectively).

Deductive rules Web queries can facilitate Xcerpt rule chaining, that is, they can access not only extensional data (i.e., data in some Web resource) but also intensional data that has been constructed with deductive rules (i.e., results of these rules). For this, an XChange program can contain Xcerpt **CONSTRUCT-FROM** rules in addition to its ECA rules. Such rules are useful for example to mediate data from different Web resources. In our example we might want to access several flight databases instead of a single one and these might have different schemas. Deductive rules can then be used to transform the information from several databases into a common schema.

1.4.5 Actions

The action part of XChange rules has the following primitive actions: raising new events (i.e., creating a new XML event message and sending it to one or more recipients) and executing simple updates to persistent data (such as deletion or insertion of XML elements). To specify more complex actions, compound actions can be constructed from these primitives.

Raising new events Events to be raised are specified as a construct terms for the new event messages. The root element of the construct term must be labeled `xchange:event` and contain at least one child element `xchange:recipient` which specifies the recipient Web node's URI. Note that the recipient can be a variable bound in the event or condition part.

The action of the ECA rule in Figure 1.16 raises (together with performing another action) an event that is sent to an SMS gateway. The event will inform the passenger that his flight has been canceled. Note that the message contains variables bound in the event part (N) and condition part (F, T).

Updates Updates to Web data are specified as so-called update terms. An update term is a (possibly incomplete) query pattern for the data to be updated, augmented with the desired update operations. There are three different types of update operations and they are all specified like subterms in an update term. An insertion operation `insert c` specifies a construct term c that is to be inserted. A deletion operation `delete q` specifies a query term q for deleting all data terms matching it. A replace operation `replace q by c` specifies a query term q to determine data items to be modified and a construct term c giving their new value. Note that update operations cannot be nested.

Together with raising a new event, the action of the ECA rule in Figure 1.16 modifies a Web resource containing shuttle reservations. It removes the reservation of our passenger's shuttle to the airport. The update specification employs variables bound in the event part (N) and condition part (F).

Due to the incompleteness in query patterns, the semantics of complicated update patterns (e.g., involving insertion and deletion in close proximity) might not always be easy to grasp. Issues related to precise formal semantics for updates that are still reasonably intuitive even for complicated update terms have been explored in [30]. So-called snapshot semantics are employed to reduce the semantics of an update term to the semantics of a query term.

Compound Actions Actions can be combined with disjunctions and conjunctions. Disjunctions specify alternatives, only one of the specified actions is to be performed successfully. (Note that actions such as updates can be unsuccessful, i.e., fail.) The order in which alternatives are tried is non-deterministic and implementation dependent. Conjunctions in turn specify that all actions need to be performed. The combinations are indicated by the keywords `or` and `and`, followed by a list of the actions enclosed in braces or brackets.

The actions of the rule in Figure 1.16 are connected by **and** so that both actions, the sending of an SMS and the deletion of the shuttle reservation, are executed.

1.4.6 Applications

Due to its built-in support for updating Web data, an important application of XChange rules is local evolution, that is updating local Web data in reaction to events such as user input through an HTML form. Often, such changes must be mirrored in data on other Web nodes: updates need to be propagated to realize a global evolution. Reactive rules are well suited for realizing such a propagation of updates in distributed information portals.

A demonstration that shows how XChange can be applied to programming reactive Web sites where data evolves locally and, through mutual dependencies, globally has been developed in [45] and presented in [23, 22]. The demonstration considers a setting of several distributed Web sites of a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS). ECSS is subdivided into participating universities, thematic working groups, and project management. Universities, working groups, and project management have each their own Web site, which is maintained and administered locally. The different Web sites are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites.

The different Web sites maintain XML data about members, publications, meetings, library books, and newsletters. Data is often shared, for example a member's personal data is present at his home university, at the management node, and in the working groups he participates in. Such shared data needs to be kept consistent among different nodes; this is realized by communicating changes as events between the different nodes using XChange ECA rules.

Events that occur in this community include changes in the personal data of members, keeping track of the inventory of the community-owned library, or simply announcing information from email newsletters to interested working groups. These events require reactions such as updates, deletion, alteration, or propagation of data, which are implemented using XChange rules. The rules run locally at the different Web nodes of the community, allowing for the processing of local and remote events.

For a concrete example, consider changing a member's personal data including his working group affiliation. The information flow is depicted in Figure 1.18. The initial change is entered by using a Web form at the member's home university LMU. The form generates event message $m1$. One ECA rule ($r1$) reacts to this event and locally updates the member's data at LMU accordingly. Another ECA rule ($r2$) forwards the change to the management node.

The management node has rules for updating its own local data about the member ($r3$) and for propagating the change to the affected working groups ($r4$ for adding, $r5$ for deleting a member). In the example, the member changes the working group affiliation from WG2 to WG3. Accordingly, event $m4$ is sent to WG3 by rule $r4$ and $m3$ is sent to WG2 by $r5$.

with component languages have been developed. Moreover, a concrete homogeneous language, XChange, has also been defined, and integrated in the general framework. Both the language XChange and the general framework have been implemented, including the implementation of the integration of XChange in the r^3 implementation of the framework.

In other words, the work in REVERSE materialised the initial vision of an active Web, where reactivity, evolution and propagation of changes play a central role. Behaviour in the Semantic Web includes being able to draw conclusions based on knowledge in each Web node, but it also includes making updates on nodes and propagate these updates. Moreover, the specification of the behaviour must itself be part of this active Semantic Web, in as much as the specification of derivation rules must be part of the (static) Semantic Web. This requires an ontology of behaviour and rules, both derivation or reactive ones, to be formulated in this ontology, as well as concrete languages for detecting events in the Web, for querying the Web and testing conditions and for acting, including updates of Web data.

Despite all the advances made with the work described here, for having a (Semantic) Web with evolution and reactive behaviour, some issues remained untouched, and some new issues were raised, all of these calling for continuing the research in this area.

To start, taking more advantage of a semantical representation of behaviour rules is still pretty much an open issue. In our work an ontology for representing active rules semantically has been developed, and an execution framework has been realised. This semantical representation can also be used for working *on* rules and reasoning about the rules as objects themselves, e.g., doing rule analysis, verification, etc. Defining declarative semantics for reactive rules, along the lines of the existing languages in AI mentioned in the introduction, is certainly an interesting and important topic for further investigation when reasoning about the rules is desired. This work could also be seen as a generalisation for reactive rules, of the existing work of combining (derivation) nonmonotonic rules with ontologies, that is described in Chapter ???. Related with reasoning about rules is also the topic of model checking and verification methods for reactive rules in the Semantic Web. Preliminary studies have been made (cf. REX tool [36]), but much more is left to be done.

In the project we have developed specific languages for event querying in the Web and for update languages of Web data. Here again there is scope for further research, namely in detection of events at the semantic level, and on definition of updates on other data and meta-data formats on the Web such as RDF or TopicMaps. The extension of XChange (with its underlying Web query language Xcerpt) and the addition of new component languages to the general framework to deal with these data formats is an aspect of both practical relevance and research interest. Versatility [27], where data in different formats must be processed and reasoned with jointly to fulfill some task, becomes an important research issue with the inclusion of new data formats in reactive languages and

frameworks. A further research issue is that Web formats such as RDF [71] (together with RDF Schema [15]) or OWL [57] can be considered more expressive than XML, allowing to specify inferences and more constraints on data. Updates on data in these formats may thus fail (because they violate constraints) or require additional, inferred updates. A related issue is also the integration of data formats and reasoning formalisms targeted for time and location, since time and location often play an important role in reactive applications.

Related to action languages, there is the whole issue of transactions in the Semantic Web which is a very important and by now almost untouched one. With an open environment as the (Semantic) Web, transactions following the ACID (Atomicity-Consistency-Isolation-Durability) properties as in databases are not desired, if at all possible. Surely *isolation* is something quite difficult to obtain in the Web, and independent nodes cannot wait, isolated, on actions being performed by other independent nodes. However, this does not rule out a relaxed notion of transaction. For instance, in our travel example, one may want to reserve both a flight and hotel room for a stay abroad in such a way that if one of these is not possible, then none should go ahead (i.e. if I cannot book the flight, then there is no point in keeping the hotel rooms, and vice-versa). Clearly, in such a case, some notion of *atomicity* is desired, even if *isolation* is not possible since one cannot expect the flights services to wait for the hotel reservation, nor vice-versa. This calls for defining a kind of long-running transactions, in the same spirit of those defined for heterogeneous databases [39], where *isolation* is only kept for (local) subtransactions, and irreversible actions on the global environment are associated to *compensation actions*, to account for a weaker form of atomicity. Though some preliminary work on transactions in the context of Web services exists [58], a lot remains to be done for having long-running transactions in the Semantic Web.

Another interesting new issue is that of automatic generation of ECA rules. ECA rules explicitly specify reactive behaviour, giving the events and conditions under which an action will be executed. Rather than authoring all rules manually, some applications may call for the automatic generation of ECA rules from higher level descriptions. Consider for example the distributed information portal with update propagation described in Section 1.4.6. Rather than writing ECA rules for all updates that are propagated manually, it may be conceivable to generate these rules automatically from a specification that takes the form of view definitions (e.g., over a global schema) that describe which nodes mirror which data. In a similar manner, the generation of ECA rules from process descriptions (e.g., in a language such as BPEL) is interesting [26].

Finally, for putting the whole work to usage in real practical applications, more work is needed regarding the efficiency of the systems, possibly fixing a smaller set of languages and services, and also on defining programming tools and methodologies for reactive rules in the Web. In fact, efficient execution of reactive rule sets has been given little consideration so far. The efficient execution of the individual parts of an ECA, i.e., event query evaluation, query

evaluation and action processing, is well-understood. However, joint optimization of all parts of a rule as well as full rule sets has received little attention. It is conceivable for example to use multi-query optimization techniques to group together and jointly evaluate queries that are shared in multiple rules. Also, current reactive rule systems primarily work by evaluating all event queries first. It is also conceivable to use the evaluation of the condition part to enable or disable rules, thus saving the evaluation cost of the event query part when the condition part is not satisfied. Note however that this requires a mechanism where the condition part is re-checked whenever its underlying data changes.

References

1. Alferes, J.J., Amador, R.: r3 - a foundational ontology for reactive rules. In: Proceedings of 6th International Conference on Ontologies, DataBases, and Applications of Semantics, Vilamoura, Algarve, Portugal (27th–29th November 2007). Volume 4803 of LNCS. (2007) 933–952
2. Alferes, J.J., Amador, R., Behrends, E., Fritzen, O., May, W., Schenk, F.: Pre-standardization of the language. Technical Report I5-D10, REWERSE EU FP6 NoE (2008) Available at <http://www.rewerse.net>.
3. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In Flesca, S., Greco, S., Leone, N., Ianni, G., eds.: Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02). Volume 2424 of LNAI., Springer (2002) 50–61
4. Bailey, J., Poullovassilis, A., Wood, P.T.: An event-condition-action language for XML. In: Proceedings of the 11th International Conference on World Wide Web (WWW 2002), ACM Press (May 2002) 486–495
5. Baral, C., Gelfond, M., Proveti, A.: Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming* **31**(1–3) (April–June 1997) 201–243
6. Behrends, E., Fritzen, O., Knabke, T., May, W., Schenk, F.: Rule-Based Active Domain Brokering for the Semantic Web. In: Web Reasoning and Rule Systems (RR). Number 4524 in Lecture Notes in Computer Science (2007) 259–268
7. Behrends, E., Fritzen, O., May, W., Schenk, F.: Embedding Event Algebras and Process Algebras in a Framework for ECA Rules for the Semantic Web. *Fundamenta Informaticae* **82** (2008) 237–263
8. Bernauer, M., Kappel, G., G.Kramler: Composite Events for XML. In: 13th Int. Conf. on World Wide Web (WWW 2004), ACM (2004)
9. Berndtsson, M., Seiriö, M.: Design and Implementation of an ECA Rule Markup Language. In: Rule Markup Languages (RuleML). Number 3791 in Lecture Notes in Computer Science, Springer (2005) 98–112
10. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), 6–8 June 2005, Stockholm, Sweden, IEEE Computer Society (2005) 14–23
11. Bonifati, A., Braga, D., Campi, A., Ceri, S.: Active XQuery. In: Intl. Conference on Data Engineering (ICDE), San Jose, California (2002) 403–418
12. Bonifati, A., Ceri, S., Paraboschi, S.: Pushing reactive services to XML repositories using active rules. In: Proceedings of the 10th International Conference on World Wide Web (WWW 2001), ACM Press (May 2001) 633–641

13. Bray, T., et al.: Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, World Wide Web Consortium (2006)
14. Bray, T., et al.: Extensible markup language (XML) 1.1 (second edition). W3C recommendation, World Wide Web Consortium (2006)
15. Brickley, D., Guha, R.V., eds.: Resource Description Framework Schema specification (RDFS). <http://www.w3.org/TR/rdf-schema/> (2000)
16. Bry, F., Eckert, M.: Processing link structures and linkbases in the Web's open world linking. In: Proc. ACM Conf. on Hypertext and Hypermedia, ACM (2005) 135–144
17. Bry, F., Eckert, M.: Processing link structures and linkbases on the Web. In: Proc. Int. Conf. on World Wide Web, posters, ACM (2005) 1030–1031
18. Bry, F., Eckert, M.: Rule-based composite event queries: The language XChange^{eq} and its semantics. In: Proc. Int. Conf. on Web Reasoning and Rule Systems. Volume 4524 of LNCS., Springer (2007) 16–30
19. Bry, F., Eckert, M.: Towards formal foundations of event queries and rules. In: Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems. (2007)
20. Bry, F., Eckert, M.: On static determination of temporal relevance for incremental evaluation of complex event queries. In: Proc. Int. Conf. on Distributed Event-Based Systems, ACM (2008) 289–300
21. Bry, F., Eckert, M.: Rules for making sense of events: Design issues for high-level event query and reasoning languages (position paper). In: Proc. AAAI Spring Symposium AI Meets Business Rules and Process Management. Number SS-08-01 in AAAI Technical Reports, AAAI Press (2008) 12–16
22. Bry, F., Eckert, M., Grallert, H., Pătrânjan, P.L.: Evolution of distributed Web data: An application of the reactive language XChange. In: Proc. Int. Conf. on Data Engineering (Demonstrations). (2006)
23. Bry, F., Eckert, M., Grallert, H., Pătrânjan, P.L.: Reactive Web rules: A demonstration of XChange. In: Proc. Int. Conf. on Rules and Rule Markup Languages (RuleML) for the Semantic Web, Posters and Demonstrations. (2006)
24. Bry, F., Eckert, M., Pătrânjan, P.L.: Querying composite events for reactivity on the Web. In: Proc. Int. Workshop on XML Research and Applications. Volume 3842 of LNCS., Springer (2006) 38–47
25. Bry, F., Eckert, M., Pătrânjan, P.L.: Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering* **5**(1) (2006) 3–24
26. Bry, F., Eckert, M., Pătrânjan, P.L., Romanenko, I.: Realizing business processes with ECA rules: Benefits, challenges, limits. In: Proc. Int. Workshop on Principles and Practice of Semantic Web. Volume 4187 of LNCS., Springer (2006) 48–62
27. Bry, F., Furche, T., Badea, L., Koch, C., Schaffert, S., Berger, S.: Querying the Web reconsidered: Design principles for versatile Web query languages. *Int. J. on Semantic Web and Information Systems* **1**(2) (2005) 1–21
28. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite events for active databases: Semantics, contexts and detection. In: Proceedings of the 20th VLDB. (1994) 606–617
29. Chakravarthy, S., Mishra, D.: Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering (DKE)* **14** (1994) 1–26
30. Coşkun, F.: Pattern-based updates for the Web: Refinement of syntax and semantics in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich (2007)
31. DeRose, S., Maler, E., Orchard, D.: XML linking language (XLink) version 1.0. W3C recommendation, World Wide Web Consortium (2001)

32. Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D.: XML-QL: A Query Language for XML. In: 8th. WWW Conference, W3C (1999) World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, www.w3.org/TR/NOTE-xml-ql.
33. Eckert, M.: Reactivity on the Web: Event queries and composite event detection in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich (2005)
34. Eckert, M.: Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events. PhD thesis, Institute for Informatics, University of Munich (2008) <http://edoc.ub.uni-muenchen.de/9405/>.
35. Eckert, M., Bry, F.: Rule-based composite event queries: The language XChange^{eq} and its semantics. *Int. Journal on Knowledge and Information Systems* (2009) To appear.
36. Ericsson, A., Berndtsson, M.: Rex, the rule and event explorer. In Jacobsen, H.A., Mühl, G., Jaeger, M.A., eds.: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems – DEBS*. Volume 233 of *ACM International Conference Proceeding Series.*, ACM (2007) 71–74
37. Fritzen, O., May, W., Schenk, F.: Markup and Component Interoperability for Active Rules. In: *Web Reasoning and Rule Systems (RR)*. Number 5341 in *Lecture Notes in Computer Science*, Springer (2008) 197–204
38. Galton, A., Augusto, J.C.: Two approaches to event definition. In: *Proc. Int. Conf. on Database and Expert Systems Applications*. Volume 2453 of *LNCS.*, Springer (2002) 547–556
39. Garcia-Molina, H., Salem, K.: Sagas. In Dayal, U., Traiger, I.L., eds.: *Proceedings of the ACM Special Interest Group on Management of Data*, ACM Press (1987) 249–259
40. Gelfond, M., Lifschitz, V.: Representing actions and change by logic programs. *Journal of Logic Programming* **17** (1993) 301–322
41. Gelfond, M., Lifschitz, V.: Action languages. *Electronic Transactions on AI* **3**(16) (1998)
42. Giunchiglia, E., Lee, J., Lifschitz, V., Cain, N.M., Turner, H.: Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming* **31** (1997) 245–298
43. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. *Artificial Intelligence* **153** (2004) 49–104
44. Giunchiglia, E., Lifschitz, V.: An action language based on causal explanation: Preliminary report. In: *AAAI'98*. (1998) 623–630
45. Grallert, H.: Propagation of updates in distributed web data: A use case for the language XChange. Project thesis, Institute for Informatics, University of Munich (2006)
46. Hoare, C.: *Communicating Sequential Processes*. Prentice Hall (1985)
47. Hornung, T., May, W., Lausen, G.: Process algebra-based query workflows. In: *Conference on Advanced Information Systems Engineering (CAiSE)*. (2009) to appear.
48. Kifer, M., Lausen, G.: F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In Clifford, J., Lindsay, B., Maier, D., eds.: *ACM Intl. Conference on Management of Data (SIGMOD)*, Portland (1989) 134–146
49. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. *New generation Computing* **4** (1986) 67–95

50. Kozlenkov, A., Schroeder, M.: PROVA: Rule-based Java-scripting for a bioinformatics semantic web. In Rahm, E., ed.: *International Workshop on Data Integration in the Life Sciences - DILS*, Springer (2004)
51. Liu, M., Lu, L., Wang, G.: A Declarative XML-RL Update Language. In: *Proc. Int. Conf. on Conceptual Modeling (ER 2003)*. LNCS 2813, Springer-Verlag (2003)
52. Magiridou, M., Sahtouris, S., Christophides, V., Koubarakis, M.: Rul: A declarative update language for rdf. In Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A., eds.: *International Semantic Web Conference*. Volume 3729 of *Lecture Notes in Computer Science*, Springer (2005) 506–521
53. May, W.: XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming* 4(3) (2004) 239–287
54. May, W., Alferes, J.J., Amador, R.: Active rules in the semantic web: Dealing with language heterogeneity. In: *Rule Markup Languages (RuleML)*. Number 3791 in *Lecture Notes in Computer Science*, Springer (2005) 30–44
55. May, W., Alferes, J.J., Amador, R.: An ontology- and resources-based approach to evolution and reactivity in the semantic web. In: *Ontologies, Databases and Semantics (ODBASE)*. Number 3761 in *Lecture Notes in Computer Science*, Springer (2005) 1553–1570
56. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* (4) (1969)
57. McGuinness, D., Harmelen, F., eds.: *OWL Web Ontology Language*. <http://www.w3.org/TR/owl-features/> (2004)
58. Mikalsen, T., Tai, S., Rouvellou, I.: Transactional attitudes: reliable composition of autonomous web services. In: *Dependable Systems and Networks Conference*. (2002)
59. Milner, R.: Calculi for synchrony and asynchrony. *Theoretical Computer Science* (1983) 267–310
60. OASIS Web Services Business Process Execution Language Technical Committee: *Business Process Execution Language (BPEL)*. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>
61. Papamarkos, G., Poulouvasilis, A., Wood, P.T.: RDFTL: An Event-Condition-Action Rule Language for RDF. In: *Hellenic Data Management Symposium (HDMS'04)*. (2004)
62. Paton, N.W., ed.: *Active Rules in Database Systems*. *Monographs in Computer Science*. Springer (1999) ISBN 0-387-98529-8.
63. Pătrânjan, P.L.: *The Language XChange: A Declarative Approach to Reactivity on the Web*. PhD thesis, Institute for Informatics, University of Munich (2005)
64. Prudhommeaux, E., Seaborne, A., eds.: *SPARQL Query Language for RDF*. <http://www.w3.org/TR/rdf-sparql-query/> (2006)
65. Romanenko, I.: *Use cases for reactivity on the Web: Using ECA rules for business process modeling*. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich (2006)
66. Sandewall, E.: *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems*. Oxford University Press (1994)
67. Schaffert, S.: *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute for Informatics, University of Munich (2004)
68. Schaffert, S., Bry, F.: *Querying the Web reconsidered: A practical introduction to Xcerpt*. In: *Proc. Extreme Markup Languages*. (2004)
69. Tatarinov, I., Ives, Z.G., Halevy, A., Weld, D.: Updating XML. In: *ACM Intl. Conference on Management of Data (SIGMOD)*. (2001) 133–154

70. Widom, J., Ceri, S., eds.: Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann (1996)
71. World Wide Web Consortium: Resource Description Framework (RDF). <http://www.w3.org/RDF> (2000)
72. XML:DB: Xupdate - xml update language. <http://xmldb-org.sourceforge.net/xupdate/> (2000)
73. Zhu, D., Sethi, A.S.: SEL, a new event pattern specification language for event correlation. In: Proc. Int. Conf. on Computer Communications and Networks, IEEE (2001) 586–589
74. Zimmer, D., Unland, R.: On the semantics of complex events in active database management systems. In: Intl. Conference on Data Engineering (ICDE). (1999) 392–399