A general language for Evolution and Reactivity in the Semantic Web

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

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

Abstract. In this paper we define the basic concepts for a general language for evolution and reactivity in the Semantic Web. We do this by exposing an UML model that specifies an ontology for the language. The proposed language is based on Event-Condition-Action rules, where different languages for events (including languages for composite events), for conditions (queries) and actions (including complex actions) may be composed, this way catering for language heterogeneity (besides heterogeneity on the data-model) that we think is essential for dealing with evolution and reactivity in the Semantic Web.

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. This vision of the Web, as well as a state of the art overview of related areas, is described in our previous work [18].

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), is capable of updating persistent data, communicating the changes, requesting changes of persistent data in other systems, and being able to react to requests from 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 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.

The importance of being able to update the Web has long been acknowledged, and several languages exist (e.g. XUpdate [25], XML-RL [16], XPathLog [17]) for just that. More recently some reactive languages have been proposed, that 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 upon that act by updating data. These are the cases of the XML active rules of [6], of Active XQuery [5], of the Event-Condition-Action (ECA) language for XML defined in [2], and RDFTL [19], which is an ECA reactive language on RDF data. The common aspect of all of these languages is the use of ECA (declarative) 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 [20, 24]) have the general form **on** *event* **if** *condition* **do** *action*. They are intuitively easy to understand, and provide a well-understood formal semantics: when an event (atomic or composite) occurs, evaluate a condition, and if the condition is satisfied then execute an action (or a sequence of actions, a program, a transaction, or even start a process).

In fact, we agree with the arguments exposed for the definition of the above languages in what regards adopting ECA rules for dealing with evolution and reactivity in the Web (declarativity, modularity, maintainability, etc). But in our opinion, these languages fall short in various aspects, when the goal is aimed at the general view of an evolving Web as described above. Namely, they lack on allowing for more complex events and actions and, most important, on dealing with heterogeneity at the level of the language. Autonomous web nodes will use different formalisms for ECA rules, and also different formalism for events, conditions and actions, depending on the requirements of their applications.

In general, actions are more than just simple updates to Web data (be it XML or RDF data). As said above, besides that, actions can be notifications to other resources, update requests of other resources, can be composition of simpler actions (like: do this, and then do that), or even transactions whose ACID properties ensure that either all actions in a transaction are performed, or nothing is done. In our view, a general language should cater for such richer actions. 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 St. Wendel is delayed ...). Moreover, as in active databases [10, 26], there may be more complex (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 this respect, there is some preliminary work on composite events in the Web [3], but that only considers composition of events of modification of XML-data in a single document.

The quite recent work on the language XChange [8] already aims at having more complex actions and events for evolution and reactivity on the Web and, in our opinion, is an important contribution in this direction. However, having in mind the requirements we set up for the general evolving Semantic Web, there are still some important aspects, that are not yet dealt with by XChange, namely that of language heterogeneity.

The problem of language heterogeneity will definitely appear when dealing with evolution and reactivity on the Web. This calls for more general languages. In such an open and heterogeneous environment as the Web, it is difficult to assume that there will be *a single* event language, or *a single* way to deal with actions. Our view is that a general language for evolution and reactivity in the Web should allow for the usage of different event languages, different condition languages, and different action languages, considering ontological descriptions and mappings for these languages. Each of these different (sub)languages should have some minimal requirements, but it should be as free as possible. The task of the general ECA language is then to combine these various (sub)languages for reacting and performing evolution in the (Semantic) Web. This requirement is far from the goals of XChange, which is based on a concrete language for all the parts of ECA.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general, especially if one wants to reason about evolution, ECA rules (or parts of them) must be communicated between different nodes, and may themselves be subject to being updated. For that, the ECA rules themselves must be represented as data in the (Semantic) Web. This need calls for a (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [4], but it does not tackle the complexity of events, actions, and the generality of rules, as described here. Moreover, to deal with the requirements of heterogeneity and of reasoning about rules, an ontology of ECA rules and (sub)ontologies for events, conditions and actions, with rules possibly specified in RDF/OWL, is required.

In this paper we define the basic concepts of a general language for evolution and reactivity in the Semantic Web that responds to the requirements just exposed. Rather than presenting an RDF/OWL ontology, in this paper we present a UML 2.0 [15] model. By doing this, we not only consubstantiate the language concepts, but also provide an abstract syntax for it, which is already a step for having a markup (XML) language for general ECA rules. For defining such a markup, it is worth noting that the UML model we present is mappable into XMI [14], this directly providing an XML representation. The modelling of the language starts in Section 2, where the global aspects are spelled out, and the composition between the various parts is discussed. The common structure of the (sub)languages for the rule parts discussed in Section 3. Then, in Section 4, the specific aspects each of the E, C and A parts is discussed and an illustrative concrete (instantiation) of each of these (sub)languages is given. We end the paper by mentioning ongoing and future work.

2 Global aspects for a general ECA language

In order to cope with the Semantic Web heterogeneity, the target of development and definition of languages for (ECA) rules, for events, for conditions and for actions should be a semantic approach, i.e., an approach based on an (extensible) ontology for rules, events, conditions and actions that also allows for reasoning about these concepts.

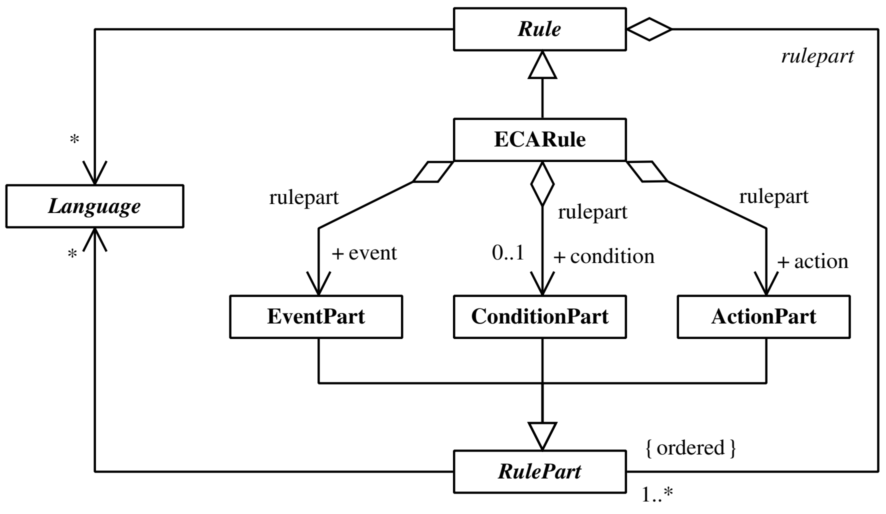

Fig. 1. General UML model for ECA rules

At a quite abstract level a rule is a composite aggregation of ruleparts and an ECA rule may be defined by the UML diagram of Figure 1. As expected, an ECA rule has 3 different parts: event, condition and action. The condition part is optional, in the sense that it can be omitted, or that languages may allow to integrate the evaluation of the condition with the event part, or with the action part. This model can be readily extended by adding a fourth rule part (also optional) – the post-condition (a *ConditionPart*) – resulting in a variation usually called ECAP rules. In most cases, this post-condition part can be omitted by allowing the action language to test for conditions inside the action part. But this rule part may have particular relevance when considered together with cascading reactions and transactional rules, in which case the post-condition part allows the declarative specification of restrictions that must apply after the whole transaction, given by the action part, is successfully executed. This will be further detailed below, when discussing action languages.

Current databases already support active concepts by triggers (e.g., SQL) where the distinction between events, conditions and action is not necessarily explicit. Such rules can be handled as *opaque* rules of a given language that are understood as a whole by an underlying system. Note that there exist well-defined mappings into the above ECA model.

When defining (ECA) rules, language heterogeneity has to be considered not only at the global rule level, but also at the rule part level. As stated before, several reactive rule languages have already been proposed (e.g. XChange [8], RDFTL [19]), introducing heterogeneity at the rule level. A generic approach for rules in the Semantic Web must be able to cover *all* such explicit language proposals. In most of these proposals there exists a pattern of language reuse, usually a query (sub)language that already exists (e.g. XQuery) is chosen for the condition part and either an existing update (sub)language is chosen (e.g. XQuery+Updates [22]) or an extension is built over the query language (e.g. Xcerpt [21]) in order to obtain a new (sub)language (e.g. XChange [8]) for the action part. Finally, an event (sub)language is defined (often based on an existing one from the field of Active Databases, e.g. the SQL3 standard).

A general approach should lead to a clean distinction between the three parts E, C, and A on the ontology level of the rules (as shown in Figure 1). Given this, additional heterogeneity is provided by using and combining different event, condition, and action sublanguages according to a global ECA schema. At the most basic level, a rule part has a textual specification. To achieve language heterogeneity at the rule part level, one must define precisely how the different parts of a rule can exchange information and interact with each other. This is achieved by a set of "bindable" names (logical variables), cf. Figure 2. These variables are declared on the rule level. In every rule part that uses them, it must be declared if they are bound by that part, or used (which requires that they are bound in the same or a preceding part). A variable must be bound only once to a value; in case that an already bound variable is "bound" again, the values must coincide, i.e., yielding an analogous semantics as in logic programming (this e.g. allows an event part that in some cases binds a variable, which is then used as a join variable in the condition part, and is otherwise bound by the latter). For each use of a logical variable (of the rule) its name in the opaque code must also be given as a use attribute (e.g., for embedding JDBC where variables are only given as ?1, ?2 etc.). Variables can be bound by (i) matching them – logic programming style, (ii) or assigning the result set of a query to them.

In case that a variable is to be bound to a result set (e.g., of a query), no use is given. This binding mechanism can be extended with a type system.

Example 1. Consider an ECA rule expressing the idea that whenever a flight is cancelled, every customer who has a reservation for this flight must be notified, preferably by SMS. Using XML, this rule could take the following form (namespace declarations omitted):

```
<eca:rule>
  <eca:bind-variable name="Reservations">
    http://www.reservations.nop/actual.xml
  </eca:bind-variable>
  <eca:variable name="Flight"/>
  <eca:variable name="Customers"/>
  <eca:event lang="http://www.flights.org/datalog">
    <eca:bind-variable name="Flight" use="F" mode="match"/>
    <eca:opaque>
      flight_cancelation(F)   <!-- matches literal against incoming event in datalog -->
```

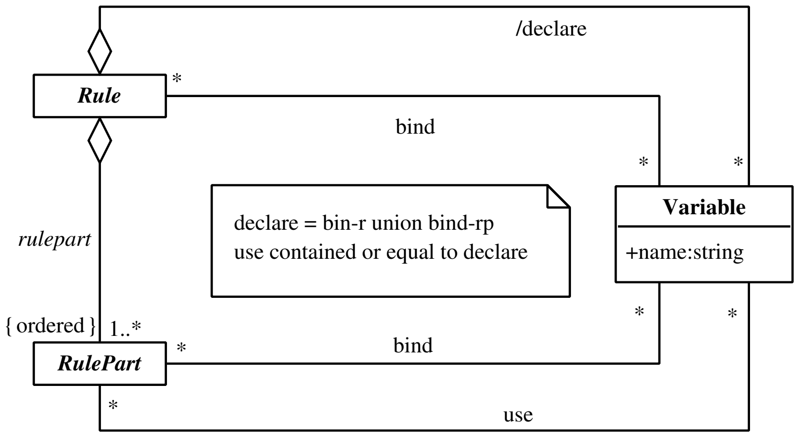

Fig. 2. Rules and Variables

```
    </eca:opaque>
  </eca:event>
  <eca:condition lang="http://www.w3.org/XPath">
    <eca:use-variable name="Flight" use="$Flight"/>
    <eca:use-variable name="Reservations" use="$Reservations" />
    <eca:bind-variable name="Customers" mode="result-set"/>
    <eca:opaque>
      document($Reservations)//flight[@id=$Flight]/reservation/customer
    </eca:opaque>
    <!-- evaluates XPath expression, binds the result to the variable 'Customers'
      and checks if it is not empty -->
  </eca:condition>
  <eca:action lang="http://www.pseudocode-actions.nop">
    <eca:use-variable name="Customers" use="Customers"/>
    <eca:use-variable name="Flight" use="Flight"/>
    <eca:opaque>
      for each C in Customers do
        notify_cancelation(Flight, sms:C)
          otherwise notify_cancelation(Flight, mail:C)
            otherwise signal_failure(notify_cancelation(Flight, C))
      done
    </eca:opaque>
  </eca:action>
</eca:rule>
```

Note in this example that each of the rule parts has a different lang attribute, i.e. each of the parts uses a different language.

3 Common Structure of E, C and A Sublanguages

The level of reasoning that can be performed with the model defined so far is yet restricted. In order to improve this level of reasoning one must know more about the structure of a (sub)language.

The generic structure of (sub)languages, independently of whether they are event, condition or action languages, is modelled in Figure 3. Each such language consists of a set of *composers* and is parameterized with a separate language of *atomic* elements. Expressions of the language are then (i) atomic elements, or (ii) composite expressions recursively obtained by applying composers to expressions. The atomic elements are part of a *domain language*, and in most cases come from an application-dependent ontology.

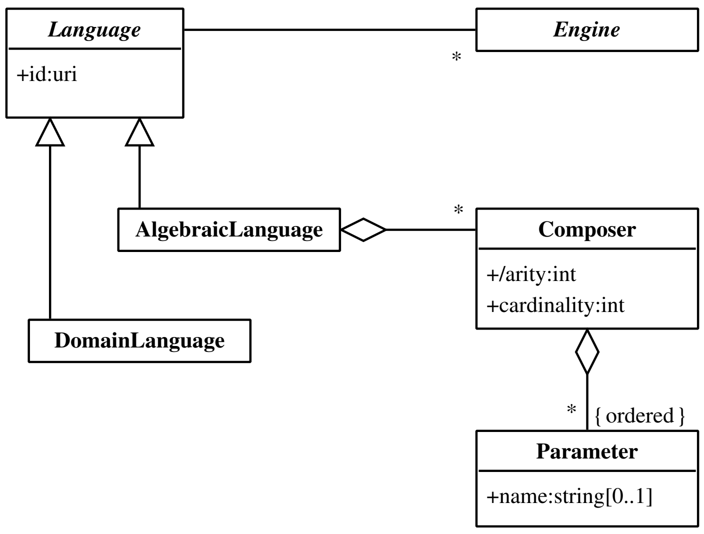

Fig. 3. Composers

Each composer has a given *cardinality* that denotes the number of expressions it can compose, and optionally has a number of parameters that determines its *arity*. Due to their structure, these languages are called *algebraic languages*, e.g. used in *event algebras*.

For instance, "E_1 followed_by E_2 within t" is a binary composer to recognize the occurrence of two events (atomic or not) in a particular order within a time interval, where t is a parameter. A language for *atomic* events that is consistent with this event algebra could define an event "received_message(M)" for receiving a message. Together with an action language that provides an action for sending a message, one could easily define a negotiation dialog between two systems by means of a set of reactive rules.

Each of these languages has an associated engine that captures the semantics of the (composers of the) language. The engines provide the (expected) interfaces for communication, must keep their own state information, including at least the current variable bindings. Specific tasks of the engines then include e.g. the evaluation of composite events (for the event languages), or the execution of transactions (for the action engines). The issue of transactions is of particular importance (see Section 4.3).

As mentioned above, for using such a language it must be parameterized with a language for atomic elements, this latter language being (an appropriate) part of a domain language. Such a domain language (e.g. a language for the domain of travels, or banking, or ...) is induced by an ontology, and provides atomic events, predicates or literals (for conditions), and atomic actions of that specific domain (e.g. events of train schedule changes, actions of reserving tickets, ...). Moreover, primitive constructs (with arguments) for events, conditions and actions must also be provided by the domain language. The above received_message(M) event is an example of such a primitive construct for the language of atomic events, where M is an argument for the message, itself described in the ontology of the domain.

Given the additional level of knowledge about the structure of a (sub)language, the modelling of the rule part specifications can be more detailed, as shown in Figure 4. Here, more information about the actual bindings of the available variables is provided. This raises the level of reasoning that may be performed about ECA rules, regardless of the degree of language heterogeneity that may be present.

The specification of a rule part (*PartSpec* in Figure 4) in its simplest (opaque) form is just some text (understood by some language engine) associated with a set of variables that can either be bound or simply used in that part. When this text is given to the respective language engine together with a set of variables, some of them already bound, the engine interprets this text, optionally producing new bindings for some of the yet unbound variables. Instead of a simple text (like the opaque specifications - eca:opaque - used in Example 1), a specification may also be marked up according to the composers of the language (see Example 2), defining an abstract tree. This abstract tree is built from atomic elements and composite specifications, respecting the defined arity and cardinality for composers. In this case, the bindings of variables can be made either inside atomic elements (described in the domain language), or with parameters of composers.

4 Concrete languages for Events, Conditions and Actions

In the previous sections we have defined a general model for ECA rules. This model allows for combining different event, condition and action languages. According to the model, each such language is specified by a set of composers, and elements of a domain language that may include basic constructs possibly

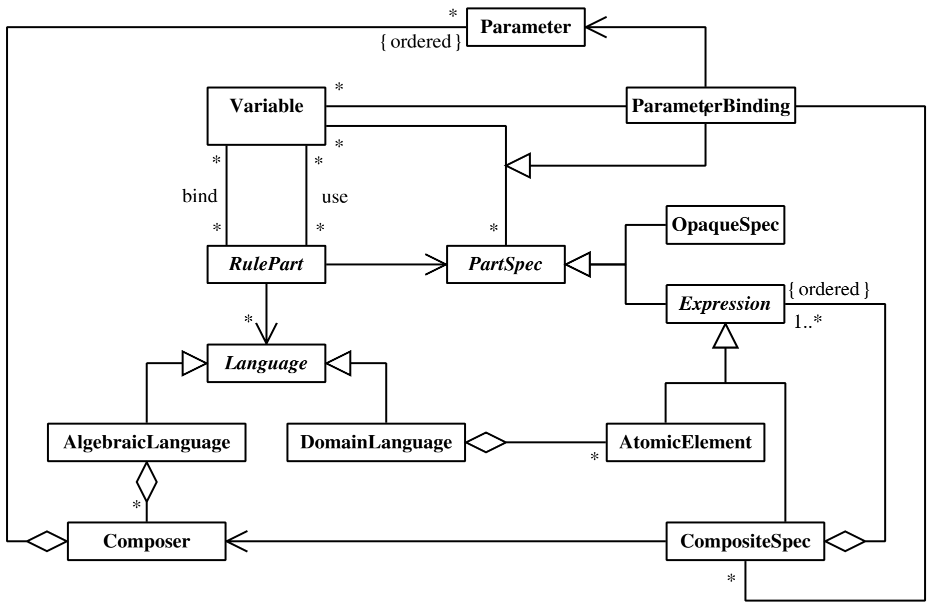

Fig. 4. Model of the Rule Part

with arguments. In this section we discuss the definition of these constructs and composers for basic languages of events, conditions and actions.

4.1 A language for events in the Web

In the context of the Semantic Web, an (atomic) event is in general any detectable occurrence. Events in the Web can be local events, e.g. updates of local data, but also incoming messages, and (explicitly communicated, or otherwise) changes in other nodes. Accordingly, a general event language should have constructs for specifying these various kinds of events, besides the domain specific ones.

Atomic Events on Web data. In the most basic (physical) level, there should be constructs to cater for the detection of changes on local data, be it on XML or RDF data, similar to those found in database triggers. Work on triggers for XQuery has e.g. been described in [5] with *Active XQuery* and in [2], emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases. The former uses the same syntax and switches as SQL. For modifications of an XML tree, we propose the following basic constructs for atomic events of modifications of XML data, that have been developed in [1]:

– `ON {DELETE|INSERT|UPDATE} OF` *xsl-pattern*: if a node matching the *xsl-pattern* is deleted/inserted/updated,

– `ON MODIFICATION OF` *xsl-pattern*: if anything in the subtree is modified,
– `ON INSERT INTO` *xsl-pattern*: if a node is inserted (directly) into a node matching the *xsl-pattern*,
– `ON INSERT [IMMEDIATELY] BEFORE|AFTER` *xsl-pattern*: if a node is inserted (immediately) before or after a node matching the *xsl-pattern*.

In all these constructs, *xsl-pattern* is a (typically input) argument. Moreover, these events should make relevant values accessible, e.g., `OLD AS ...` and `NEW AS ...` (like in SQL), both referencing the complete node to which the event happened, additionally `INSERTED AS`, `DELETED AS` referencing the inserted or deleted node. These relevant values are additional arguments of the above constructs, typically (output) to be bound with variables. The implementation of these events in XML repositories is probably to be based on the *DOM Level 2/3 Events* [11].

Regarding RDF data, RDF triples, describing properties/values of a resource, are much more similar to SQL. In contrast to XML, there is no assignment of data with subtrees, which makes it impossible to express "deep" modifications in a simple event. A proposal can be found in [19], and in [1]. The latter considers the following basic constructs:

– `ON {DELETE|INSERT|UPDATE} OF` *property* `[OF` *class*`]`: if a property is removed from/added to/updated of a resource of a given class, then such an event is raised;
– `ON CREATE OF` *class*: it is raised if a new resource of a given class is created;
– `ON NEW CLASS`: is raised if a new class is introduced,
– `ON NEW PROPERTY [OF CLASS` *class*`]`: is raised, if a new property (optionally: to a specified class) is introduced.

Besides the `OLD` and `NEW` values mentioned for XML, these events should consider as arguments (to bind variables) `RESOURCE AS ...` and `PROPERTY AS ...` referring to the modified resource and the property (as URIs), respective.

Communication events. Besides the above events that react on updates on a given data model level, communication events are raised by messages, independent from the abstraction level of the rule. We propose the following basic constructs:

– `ON MESSAGE` *content* `[OF` *sender*`] [AT` *time*`]`
– `ON MESSAGE MATCHING` *pattern* `[OF` *sender*`] [AT` *time*`]`

In the first construct, typically both the *content* and *time* arguments are to be bound to variables upon receipt of the message. However, one might want to trigger such an event only when a message with a specific content (or a content matching a given pattern) is received, as in the second construct. In this case a methodology for making the matching of *pattern* must be specified. More elaborate constructs for incoming messages are possible, e.g. with parameters for specifying an ontology describing the language of the message, or along the lines of the FIPA language for communication among agents [12].

Composite events. In general, as argued above, events should not be restricted to atomic ones. For dealing with composite events in the context of the ECA rules proposed here, the event language must define several composers. We propose at least the following composers of events: "E_1 `OR` E_2", "E_1 `AND` E_2", and "E_1 `AND THEN` E_2 `[AFTER PERIOD` *time*`]`" the latter one with two events and an additional parameter *time*, standing for the time that has passed between the occurrence of E_1 and E_2. The actual semantics of composers must be done similarly to that of operators in event algebras in the context of active databases [26]. In it, detection of a composite event means that its "final" atomic subevent is detected. Event algebras contain not only the aforementioned straightforward basic conjunctive, disjunctive and sequential connectives, but also additional operators. A bunch of event algebras have been defined that provide also e.g. "negative events" in the style that "when E_1 happened, and then E_3 but not E_2 in between, then do something", "periodic" and "cumulative" events, e.g., the SNOOP event algebra [9] of the "Sentinel" active database system. A quite rich set of composers for events in the Web is being also considered in the language XChange [8], where exclusions, repetitions, and cardinality are also being explored.

Example 2. The following specifies, in an illustrative (XML) markup, an event for (very simplified) detection of a late train. It is a composite event in the SNOOP (algebraic) language, and uses a basic atomic event language of a domain of train travels, including constructs of those just exposed. The detection of late trains is made either by being warned by the travel agency, or by the occurrence of domain specific event signaling changes in a given (pre-defined) source with expected arrival times:

```
<eca:event xmlns:snoop="http://snoop.nop"
  xmlns:msg="http://www.basic.nop/events"
  xmlns:mytravel="http://www.trains.tr">
  <eca:bind-variable name="newArrival"/>
  <!-- The 2 variable below must be bound as constants on the rule level -->
  <eca:use-variable name="myTravelAgent" use="$myAgent"/>
  <eca:use-variable name="myTrain" use="$myTrain"/>
  <snoop:or>
    <snoop:atomic detect="xml-pattern">
      <msg:receive-message sender="$myAgent">
        <content> <delayed train={$myTrain}/> </content>
      </msg:receive-message>
      <snoop:variable name="newArrival" select="
          $event/content/delayed/@arrivalTime" />
    </snoop:atomic>
    <snoop:atomic detect="xpath">
      <snoop:cond test="$event/name()='mytravel:changeTime'">
      <snoop:cond test="$event/@trainId=$myTrain">
      <snoop:variable name="newArrival" select="$event/newTime"/>
    </snoop:atomic>
  </snoop:or>
```

```
</eca:event>
```

This event part binds the variable newArrival with the (reported or detected) new time of arrival. It is worth noticing here how, in each of these cases, the variable is bound. In the case of reporting, there must occur an event of a received message (marked-up in XML and represented by $event) with an attribute sender which is equal to the value at the variable myAgent, and with a content with a delayed element with an attribute train equaling that of myTrain. The mechanism used here for testing this matching with the event that occurred is an xml-pattern. In this case, newArrival is bound to the value in attribute arrivalTime of that delayed element. In the other case, a domain specific event changeTime must have occurred, and this event must have an attribute trainId equal to that in the variable myTrain, this matching being made by xpath. In this case, newArrival is bound to the value in element newTime of that delayed element.

4.2 Conditions in ECA rules for the Web

Conditions in ECA rules basically amount to queries in the (Semantic) Web, that possibly bind rule variables to be then used in the action part. For this purpose, and in case reasoning is not required inside the condition part, one can envisage the condition language specification simply as opaque (see Section 2), where besides the reference for the language being used (possibly with URI for the respective language and engine) one further gives a text string with the query in that language. This way, e.g. XPath, XQuery, RDQL, or Xcerpt can be used in the condition part.

In case reasoning about the condition part is desired, an ontology for the query language(s) is needed, that models the basic constructs and composers of the language in the terms described above. Such work in the direction of modelling query languages for the Web already exists, e.g. in [23] where a UML modelling of the language Xcerpt [21] is shown.

4.3 Actions and Transactions

As for events, also (atomic) actions in the Web can be considered at various levels: there can be local actions of updating web data; event raising; external update requests to other nodes; general (local or remote) method calls.

Local update actions can be specified in any appropriate language for changing web data, such as XUpdate [25], XML-RL [16], or XPathLog [17]. Their integration in the ECA framework can be done as just described for conditions, i.e. either as opaque specification, or by providing a proper ontology, based on constructs and composers, specifying those update languages.

Activities of remote nodes can be invoked by sending a message with an update (request) statement. Here a basic construct for sending a message is required, the simplest one being: `SEND MESSAGE` *message* `TO` *recipient*. This message sending can also be used for event-raising actions, in this case making sure that the event raised is then collected by a corresponding `ON MESSAGE` construct.

As for events, more elaborate action constructs can be defined. General action constructs that can be defined may be those for (remote) procedure/method calls to Web Services, where the SOAP protocol can be used.

The execution of an action may in general succeed or fail. Considering failure of actions is important e.g. in the case of remote update requests: once the request is issued, it is important to be able to receive feedback on whether the update was actually done, or not. For example, upon request of a flight reservation, it is important to know whether the reservation was accepted or not. The operational semantics of a general language for actions, and a corresponding processor, should thus allow for failure of atomic actions. Moreover, when used with non-deterministic condition languages, the failure of an action should somehow "backtrack" into the condition part to check for alternative bindings of variables that may result in successful actions.

Complex actions can be defined by composing atomic actions. This is done by enriching the action language with appropriate composers. The most basic composers for actions are those of (parallel) conjunction of actions (A_1 `AND` A_2) sequential execution of actions (A_1 `;` A_2). Other more elaborate composers can be defined in action languages, such as if-then-else composers (`IF` *test* `THEN` A_1 `ELSE` A_2), while-iterations (`WHILE` *test* `DO` A), and forall-iterations (`FORALL` *variable* `DO` A). Note that some of these complex actions already require the use of a condition language in the action language for evaluating conditions. This idea can be further exploited by introducing an action construct – `TEST CONDITION` *condition* which tests the condition and either fails if the condition is false, or does nothing in case it is true but possibly binding some extra variables). With such a rich action language, similar to Transaction Logic [7], combining condition testing with (trans)actions, the condition part of rules can be omitted.

In general, each of the complex actions should be allowed to be specified as a transaction with ACID properties, in particular where either all of the actions are executed, or the whole composite actions fails, and no action is performed. This can be done by having a composer `TRANSACTION` *id* A, where *id* is a parameter for storing a unique identifier of the transaction, and A is the (complex) action. Note here how some form of post-condition, in the line of those mentioned in Section 2 may be specified by combining these transactions with the above condition testing. While the transaction composer is easy to understand in case all atomic actions in A consist of local updates, this is not the case when A involves actions like e.g. sending messages, or remote method calls. In fact, in these cases, what should be the meaning of rolling back over such an action? When a message is sent, what does it mean to rollback on sending it? It is our stance that in these cases, compensation actions must be specified, to be executed when rolling back is not possible. This, and a deeper study of transactions in this context (including considering transactions that are not limited to a single rule), is not detailed further here, and is subject of ongoing work.

5 Conclusions

In this paper we describe the basic concepts and a UML modelling of an ECA-rules-based general language for the Semantic Web. Moreover, we discuss concrete languages for events, conditions and actions to be composed in this general language. It is our stance that this sets the ground for a general framework for evolution and reactivity on the Web, where heterogeneity of languages is taken into account, and reasoning over the rules is made possible. In particular, it may serve as a general framework for integrating several existing languages. The integration of other ECA-based languages in this framework, such as the ones mentioned in the introduction, is a subject of ongoing and future work. In this respect, special attention will be paid to the language XChange, as it is the one which already consider richer events and actions.

Lack of space prevents us from elaborating here on further ongoing work that is being developed by us in the context of the general language. Namely, further detailing the concepts involved in the definition of domain languages, and also the definition of a general architecture for executing the ECA-Rules are left out. This general architecture also raises the issue of communication strategies regarding event and actions (are events raised by actions "pushed" into (respective) nodes? or do they (periodically) "pull" for events that may have occurred?). Another important issue that is also related with the execution, and that was only briefly addressed here is that of transactions. It is our belief that the issue of transactions on the Web is an important and difficult subject, that will gain increasing importance and interest in a near future. It is in our agenda to continue working in this subject, along the lines exposed above.

Acknowledgements

This research has been funded by the European Commission within the 6th Framework Programme project REWERSE, number 506779.

References

1. J. J. Alferes, M. Berndtsson, F. Bry, M. Eckert, N. Henze, W. May, P. L. Pătrânjan, and M. Schroeder. Use-cases on evolution. Technical Report IST506779/Lisbon/I5-D2/D/PU/a1, REWERSE, 2005.
2. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action Language for XML. In *Int. WWW Conference*, 2002.
3. M. Bernauer, G. Kappel, and G.Kramler. Composite Events for XML. In *13th Int. Conf. on World Wide Web (WWW 2004)*. ACM, 2004.
4. Harold Boley, Benjamin Grosof, Michael Sintek, Said Tabet, and Gerd Wagner. *RuleML Design*. RuleML Initiative, http://www.ruleml.org/, 2002.
5. Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *"Intl. Conference on Data Engineering (ICDE)"*, pages 403–418, 2002.
6. Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *WWW'01*, pages 633–641, 2001.

7. A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.
8. F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th ACM Symp. Applied Computing.* ACM, 2005.
9. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *20th VLDB*, 1994.
10. Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14:1–26, 1994.
11. Document object model (DOM). `http://www.w3.org/DOM/`, 1998.
12. Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification. Technical Report SC00061G, http://www.fipa.org, Dec. 2002.
13. Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification.* OMG, 2003. http://www.omg.org/cgi-bin/doc?ptc/2003-10-04.
14. Object Management Group. *XML Metadata Interchange (XMI) 2.0 Specification.* OMG, 2003. http://www.omg.org/cgi-bin/doc?formal/2003-05-02.
15. Object Management Group. *OMG Unified Modelling Language (UML) 2.0 Superstructure.* OMG, 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.
16. Mengchi Liu, Li Lu, and Guoren Wang. A Declarative XML-RL Update Language. In *Proc. Int. Conf. on Conceptual Modeling*, pages 506–519. Springer, 2003.
17. Wolfgang May. XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming*, 4(3), 2004.
18. Wolfgang May, José Júlio Alferes, and François Bry. Towards generic query, update, and event languages for the Semantic Web. In *PPSWR'04*. Springer, 2004.
19. George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Rule Languages for RDF. In *HDMS'04*, 2004.
20. N. W. Paton, editor. *Active Rules in Database Systems.* Monographs in Computer Science. Springer, 1999.
21. S. Schaffert and F. Bry. A practical introduction to Xcerpt. In *Int. Conf. Extreme Markup Languages*, 2004.
22. Igor Tatarinov, Zachary G. Ives, Alon Halevy, and Daniel Weld. Updating XML. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 133–154, 2001.
23. G. Wagner, C. V. Damásio, and S. Lukichev. First-version rule markup languages. Technical Report IST506779/Eindhoven/I1-D3/D/PU/ab1, REWERSE, 2005.
24. Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.
25. XML:DB Initiative, http://xmldb-org.sourceforge.net/. *XUpdate - XML Update Language*, September 2000.
26. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.