SeLeNe - Self E-Learning Networks
IST-2001-39045

WP4 Deliverable 4.4

ECA Rule Languages for Active Self e-Learning Networks

G.Papamarkos, A.Poulovassilis, P.T.Wood

School of Computer Science and Information Systems, Birkbeck, University of London

Abstract

The Semantic Web is based on XML and RDF as its fundamental standards for exchanging and storing information on the World Wide Web. SeLeNe will manage RDF descriptions of learning objects and users, and will need to provide reactive functionality over such metadata. We begin this report by summarising which aspects of the user requirements identified in Deliverable 2.2 will be provided by SeLeNe's reactive functionality, and discussing why event-condition-action (ECA) rules are a natural candidate for implementing this kind of functionality. We then describe a language for ECA rules on XML and a prototype implementation of this language. We next describe a language for ECA rules operating on a graph/triple representation of RDF. We finally describe the architecture of a distributed deployment of such RDF ECA rules. Along the way we identify some technical challenges and directions of further work necessary for the two languages and for the architecture.

London, UK
Jan 7, 2004

The SeLeNe Project

Life-long learning and the knowledge economy have brought about the need to support a broad and diverse community of learners throughout their lifetimes. These learners are geographically distributed and highly heterogeneous in their educational backgrounds and learning needs. The number of learning resources available on the Web is continuously increasing, thus indicating the Web's enormous potential as a significant resource of educational material both for learners and instructors.

The SeLeNe Project aims to elaborate new educational metaphors and tools in order to facilitate the formation of learning communities that require world-wide discovery and assimilation of knowledge. To realize this vision, SeLeNe is relying on semantic metadata describing educational material. SeLeNe offers advanced services for the discovery, sharing, and collaborative creation of learning resources, facilitating a syndicated and personalised access to such resources. These resources may be seen as the modern equivalent of textbooks, comprising rich composition structures, 'how to read' prerequisite paths, subject indices, and detailed learning objectives.

The SeLeNe Project (IST-2001-39045) is a one-year Accompanying Measure funded by EU FP5, running from 1st November 2002 to 31st October 2003. The project falls into action line V.1.9 CPA9 of the IST 2002 Work Programme, and is contributing to the objectives of Information and Knowledge Grids by allowing access to widespread information and knowledge, with e-Learning as the test-bed application. The project is conducting a feasibility study of using Semantic Web technology for syndicating knowledge-intensive resources (such as learning objects) and for creating personalized views over such a Knowledge Grid.

Executive Summary

This deliverable (4.4) is part of the SeLeNe Workpackage 4, on Syndication and Personalisation of Educational Resources. Workpackage 4 has two main objectives:

- To investigate techniques for syndication and personalization of distributed, autonomous RDF description bases.
- To design language primitives for defining user views over distributed RDF description bases and for deriving composite learning objects' descriptions from those of their constituent learning objects.

Accessing RDF description bases in SeLeNe raises two basic technical challenges: (1) flexible mediation of the different RDF schemas employed by the RDF description bases, and (2) personalization of learning objects' descriptions and schemas according to the educational needs and interests of learning objects' providers (i.e., instructors) and consumers (i.e., learners).

Concerning problem (1), the IEEE LOM has effectively achieved the integration of the various educational metadata standards, as is reported in Deliverable 2.1 of Workpackage 2. Thus, in the context of a SeLeNe, we are assuming that metadata about learning

objects are represented using the RDF/S binding of the IEEE LOM. However, fine-grained descriptions expressed in domain or topic-specific taxonomies may also be made available by instructors. Hence, this workpackage is investigating a flexible articulation of different domain/topic-specific taxonomies which can be used for e-learning, as well as the automatic generation of semantic descriptions for composite learning objects using the descriptions of their constituent learning objects. The taxonomies used for this purpose and the resulting descriptions can easily represented RDF/S. This work is reported in Deliverable 4.1.

Concerning problem (2) above, two major issues are involved: (a) specification by learners of their educational needs, and (b) adaptation of learning objects to these needs. Issue (a) requires the representation of educational needs in a "learner profile" using the e-learning schemas as well as the domain or topic-specific taxonomies available in SeLeNe. It also requires unstructured, keyword-based querying facilities, which can be translated automatically into the structured RDF/S queries supported by the SeLeNe system. The result of these unstructured queries may be returned in a special form of composite learning object called "trails". To address issue (b) we need methods for dynamically adapting learning material to the preferences of a learner. This requires ranking of query results by matching the descriptions of the returned learning objects against the a learner's profile. These issues are discussed in Deliverable 4.2.

Specifying educational needs or describing educational material according to personalized e-learning RDF/S schemas (for both learners and instructors) requires formalisms for defining declarative views over learning object descriptions and schemas, and this work is reported in Deliverable 4.3. This deliverable also discusses the structured RDF/S querying facilities supported by SeLeNe.

Also needed are techniques for detecting changes in learning objects' descriptions or users' personal profiles, and for notifying users who have subscribed to be notified of such changes. Techniques for the provision of this kind of reactive functionality over RDF descriptions of learning objects and users are reported in Deliverable 4.4.

We begin this Deliverable 4.4 by summarising which aspects of the user requirements identified in Deliverable 2.2 will be provided by SeLeNe's reactive functionality and discussing why event-condition-action (ECA) rules are a natural candidate for implementing this kind of functionality. We then describe a language for ECA rules on XML and a prototype implementation of this language. We next describe a language for ECA rules operating on a graph/triple representation of RDF. We finally describe the architecture of a distributed deployment of such RDF ECA rules. Along the way we identify some technical challenges and directions of further work necessary for the two languages and for the architecture.

Revision Information

Revision Date	Version	Changes
July 25, 2003	0.1	First Draft Proposal
August 15, 2003	0.2	Second Draft Proposal - Minor amendments
October 31, 2003	1.0	Final Version - Final RDF ECA Language. New examples. Appendices A and B.
Jan 7, 2004	1.1	Minor amendments
Jan 31, 2004	1.2	Minor amendments

Table of Contents

1 Providing SeLeNe's Reactive Functionality

Peers in a SeLeNe network will store RDF/S descriptions relating to learning objects (LOs) registered with the SeLeNe and users of the SeLeNe. Peers may also store system-related RDF/S descriptions. In a centralised environment, there will be just one 'peer' server which will manage all of the RDF/S descriptions. In a distributed environment, each peer will manage some fragment of the overall RDF/S descriptions.

SeLeNe's reactive functionality will provide the following aspects of the user requirements discussed in [15][1]:

- automatic notification to users of the registration of new LOs of interest to them;
- automatic notification to users of the registration of new users who have information in common with them in their personal profile;
- automatic notification to users of changes in the description of resources of interest to them;
- automatic propagation of changes in the description of one resource to the descriptions of other, related resources, including:
  - propagating changes in the taxonomical description of a LO to the taxonomical description of any composite LOs depending on it;
  - propagating changes in a learner's history of searches/accesses of LOs to the learner's personal profile;
  - propagating changes in a learner's level of knowledge, learning experience, or learning goals to the set of recommended trails for that learner;
  - similarly updating a group's set of recommended trails depending on individual or collective changes to the group's metadata;
  - automatic generation and update of 'emergent' trails.

Event-condition-action (ECA) rules automatically perform actions in response to events provided that stated conditions hold. There are several advantages in using ECA rules to implement a system's reactive functionality, rather than implementing it directly in application code. Firstly, ECA rules allow this functionality to be specified and managed within a single rule base rather than being encoded in diverse programs, thus enhancing the system's modularity, maintainability and extensibility. Secondly, ECA rules have a high-level, declarative syntax and are thus amenable to analysis and optimisation techniques which cannot be applied if the same functionality is expressed directly in application code. Thirdly, ECA rules are a generic mechanism that can abstract a wide variety of reactive

[1]The term 'reactive' refers to a system's ability to detect the occurrence of specific events or changes in information content, and to respond by automatically executing the appropriate application logic.

behaviours, in contrast to application code that is typically specialised to a particular kind of reactive scenario.

Motivated by these advantages of ECA rules, we provide SeLeNe's reactive functionality by means of ECA rules over RDF descriptions. ECA rules have been used in many settings, including active databases [23, 18], personalisation and publish/subscribe technology [3, 8, 9, 11, 19], and specifying and implementing business processes [2, 10, 13]. An ECA rule has the general syntax

`on` *event* `if` *condition* `do` *actions*

The event part specifies when the rule should be triggered, the condition part is a query which determines if the system is in particular state, and the action part states the actions to be performed automatically if the condition holds. Executing a rule's actions may in turn trigger further ECA rules, and the rule execution proceeds until no more rules are triggered.

The rest of this report is as follows. We begin with a brief review of previous work on ECA rules for XML, since one possibility in SeLeNe is that its RDF descriptions will be serialised as XML. We describe a language for specifying ECA rules on XML repositories, and present a prototype implementation of it. We are also exploring ECA rule languages for RDF that will operate on a graph/triple representation, as this is also a possibility in SeLeNe, and we present such a language. Finally, we present an architecture for distributed deployment of XML or RDF ECA rules. Along the way, we discuss directions of further work for both languages and for the architecture.

2 ECA Rules for XML

2.1 Previous work in this area

In recent work [6, 5], we specified a language for defining ECA rules on XML data, based on the XPath and XQuery standards. We also developed techniques for analysing the triggering and activation relationships between such rules[2] and showed how these techniques can be used to detect possibly non-terminating sets of ECA rules. A number of other ECA rule languages for XML have also been proposed, although none of this other work has focussed on analysing the behaviour of the ECA rules.

Reference [8] discusses extending XML repositories with ECA rules in order to support e-services. Active extensions to the XSLT [25] and Lorel [1] languages are proposed which handle insertion, deletion, and update events on XML documents. Reference [9] discusses a more specific application of the approach to push technology where rule actions are methods that cannot update the repository, and hence cannot trigger other rules.

Reference [7] also defines an ECA rule language for XML. The rule syntax is similar to

[2]A rule r_i *may trigger* a rule r_j if execution of the action of r_i may generate an event which triggers r_j. A rule r_i *may activate* another rule r_j if r_j's condition may be changed from False to True after the execution of r_i's action. A rule r_i *may activate* itself if its condition may be True after the execution of its action.

ours, but the rule execution model is different. In our case we treat insertions or deletions of XML fragments as "atomic" and rules are triggered only after the completion of such an update. In contrast, in [7] insertions or deletions of XML fragments are broken up into a sequence of finer granularity update requests, and triggering of rules is interleaved with execution of these smaller updates. In general, these semantics may produce different results for the same initial update and it is a question of future research to determine their respective performance trade-offs and suitability in different application situations.

ARML [12] provides an XML-based rule description for rule sharing among different heterogeneous ECA rule processing systems. In contrast to our language, conditions and actions are defined abstractly as XML-RPC methods which are later matched with system-specific methods.

GRML [22] is a multi-purpose rule markup language for defining integrity, derivation and ECA rules. GRML uses an abstract syntax based on RuleML, leaving the mapping to a real language for each underlying system implementation. GRML aims to provide semantics for defining access over distributed, heterogeneous data sources for rule evaluation and allows the user to declare most of the semantics necessary for processing a rule, and to evaluate events and conditions coming from heterogeneous data sources.

Other related work is [21] which proposes extensions to the XQuery language [26] to incorporate update operations. These are more expressive than the actions supported by our ECA rule language since they also include renaming and replacement operations, and specification of updates at multiple levels of documents. Triggers are discussed in [21] as an implementation mechanism for deletion operations on the underlying relational store of the XML. However, provision of ECA rules at the "logical" XML level is not considered.

2.2 Our XML ECA Language

An XML repository consists of a set of XML documents. In our XML ECA rule language, we use XPath [24] and XQuery [26] to specify the event, condition and actions parts of rules. XPath is used for selecting and matching fragments of XML documents within the event and condition parts while XQuery is used within insertion actions, where there is a need to be able to construct new XML fragments.

The event part of an XML ECA rule is an expression of one of the following two forms:

`INSERT` e

`DELETE` e

where e is an XPath expression which evaluates to a set of nodes. The rule is *triggered* if this set of nodes includes any node in a new XML fragment, in the case of an insertion, or in a deleted fragment, in the case of a deletion.

The system-defined variable `$delta` is available for use within the condition and actions parts of the rule, and its set of instantiations is the set of new or deleted nodes returned by e.

The condition part of a rule is either the constant `TRUE`, or one or more XPath expressions connected by the boolean connectives `and`, `or`, `not`. The rule *fires* if it is triggered and its condition evaluates to true.

The actions part of a rule is a sequence of one or more actions:

$$action_1;\ \ldots;\ action_n$$

where each $action_i$ is an expression of one of the following three forms:

INSERT r BELOW e BEFORE q

INSERT r BELOW e AFTER q

DELETE e

Here, r is an XQuery expression, e is an XPath expression and q is either the constant `TRUE` or an XPath qualifier.

In an `INSERT` action, the expression e specifies the set of nodes, N, immediately below which new XML fragment(s) will be inserted. These fragments are specified by the expression r. If e or r references the `$delta` variable, then one XML fragment is constructed for each instantiation of `$delta` for which the rule's condition evaluates to True. If neither e nor r references `$delta`, then a single fragment is constructed. The expression q is an XPath qualifier which is evaluated on each child of each node $n \in N$. For insertions of the form `AFTER` q, the new fragment(s) are inserted after the last sibling for which q is True, while for insertions of the form `BEFORE` q, the new fragment(s) are inserted before the first sibling for which q is True. The order in which new fragments are inserted is non-deterministic.

In a `DELETE` action, the expression e specifies the set of nodes which will be deleted (together with their descendant nodes). Again, e may reference the `$delta` variable.

Example 1 Consider an XML repository containing metadata about LOs available on the web, as well as personal metadata about users of these LOs. The XML document `los.xml` contains information about the LOs, and we show below some of the information held for a particular book, "Data On the Web". Under `annotations`, a new `review` is appended every time a user submits a review of the book.

```
<LOs>
  ..
  <LO type="book" title="Data On the Web">
    <subject>Computer Science</subject>
    <creator>S. Abiteboul</creator>
    <creator>P. Buneman</creator>
    <creator>D. Suciu</creator>
    <description>From Relations to Semistructured data and XML
    </description>
    <publisher>Morgan Kaufmann</publisher>
    <isbn>1-55860-621-Y</isbn>
    <annotations>
      <review>
        <reviewer>Teacher Education Review Panel</reviewer>
        <date>2002-10-20</date>
        <rating>9</rating>
        <description>
          This book gives a comprehensive, state-of-the art
          discussion of data models, query languages and ...
        </description>
      </review>
      <review>
        <reviewer>John Smith</reviewer>
        <date>2002-12-20</date>
        <rating>10</rating>
        <description>
          I found this a great book to learn about querying
          semi-structured data, which I didn't know much about.
        </description>
      </review>
    </annotations>
  </LO>
  ...
</LOs>
```

The XML document `users.xml` contains information about users, and we show below some of the information held for a particular user “Johnny Mnemonic”. Users can subscribe to be notified of the latest review submitted for books in subjects that they are interested in, and this information is used to automatically update their personal metadata:

```
<users>
  ...
  <user id="217">
    <name>Johnny Mnemonic</name>
    <profession>student</profession>
    <subjects>
      <subject>Computer Science</subject>
      <subject>Mathematics</subject>
      <subject>Economics</subject>
    </subjects>
    <LOs>
      <LO type="book" title="Data On the Web">
        <isbn>1-55860-621-Y</isbn>
        <latest-review>
          <reviewer>John Smith</reviewer>
          <date>2002-12-20</date>
          <rating>10</rating>
          <description>
            I found this a great book to learn about querying
           semi-structured data, which I didn’t know much about.
           </description>
         </latest-review>
       </LO>
       ...
     </LOs>
   </user>
   ...
 </users>
```

Johnny Mnemonic is interested in “Computer Science” and the following rule replaces the current latest review (if there is one) of any Computer Science book in his personal metadata by a new review of that book:

```
ON INSERT document('los.xml')/LOs/LO/annotations/review
IF $delta/../../subject[.='Computer Science']
DO DELETE document('users.xml')/users/user[@id="217"]/LOs/LO
          [isbn=$delta/../../isbn]/latest-review;
   INSERT <latest-review>{$delta/*}</latest-review>
   BELOW  document('users.xml')/users/user[@id="217"]/LOs/
          LO[isbn=$delta/../../isbn]
   AFTER  isbn
```

Here, the system-defined `$delta` variable is bound to a newly inserted `review` node detected by the event part of the rule. The rule’s condition checks that the subject of the book in question is Computer Science. The rule’s action then deletes the existing latest review for this book within Johnny Mnemonic’s metadata (if there is one) and inserts the new review.

Suppose now that the following update occurs, appending a new review for the “Data On the Web” book:

```
INSERT <review>
         <reviewer>Neo Anderson</reviewer>
         <date>2003-04-29</date>
         <rating>9</rating>
         <description>
           Very clearly written and very well-organised.
           Describes in detail all the ...
         </description>
       </review>
BELOW document('los.xml')/LOs/LO[isbn="1-55860-621-Y"]/annotations
AFTER TRUE
```

This update triggers the rule above, causing the replacement within Johnny Mnemonic’s personal metadata of the previous review submitted by John Smith by the new review submitted by Neo Anderson.

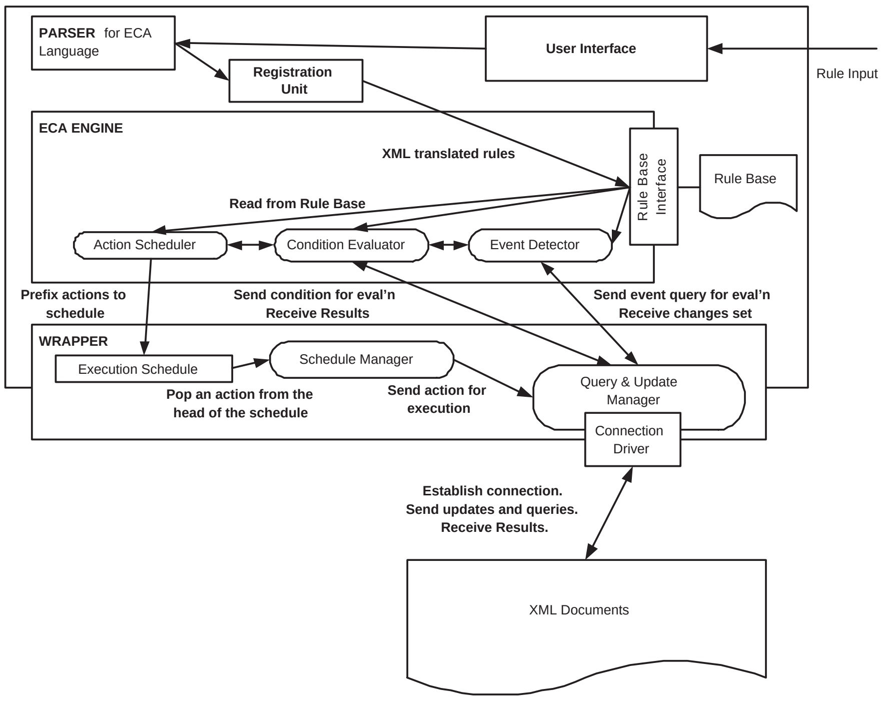

Figure 1: ECA Engine Architecture

As another example rule, the following rule removes the current latest review (if there is one) of a Computer Science book in Johnny Mnemonic's personal metadata if this review is removed from the list of reviews for this book (this rule assumes that each reviewer reviews a book only once):

```
ON DELETE document('los.xml')/LOs/LO/annotations/review
IF $delta/../../subject[.='Computer Science']
DO DELETE document('users.xml')/users/user[@id="217"]/LOs/LO
          [isbn=$delta/../../isbn]/latest-review[reviewer=$delta/reviewer]
```

We refer the reader to [6, 5] for a more detailed discussion of the syntax and semantics of our XML ECA rule language. Here, we next describe a prototype implementation.

2.3 A Prototype Implementation

For this first prototype implementation we have used flat files and have exploited the functionality provided by the W3C DOM standard [27] for interacting with them. The architecture of our system is illustrated in Figure 1.

The *Parser* parses and checks the syntactic validity of a new rule. For construction of the parser, we have used the JavaCC lexer-parser generator. Valid rules are translated into an XML form and are added by the *Registration Unit* to the *Rule Base* (which is an XML file). Details about each rule are stored here, including its name, priority, and event, condition and action parts.

The *ECA Engine* encapsulates the rule processing functionality. In particular, the *Event Detector*, *Condition Evaluator* and *Action Scheduler* implement these aspects of the rule processing, as we describe in more detail below. All of these components interface with the Wrapper in order to send and receive data to and from the underlying XML files.

The *Execution Schedule* contains a sequence of *updates* — these have the same syntax as rule actions except that they do not contain any $delta expressions within them. By "$delta expression" we mean an XPath expression (either stand-alone or possibly nested within another XPath expression) that starts with `$delta`. These portions of a rule's action part are replaced by the result of evaluating the expressions on the current document — see below.

The *Wrapper* interfaces with the XML files on disk. All update and query requests from the upper levels of the system pass through this component, which coordinates them. It undertakes to open files, submit queries and updates, and receive back results from them. The Wrapper performs these services by using the functionality of the Apache Xalan API. All queries are performed directly by using XPath. For deletions, we identify the set of nodes that will be deleted by using the XPath expression within the `DELETE` part of the request, and we then remove all the subdocuments rooted at the nodes identified. For insertions, we identify the set of nodes that will be affected by using the XPath expression within the `BELOW` part of the request, and we then add the fragment specified within the `INSERT` part as a new child of each of the nodes identified, placing it relative to the existing children according to the `AFTER` or `BEFORE` qualifier.

Rule execution begins with a request from the *Schedule Manager* to the *Query & Update Manager* to execute the update currently at the head of the schedule. In case of an insertion, the Query & Update Manager executes the update and annotates the newly inserted nodes, while in the case of a deletion it annotates the nodes to be deleted without executing the deletion yet[3].

Following the execution of the update, control then passes to the *Event Detector*. This requests the Query & Update Manager to evaluate the XPath query of the event part of each rule that may be triggered by the update that was just executed. For each rule whose query result set contains annotated nodes (either newly inserted or about-to-be deleted), the Event Detector creates a `changes` set containing these annotated nodes, and the rule is triggered.

The *Condition Evaluator* then requests the Query & Update Manager to evaluate the condition part of each triggered rule on the affected document, using as the evaluation context either the root node if there are no occurrences of `$delta` within a query, or each instance of the `changes` set otherwise. The rule's `delta` set is thus created, consisting of those members of its `changes` set for which the condition evaluates to true. If the `delta` set is non-empty, the rule fires and control is passed to the Action Scheduler to further process the rule. Otherwise, processing of this rule ends.

The *Action Scheduler* reformulates a given rule's action(s) in order to eliminate any

[3]The annotation of nodes is performed using non-DOM methods provided by Apache Xerces API that allow us to attach data to XML nodes without affecting the physical representation of the file.

instances of `$delta` expressions within them. The reformulation algorithm performs the following steps for each node within the rule's `delta` set:

- Replaces the `$delta` variable in each of the $delta expressions by the current node of the `delta` set.
- Evaluates each of the modified $delta expressions with respect to the updated document.
- Replaces each $delta expression within the rule's action(s) by the corresponding result of the previous step.

The outcome of this reformulation is that one instance of the rule's action(s) is created for each node in the rule's `delta` set. These updates are now prefixed, in an arbitrary order, to the front of the schedule — this is known as *Immediate* scheduling, although other alternatives are also possible (see [18]). If multiple rules have fired as a result of the last update executed, then the updates that result from their actions are prefixed the schedule in order of the rules' specified priorities. Control then passes once more to the Schedule Manager and the cycle repeats. If the last update executed by the Query & Update Manager was a `DELETE`, then before control passes back to the Schedule Manager, the actual deletion of the annotated nodes is first performed.

2.4 Challenges and Future Work

There is as yet no accepted standard update language for XML. If ECA rules are to be supported on XML repositories, then whatever standard eventually emerges, there is also the parallel issue of designing the event language to match up with this update language.

In this report we have seen how this was done in the context of our particular update language for XML. Elsewhere [5] it is shown how triggering and activation relationships can be detected for our particular XML ECA rules. In general, the ability to analyse and optimise ECA rules needs to be balanced against their complexity and expressiveness, and this issue also needs to be borne in mind in future developments in ECA rule languages for XML, and indeed for RDF.

It would be straightforward to extend our language to also support `REPLACE` events and actions. A `REPLACE` event would have the syntax

`REPLACE` e

while a `REPLACE` action would have the syntax

`REPLACE` e `BY` r

where e is an XPath expression and r is an XQuery expression. The meaning of a `REPLACE` action is that the set of nodes identified by e (and their descendants) should be replaced by the XML fragments denoted by r. For example, the pair of actions in the first rule in Example 1 could be replaced by the single action

```
REPLACE document('users.xml')/users/user[@id="217"]/LOs/LO
        [isbn=$delta/../../isbn]/latest-review
```

```
BY       <latest-review>{$delta/*}</latest-review>
```

In general, our `INSERT` actions may result in non-determinism in the order in which a set of new fragments are inserted under a common parent, since the `BEFORE` and `AFTER` constructs only specify the ordering of new fragments with respect to the existing document content. It is an area of further work to extend our XML ECA language to capture ordering relationships between new fragments being inserted into a document.

At present we assume Immediate scheduling of rules that have fired, though it would be straightforward to also allow rules with other scheduling modes. However, the practical applicability and performance implications of these extensions is an area that requires further investigation.

Another important area is combining ECA rules with transactions and consistency maintenance in XML repositories.

3 ECA rules for RDF

The above language can be used for RDF which has been serialised as XML. However, we are also exploring ECA rule languages for RDF that will operate directly on a graph/triple representation. We described an archetypal such language in [17]. Since the publication of that paper, our language has evolved to match more closely the update facilities provided by FORTH's RDFSuite [4], which is ultimately likely to be SeLeNe's RDF repository.

3.1 Our RDF ECA Rule Language

In our RDF ECA rule language, the event part of a rule is an expression of one of the following three forms:

1. `[`*`let-expressions`*` IN] (INSERT | DELETE)` e `[AS INSTANCE OF `*`class`*`] [USING NAMESPACE `*`nspace`*`]`

   This detects insertions or deletions of resources described by the expression e. e is a path expression which evaluates to a set of nodes. Optionally, *class* is the name of the RDF Schema class to which at least one of the nodes identified by e must belong in order for the rule to trigger. To ensure uniqueness and be more specific on the resources that trigger the rule we can also optionally specify the *namespace* they belong to.

   let-expressions is an optional set of local variable definitions of the form `let` *variable* `:=` e', where e' is a path expression.

   The rule is triggered if the set of nodes returned by e includes any new node (in the case of an insertion) or any deleted node (in the case of a deletion) that is an instance of the *class*, if specified. The system-defined variable `$delta` is again available for use within the condition and actions parts of the rule, and its set of instantiations is the set of new or deleted nodes that have triggered the rule.

2. `[`*`let-expressions`*` IN] (INSERT | DELETE) `*`triple`*

   This detects insertions or deletions of arcs specified by *triple*, which has the form *(source_node, arc_name, target_node)*. The wildcard '_' is allowed in the place of any of a triple's components.

   The rule is triggered if an arc labelled *arc_name* from *source_node* to *target_node* is inserted/deleted. The variable `$delta` has as its set of instantiations the values of *source_node* of the arc(s) which have triggered the rule.

3. `[`*`let-expressions`*` IN] UPDATE `*`upd_triple`*

   This detects updates of arcs. *upd_triple* is a special form of triple. Its form is *(source_node, arc_name, old_target_node* $\rightarrow$ *new_target_node)*. Here, *old_target_node* is where the arc labelled *arc_name* from *source_node* used to point before the update, and *new_target_node* is where this arc points after the update. Again, the wildcard '_' is allowed in the place of any of these components.

   The rule is triggered if an arc labelled *arc_name* from *source_node* changes its target from *old_target_node* to *new_target_node*. The variable `$delta` has as its set of instantiations the values of *source_node* of the arc(s) which have triggered the rule.

The condition part of a rule is a query which may reference the `$delta` variable. Analogously to our XML ECA rule language, condition queries consist of conjunctions, disjunctions and negations of path expressions.

The actions part of a rule is a sequence of one or more actions. Actions can `INSERT` or `DELETE` a resource — specified by its URI — and `INSERT`, `DELETE` or `UPDATE` an arc. The actions language has the following form for each one of these cases (note that this actions language can also serve more generally as an update language for RDF):

1. `[`*`let-expressions`*` IN] INSERT `*`e`*` AS INSTANCE OF `*`class`*
   `[USING NAMESPACE `*`nspace`*`]`

   `[`*`let-expressions`*` IN] DELETE `*`e`*` [AS INSTANCE OF `*`class`*`]`
   `[USING NAMESPACE `*`nspace`*`]`

   for expressing insertion and deletion of a resource.

2. `[`*`let-expressions`*` IN] (INSERT | DELETE) `*`triple`*` (',' `*`triple`*`)*`

   for expressing insertion or deletion of the arcs(s) specified.

3. `[`*`let-expressions`*` IN] UPDATE `*`upd_triple`*` (',' `*`upd_triple`*`)*`

   for updating arc(s) by changing their target node.

The `AS INSTANCE OF` keyword classifies, according to the RDF Schema, the resource to be deleted or inserted. In the case of insertions, the classification of the new resource is obligatory, while in the case of deletions it is optional. Specification of the namespace where the resource belongs, using the `USING NAMESPACE nspace` construct, is also optional.

The triples in the case of arc manipulation have the same form as in the event sub-language. In the case of arc insertion and deletion they have the form *(source_node, arc_name, target_node)* while in the case of arc update, the old and the new target node are also specified, so that the triple has the form *(source_node, arc_name, old_target_node* $\rightarrow$ *new_target_node).*

The wildcard '_' may also appear inside triples in the action sub-language, as follows: In the case of a new arc insertion, '_' is allowed in the place of the *source_node* and has the effect of inserting the new arc for all stored resources. In the case of arc deletion, if '_' replaces the *arc_name* then all the arcs from *source_node* pointing to *target_node* will be deleted; if '_' replaces the *source_node*, the action deletes all the arcs labelled *arc_name*; replacing the *target_node* by '_' deletes the arc *arc_name* from the *source_node* regardless of where it points to. In case of a arc update, '_' can be used in place of the *source_node* or the *old_target_node*; in the first case, it indicates replacement of the target node for all arcs labelled *arc_name*; in the second case, use of '_' indicates update of the target node regardless of its previous value. The use of combinations of the above wildcards in a triple is also allowed, in order to express more complex update semantics that combine those given above.

Examples. These examples refer to the LO metadata illustrated in Figure 2 and to the fragment of a user's personal metadata illustrated in Figure 3. In Figure 3, `ext1` is the IMS-LIP schema and `ext3` the SeLeNe User Profile schema (see [14]).

Suppose that user 128 wants to be notified whenever that a new user registers with the system who has an area of interest in common with user 128. When such a new user registers, the following ECA rule adds a new arc linking the newly registered user into the `new_users` collection in user 128's personal messages:

```
ON INSERT resource() AS INSTANCE OF Learner USING NAMESPACE ext3
IF $delta/target(ext1:interest)/element()/target(ext1:interest_typename)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(ext1:interest)/element()/target(ext1:interest_typename)
DO LET $new_users := resource(http://www.dcs.bbk.ac.uk/users/128)
                     /target(ext3:messages)/target(ext3:new_users) IN
   INSERT ($new_users,seq++,$delta);;
```

Here, the event part of the rule checks whether a new resource belonging to the `Learner` class in namespace `ext3` has been added. The condition part checks if the new user has an area of interest in common with user 128. If so, the action part inserts a new arc between the `new_users` collection in user 128's personal messages and the resource representing the new user (we use the syntax `seq++` to indicate an increment in the collection's element count).

As another example, if a LO is inserted whose subject is the same as one of user 128's areas of interest, then the following rule adds a new arc linking the newly inserted LO into the `new_LOs` collection in user 128's personal messages:

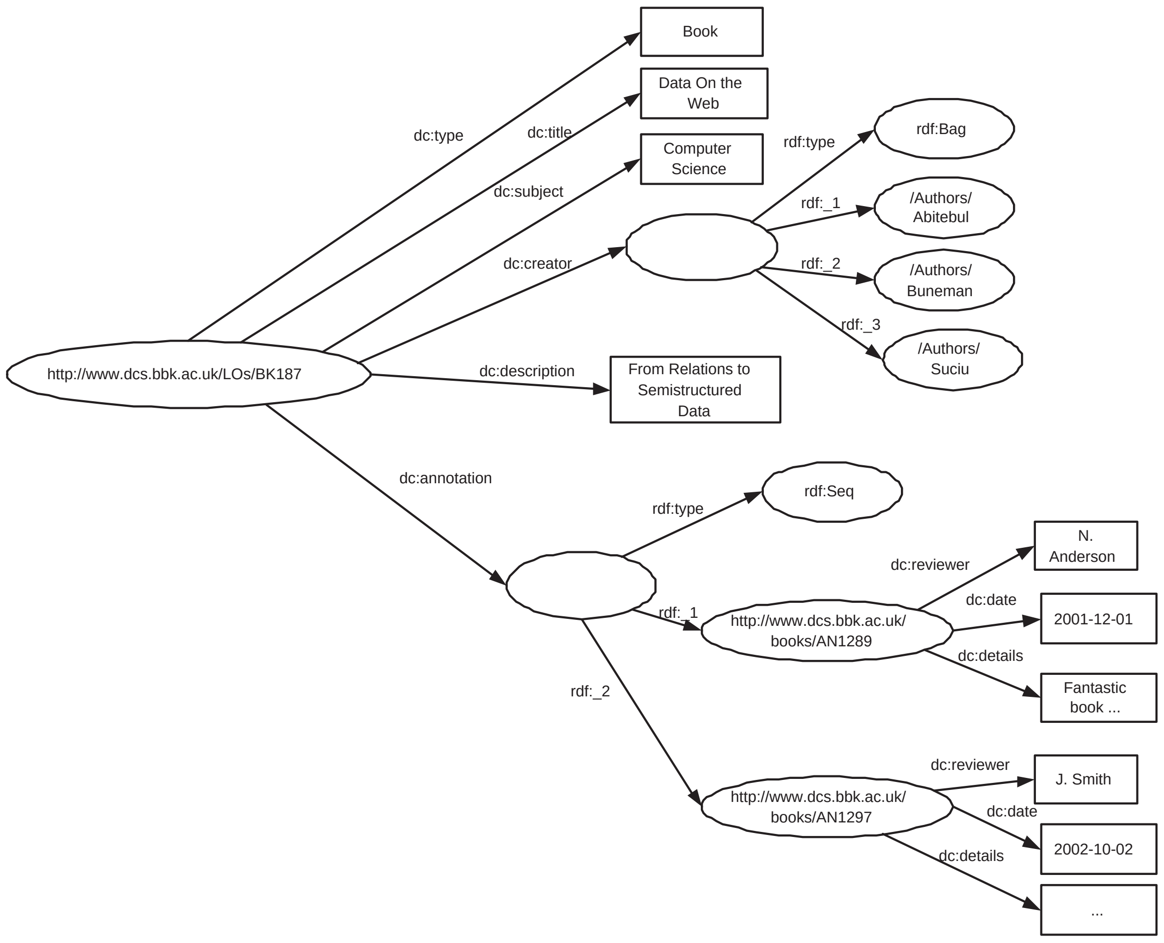

Figure 2: Example of LO Metadata

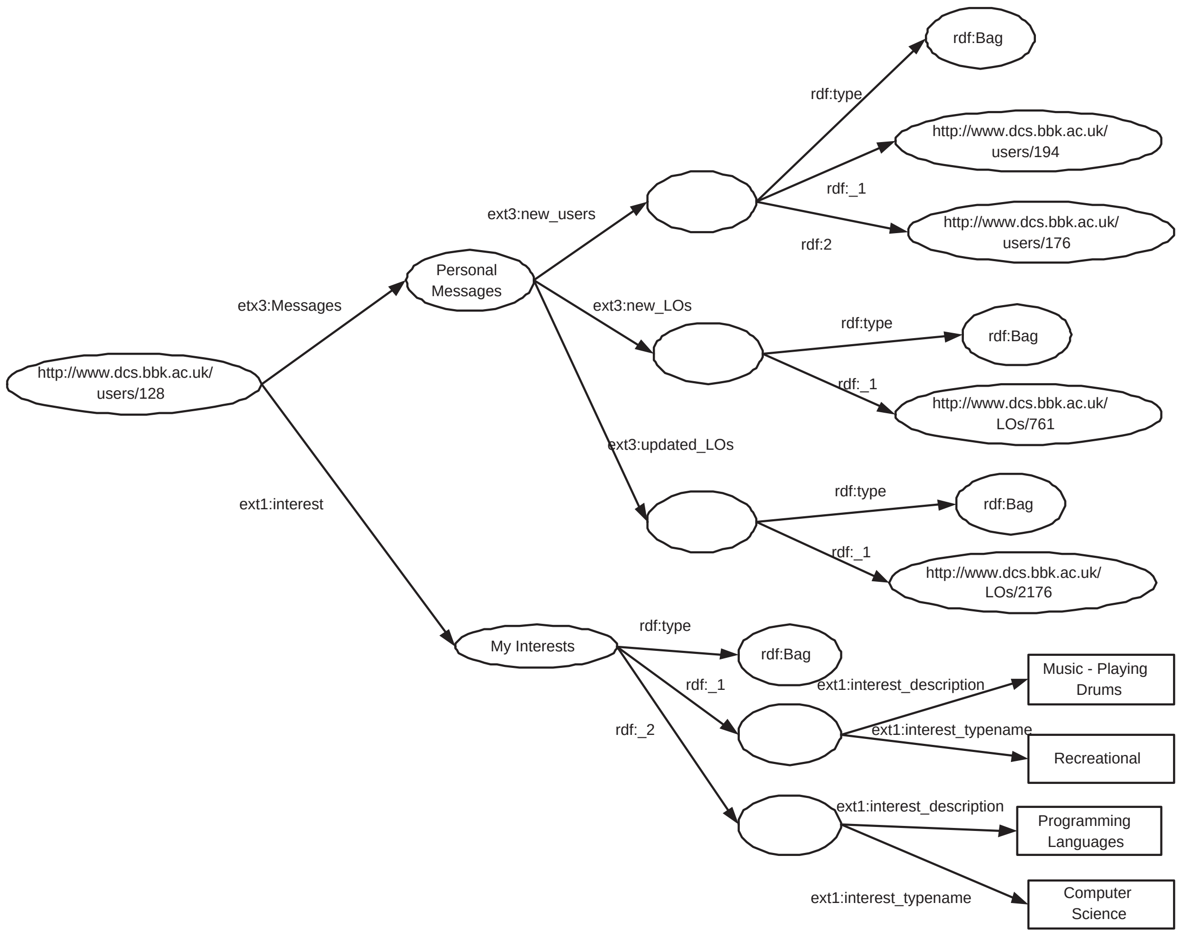

Figure 3: Example of User Metadata

```
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(ext1:interest)/element()/target(ext1:interest_typename)
DO LET $new_los := resource(http://www.dcs.bbk.ac.uk/users/128)
                   /target(ext3:messages)/target(ext3:new_LOs) IN
   INSERT ($new_los,seq++,$delta);;
```

Here, the event part checks if a new resource belonging to the `LO` class has been inserted. The condition part checks if the inserted LO has a subject which is the same as of one user 128's areas of interest. The `LET` clause in the rule's action defines the variable `$new_los` to be user 128's new LOs collection. Finally, the `INSERT` clause inserts a new arc from `$new_los` to the newly inserted LO.

As a third example, if the description of a LO whose subject is the same as one of user 128's areas of interest changes, then a new arc is inserted from user 128's `updated_LOs` collection to the modified LO:

```
ON UPDATE (resource(),dc:description,_->_)
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(ext1:interest)/element()/target(ext1:interest_typename)
DO LET $updated_lo_list := resource(http://www.dcs.bbk.ac.uk/users/128)
          /target(ext3:messages)/target(ext3:updated_LOs) IN
   INSERT ($updated_lo_list,seq++,$delta);;
```

3.2 Challenges and Future Work

There is as yet no standard query/update language for RDF and hence our RDF ECA language is even more prototypical than our XML ECA language. Some of the observations we made in Section 2.4 regarding the XML ECA language also apply here, namely the need to match up the event sub-language with the update sub-language, the need to balance expressiveness of ECA rules against the ability to analyze and optimize them, the possibility of a variety of scheduling modes beyond Immediate rule scheduling, and combining ECA rules with transactions and consistency maintenance in RDF repositories.

For the immediate future, we plan to

- explore more deeply the expressiveness our RDF ECA language (Appendix B shows that it is computationally complete, but we plan to investigate its query and update expressiveness as well);
- implement the language over FORTH's RDFSuite [4]; and
- experiment with the language using as a testbed SeLeNe's learning object and user metadata.

4 ECA Rules in a Distributed Environment

4.1 A Distributed ECA Architecture

The architecture described in Section 2.3 for our XML ECA rules is applicable also for centralised management and processing of our RDF ECA rules, with suitable modifications to particular components to handle RDF rather than XML rules and data. One ECA Engine is installed on the central server hosting the RDF/S repository. Any update request to this repository passes through the ECA Engine, which in coordination with SeLeNe's Update service enables both event detection and update processing at the server.

Beyond the above centralised version of our system, we are investigating a distributed version supporting ECA rules on distributed RDF repositories. In a distributed environment, an ECA rule generated by one peer might be triggered, evaluated, and executed at different peers of the network. The architecture that we envisage is illustrated in Figure 4. Each 'super-peer' server shown in that diagram may be coordinating a group of further 'peer' servers, as well as itself hosting a fragment of the overall RDF/S descriptions. At each super-peer there is installed one local ECA Engine, which has the same features and components as the centralised architecture discussed in Section 2.3 and illustrated in Figure 1. One possibility is that each local ECA Engine can operate as a Web Service and that the communication between them can be via XML messages (e.g. SOAP).

Whenever a new ECA rule r is registered at a peer P, it will be sent to P's super-peer for storage. As we will see below, from there r will also be sent to all other super-peers, and a replica of it will be stored at those super-peers that are *relevant* to r i.e. where an event may occur that may trigger r's event part, or which may participate in evaluating r's condition part, or where r's actions may have to be scheduled for execution.

Within the event, condition and action parts of SeLeNe's ECA rules, there might or might not be references to specific RDF sources i.e. ECA rules may be resource-specific or generic. In this first version of our distributed ECA architecture, we assume that at run-time rules are triggered by events occuring within a single peer's local RDF repository and that individual rule actions execute within a single peer's RDF repository. In other words, there is no need for distributed event detection or distributed execution of individual actions (although the condition parts of rules may be distributed, and different actions from one rule may execute at different peers). These assumptions are justified by the nature of SeLeNe's metadata descriptions, since the description relating to a LO registered at a peer will be stored at that peer, and similarly the personal profile of a user will be stored locally at one peer. Mechanisms to ensure that ECA rules indeed conform to these restrictions is a topic requiring further investigation.

Below, we consider two possible scenarios for the way in which super-peers coordinate the information managed by their peers — *mediated* and *peer-to-peer*. In a mediated scenario, each peer manages a local RDF Schema to which its RDF descriptions conform. Each super-peer manages a global RDF Schema GS which semantically integrates the set of local RDF Schemas of its own peer group $LS_1, \ldots, LS_n$ as well as the global schemas of the other super-peers $GS_1, \ldots, GS_m$ to which it is connected — thus, in effect, each

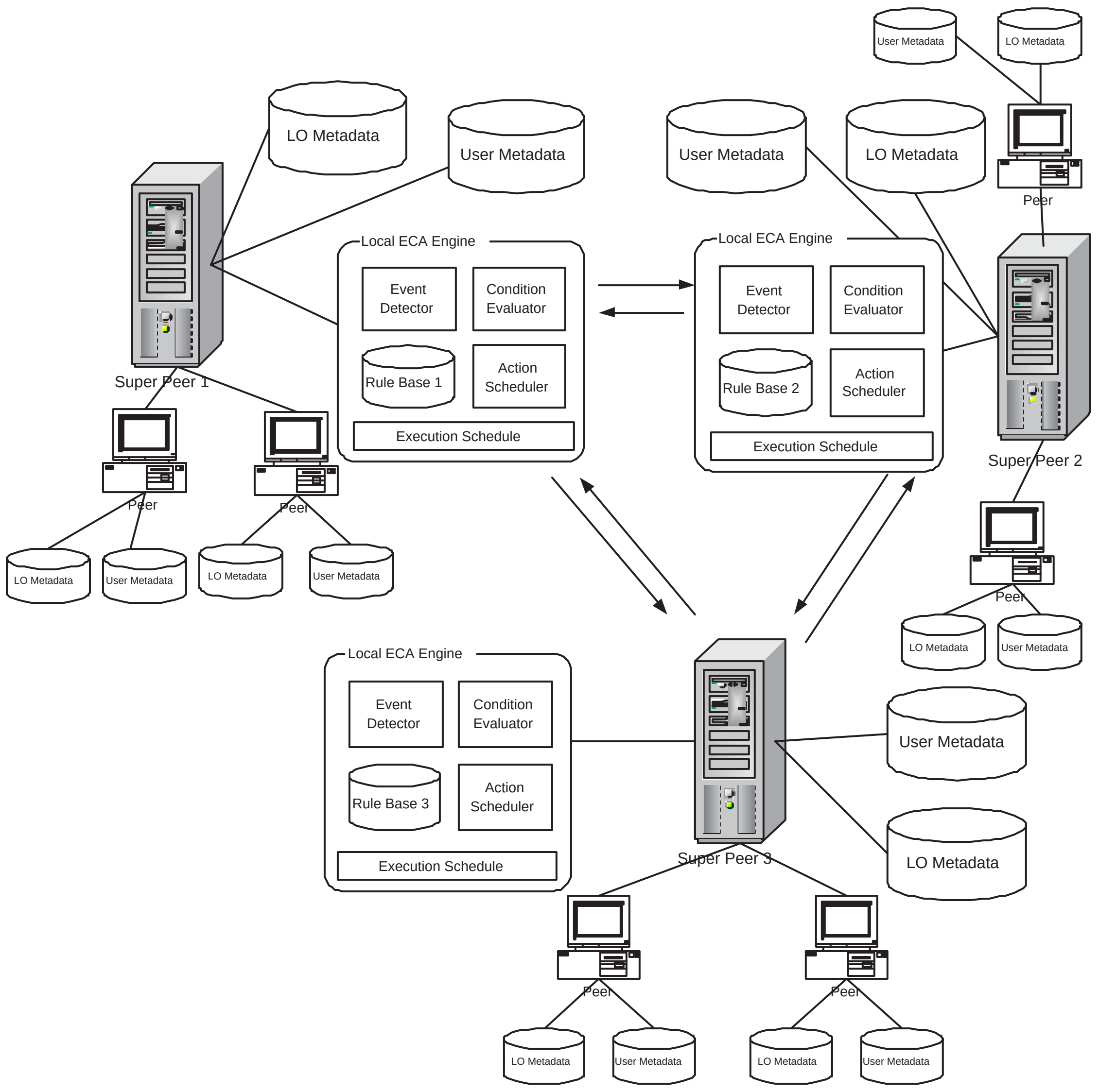

Figure 4: Distributed System Architecture

super-peer has the role of a mediator. The set of mappings between GS and $LS_1, \ldots, LS_n$, $GS_1, \ldots, GS_m$ is managed by the super-peer as part of SeLeNe's Syndication service [20]. These schemas and mappings may evolve with time, but this will be driven by changes to the local or global RDF Schemas, under the control of the overall SeLeNe system.

In a P2P scenario each peer again stores a fragment of the overall distributed RDF descriptions, but this need not conform to the global RDF Schema of its coordinating super-peer, and changes to local and global RDF Schemas may occur as a result of changes to the peers' RDF descriptions. Moreover, peers may dynamically join or leave the SeLeNe network. We discuss first this scenario, and then consider the simpler mediated scenario.

4.2 P2P Scenario

4.2.1 Indexing at Peers and Super-Peers

In order to determine the relevance of super-peers to particular ECA rules, a simple *index* can be kept at each peer and super-peer. There are a number of possibilities for doing this and we indicate here one solution:

As the RDF descriptions stored at each peer change over time, so each peer maintains an annotated copy of its local RDF Schema, which shows for each node in the schema whether or not there is RDF data of this type at this peer (a '0' or '1' bit).

This information is also propagated to the peer's coordinating super-peer. This super-peer maintains a combined RDF Schema which is annotated so that each node shows the set of peers in its own peer group that manage data of this type (a set of peer IDs), and also the remote super-peers to which it is connected and whose peer group manages such data (a set of super-peer IDs).

The latter information is gathered and maintained as follows. If a node in the RDF Schema of a super-peer, SP, changes from not having any data in this peer group to having data, or vice versa, this change is notified to SP's neighbouring super-peers, so that these too can update the relevant annotation in their RDF Schemas. If one of these RDF Schemas changes from not having data associated with a particular RDF Schema node to having such data, this change is notified to this super-peer's neighbours. The process repeats, thereby propagating the necessary changes throughout the network of super-peers.

Note that in general the super-peers may hold heterogeneous RDF Schemas, so there will need to be an RDF Schema mapping service between super-peers' RDF Schemas — as is indeed envisaged for SeLeNe, and is provided by its Syndication service.

Finally, as well as this annotated RDF Schema, each super-peer also keeps for each node annotated with a '1' in its RDF Schema a list of the RDF resources of this type that each peer in its peer group references — we call these lists of RDF resources *resource indexes*.

Comparison with related approaches: Querying and indexing data in a distributed RDF-based P2P network is more complex than for distributed structured databases. In the latter, the database servers and the database schema at each of them is known and fixed whereas in the former peers may dynamically join or leave the network and may manage data conforming to varying schema fragments. Schema-based routing indexes have been proposed to address this problem in Edutella [16]. Edutella uses two kinds of routing indexes: Super-Peer/Peer (SP/P RI) and Super-Peer/Super-Peer (SP/SP RI).

An SP/P RI stores information about metadata usage in each peer in its peer group. This includes information such as the schemas (e.g., `dc` or `lom`) or properties (e.g., `dc:subject`) used, as well as possibly conventional indexes on property values. When a peer registers with a super-peer, it provides the super-peer with its metadata usage, a process called advertisement. The peer undertakes to keep this advertisement up-to-date by informing its super-peer each time that a change affecting the advertised metadata takes place. At each super-peer, query fragments are matched against the SP/P RIs in order to determine peers that are relevant to this query (although this gives no guarantee that the returned result set from a peer is not empty). A similar approach is used in SP/SP RIs, but at a higher level of granularity and possibly only representing approximations of the information regarding their peers. A further difference to the SP/P RI is that an SP/SP RI contains information only about its neighbouring super-peers in the network. Update of SP/SP RIs is again based on broadcast messages sent between super-peers.

For our purposes, we want to maintain more precise information about where various forms of metadata reside in the network and, as far as possible, do not want unnecessary routing of queries and updates to peers and super-peers that are not relevant. Hence, we have adopted the approach of using annotations on a full RDF schema and also resource indexes. The scalability of our proposal, however, still needs to be investigated.

4.2.2 Registering an ECA rule

When a new rule is generated at a peer, it is sent to the peer's own super-peer for storage in its local rule base. The super-peer annotates the event, condition and action parts of the rule with the local peers that are relevant to each part (a list of peer IDs).

This can be determined by matching each part of the rule against the super-peer's annotated RDF Schema and/or its resource indexes — the former is useful if no resource is specified in this part of the rule and the latter is useful if a resource is specified. As the annotated RDF Schema and resource indexes at the super-peer evolve, so the annotations on the ECA rules can also be evolved to maintain consistency.

The rule is also sent to all connected super-peers that may be relevant to it — this is determined from the super-peer ID annotations on the originating super-peer's RDF Schema. These super-peers repeat the process of matching each part of the rule against their own annotated RDF Schema, and storing the resulting annotated rule in their own rule base if it is indeed relevant to any of their peer group. We note that, due to schema heterogeneity, the rule may first have to be translated so that its parts are expressed with respect to the local RDF Schema — this is done by calling SeLeNe's Syndication service [20], using the mappings between schemas. This first round of super-peers then propagate the rule to relevant super-peers to which they are connected, and the process continues, thereby propagating the rule through the whole network. The final result is a replica of the rule at each super-peer which is relevant to the rule, annotated with local information about which peers may be affected by each part of the rule.

As the information at super-peers changes with time, it may be that an ECA rule is no longer relevant to that super-peer, in which case the rule can be deactivated in the super-peer's local rule base. Conversely, an ECA rule stored somewhere else may become relevant to a super-peer.

For the first case, if a peer, P, changes so that it no longer has any data associated with a particular RDF Schema node, this change is propagated to the combined RDF Schema stored at the P's super-peer, SP. Using this updated RDF Schema, the annotation of each part of each rule in SP's rule base is updated. If as a result there is a rule r such that the annotation of each part of r is empty, then the rule can be deactivated in SP's rule base (it is not deleted, in case subsequent changes in data require it to be reactivated).

Conversely, if a peer, P, changes so that it now has data associated with a particular RDF Schema node, this change is again propagated to the RDF Schema stored at the P's super-peer, SP. The annotation of each part of each rule in SP's rule base is again updated. Moreover, if the combined RDF Schema at SP has changed from having no data associated with a particular RDF Schema node to now having such data, this change is notified to SP's neighbouring super-peers. If any of these neighbours have ECA rules which may have been made relevant by the new change at SP, they send these ECA rules to SP. These super-peers also request from their neighbours (other than SP) their current set of ECA rules which are potentially relevant to the change, and they forward these rules on to SP. This process repeats until all the potentially relevant ECA rules throughout the network have been sent to SP.

4.2.3 Rule triggering and execution

At run-time, whenever an event E occurs at a peer P, it will notify its super-peer. This will determine whether E may trigger any ECA rule annotated with P's ID. If a rule r might have been triggered, the super-peer will send P r's event query to evaluate.

If r has indeed been triggered, its condition will need to be evaluated, after generating an instantiation of it for each value of the `$delta` variable if this is present in the condition. The annotations on r can be used to determine to which local peers and other super-peers sub-queries of the condition should be dispatched for evaluation. If the `$delta` variable is present in the condition, it will have been instantiated and so we also consult the super-peers' resource indexes for more precise information about which local peers are relevant to sub-queries of the instantiated condition.

If a condition evaluates to true, each corresponding rule action will be sent to, and scheduled, by the super-peers that will execute it. Again this can be determined by the annotations on the rule action and consulting the super-peers' resource indexes.

4.3 Mediated Scenario

In this case, each super-peer manages a global RDF Schema that semantically integrates the local RDF Schemas of its own peer group as well as the global schemas a fixed set of other super-peers. The set of mappings between the local and global RDF Schemas are also managed by the super-peer. These schemas and mappings may evolve with time, but this will be driven by changes to the local or global RDF Schemas, under the control of the overall SeLeNe system. It will not be possible for a peer's RDF description not to conform to the peer's local RDF Schema. Peers cannot dynamically join or leave the network, and so there is no need for an advertisement process from peers to super-peers.

As with the P2P scenario, each super-peer maintains for each node in its global RDF Schema a list of the RDF resources of this type that each peer in its peer group references (a resource index).

Registration of new ECA rules is handled as in the P2P scenario. However, in the mediated scenario RDF Schemas are likely to evolve much less frequently than in the P2P scenario, and so annotations on existing ECA rules are likely to need much less frequent update.

Rule triggering and execution is handled as in the P2P scenario.

4.4 Challenges and Future Work

There are several open issues remaining in realising the distributed ECA architecture we describe above:

- developing algorithms for matching rule event, condition and action parts with the schema-based indexes;
- defining the syntax of messages that will be passed between peers for distributed processing of ECA rules;
- defining the coordination with SeLeNe's distributed query processor for the evaluation of rule conditions;
- defining the coordination with SeLeNe's syndication service, for translating data and rules between heterogeneous schemas;
- more generally, mapping this distributed ECA functionality onto SeLeNe's service-based architecture;
- exploring distributed transactional aspects of the ECA rules (even though we assume that individual events and actions will occur at a single peer, the execution of an ECA rule may trigger another ECA rule and this whole cascade of rule firings may need to have the semantics of a single transaction).

5 Conclusions

In this report we have discussed the requirements for SeLeNe's reactive functionality and its provision via ECA rules over XML or RDF repositories. We have highlighted some of the new issues that arise in the context of such data. We have described a language for ECA rules on XML, and a language for ECA rules on a graph/triple representation of RDF. We have described a prototype centralised implementation of the XML ECA rule language, and the architecture of a distributed implementation of the RDF ECA rule language.

There are several challenges and directions to explore, as highlighted in Sections 2.4, 3.2 and 4.4 above. An important issue is to evaluate the applicability and scalability of our languages, their execution models, and implementation. Deployment in an implementation of the envisaged SeLeNe architecture will allow us to assess the impact of moving from a centralised to a distributed environment, with the additional challenges of network delay, network reliability, synchronisation of rule execution, transactional issues and maintaining consistency of the distributed metadata, tolerance of delays and failures etc.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *VLDB Journal*, 1(1):68–88, 1997.

[2] S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[3] A. Adi, D. Botzer, O. Etzion, and T. Yatzkar-Haham. Push technology personalization through event correlation. In *Proc 26th Int. Conf. on Very Large Databases*, pages 643–645, 2000.

[4] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proc. 2nd. Int. Workshop on the Semantic Web (SemWeb 2001)*, 2001.

[5] J. Bailey, A. Poulovassilis, and P.T. Wood. An Event-Condition-Action Language for XML. In *Proc. WWW'2002*, Hawaii, 2002.

[6] J. Bailey, A. Poulovassilis, and P.T. Wood. Analysis and optimisation for event-condition-action rules on XML. *Computer Networks*, 39:239–259, 2002.

[7] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*, 2002.

[8] A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML: A new paradigm for e-services. *VLDB Journal*, 10(1):39–47, 2001.

[9] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. In *WWW'01*, 2001.

[10] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology.* Addison-Wesley, 1997.

[11] S. Ceri, P. Fraternali, and S. Paraboschi. Data-driven one-to-one web site generation for data-intensive applications. In *Proc. 25th Int. Conf. on Very Large Databases*, pages 615–626, 1999.

[12] E. Cho, I. Park, S. J. Hyum, and M. Kim. ARML: an active rule mark-up language for heterogeneous active information systems. In *Proc. RuleML 2002*, Sardinia, June 2002.

[13] H. Ishikawa and M. Ohta. An active web-based distributed database system for e-commerce. In *Proc. Web Dynamics Workshop, London*, 2001. `http://www.dcs.bbk.ac.uk/webDyn/`.

[14] K. Keenoy, M. Levene, and D. Peterson. Personalisation and Trails in Self e-Learning Networks. See `http://www.dcs.bbk.ac.uk/selene/reports/Del42.pdf`, October 2003. SeLeNe Project Deliverable 4.2, Version 1.1.

[15] K. Keenoy *et al.* Self e-Learning Networks — Functionality, User Requirements and Exploitation Scenarios. See `http://www.dcs.bbk.ac.uk/selene/reports/Del22.pdf`, August 2003. SeLeNe Project Deliverable 2.2.

[16] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Loser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proc. WWW2003*, pages 536–543, 2003.

[17] G. Papamarkos, A. Poulovassilis, and P.T. Wood. Event-condition-action rule languages for the semantic web. In *Proc. Workshop on Semantic Web and Databases, at VLDB'03, Berlin*, 2003.

[18] N. Paton, editor. *Active Rules in Database Systems.* Springer-Verlag, 1999.

[19] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proc 7th Int. Conf. on Cooperative Information Systems (CoopIS'2000)*, pages 162–173, 2000.

[20] G. Samaras, K. Karenos, and E. Christodoulou. A Grid Service Framework for Self e-Learning Networks. See `http://www.dcs.bbk.ac.uk/selene/reports/Del3.pdf`, August 2003. SeLeNe Project Deliverable 3.

[21] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 413–424, 2001.

[22] G. Wagner. How to design a general rule markup language? In *Invited talk at the Workshop XML Technologien für das Semantic Web (XSW 2002)*, Berlin, June 2002.

[23] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, San Mateo, California, 1995.

[24] World Wide Web Consortium. XML Path Language (XPath), Version 1.0. See `http://www.w3.org/TR/xpath`, November 1999. W3C Recommendation.

[25] World Wide Web Consortium. XSL Transformations (XSLT), Version 1.0. See `http://www.w3.org/TR/xslt`, November 1999. W3C Recommendation.

[26] World Wide Web Consortium. XQuery 1.0: An XML Query Language. See `http://www.w3.org/TR/xquery`, November 2002. W3C Working Draft.

[27] World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. See `http://www.w3.org/TR/DOM-Level-3-Core/`, February 2003. W3C Working Draft.

A BNF of the RDF ECA Language

```
rule      ::= 'ON' event
              'IF' condition
              'DO' action  ';;'

event     ::= ['LET' let-expr (',' let-expr)* 'IN']
              (('INSERT' | 'DELETE') e ['AS' 'INSTANCE' 'OF' class]
                                       ['USING' 'NAMESPACE' nspace]
               |
               ('INSERT' | 'DELETE') triple
               |
               'UPDATE' upd_triple
              )

condition ::= ['not'] ce ((('and' | 'or') ['not'] ce)+)? | 'TRUE'

action    ::= (['LET' let-expr (',' let-expr)* 'IN']
               (('INSERT' | 'DELETE') triple (',' triple)* ';'
                |
                'UPDATE' upd_triple (',' upd_triple)* ';'
                |
                'INSERT' e 'AS' 'INSTANCE' 'OF' class
                ['USING' 'NAMESPACE' nspace] ';'
                |
                'DELETE' e ['AS' 'INSTANCE' 'OF' class]
                ['USING' 'NAMESPACE' nspace] ';'
               )+
              )+

e          ::= 'resource(' URI ')' ('/' p)? | var ('[' q ']')* ('/' p)?

p          ::= p '/' p | p '[' q ']' | 'target' '(' arc_name ')'
               | 'source(' arc_name ')' | 'element()' | '()'

q          ::= q 'and' q | q 'or' q | e | p | (p | e) operator (p | e | literal)

ce       ::= e ( operator e)?

let-expr  ::= var ':=' e

triple    ::= '(' source_node ',' arc_name ',' target_node ')'

upd_triple::= '(' source_node ',' arc_name ',' target_node '->' target_node ')'

source_node::= (e | '_') ['AS' 'INSTANCE' 'OF' class]
                         ['USING' 'NAMESPACE' nspace]

arc_name   ::= (string | '_') ['USING' 'NAMESPACE' nspace]

target_node::= (e | '_' | string) ['AS' 'INSTANCE' 'OF' class]
                                  ['USING' 'NAMESPACE' nspace]

var         ::= '$' attribute

class       ::= string

nspace      ::= string

URI         ::= string
```

B Computational Completeness of the RDF ECA Language

To show computational completeness of our language over integers, we can show that it simulates *while* programs (which are themselves a higher-level syntax for *counter* programs). While programs are constructed from the following constructs:

1. sequential composition of statements: `s_1`; `s_2`; ...; `s_n`
2. variables x, y, z ... over natural numbers
3. assignment statements: x:= 0, x:=x+1, x := y, x:=x−1
4. conditional statements, where `s` and `s'` are statements: `if x = 0 then s else s'`
5. while statements, where `s` is a statement: `while x > 0 do s`

We can encode while programs in our RDF ECA Language as follows:

In the RDF description base, the natural numbers are represented as resources of a class `Number`. There is a property `succ` from instances of `Number` to instances of `Number` (the successor function) such that: one instance of Number has label `0`, no incoming `succ` arc, and one outgoing `succ` arc (to the resource representing 1); all other instances of `Number` have one incoming `succ` arc (from their predecessor) and one outgoing `succ` arc (to their successor).

Given a while program P, let `s_1`; `s_2`; ...; `s_m` be all the statements appearing in P. Assume there is a class `Counter` of resources labelled `flag_1`, `flag_2`, ...`flag_m`, where `flag_i` is associated with statement `s_i`. Also, assume that initially there are no instances of `Counter` present.

The encoding of the constructs of a while program P is then as follows:

1. Sequential composition of statements `s_1`; `s_2`; ...; `s_n`:

   As stated above, we assume that initially there are no instances of `Counter` present.

   The sequence of statements is triggered by the insertion of resource `flag_1` associated with stagement `s_1`.

   The sequential composition of the statements `s_1`;`s_2`; ...; `s_n` is encoded by the following set of ECA rules, which has a lower priority than the rule encoding statement `s_1` (so that the ECA rule(s) encoding `s_1` will be executed first, followed by this rule):

   ```
   ON INSERT resource('flag_1') AS INSTANCE OF Counter
   IF TRUE
   DO INSERT resource('flag_2') AS INSTANCE OF Counter;
      ...
      INSERT resource('flag_n') AS INSTANCE OF Counter;
   ```

2. Variables x, y, z . . . over natural numbers:

   Variables x, y, z . . . are represented as resources of a class `Variable`, and are labelled `x`, `y`, `z` . . .

   There is property `has-value` from instances of `Variable` to instances of `Number`, which indicates the current value of each variable.

3. An assignment statement `s_i`:

   This is represented by a rule with event part `ON INSERT resource('flag_i')`, condition part `TRUE` and action part as follows:

   For x:=0:

   ```
    UPDATE (x,has-value,_->0);  DELETE resource('flag_i');
   ```

   For x:=y:

   ```
    LET $new = resource('y')/target(has-value) IN
    UPDATE (x,has-value,_->new);  DELETE resource('flag_i');
   ```

   For x:=x+1:

   ```
    LET $new = resource('x')/target(has-value)/target(succ) IN
    UPDATE (x,has-value,_->new);  DELETE resource('flag_i');
   ```

   For x:=x−1:

   ```
    LET $new = resource('x')/target(has-value)/source(succ) IN
    UPDATE (x,has-value,_->,new);  DELETE resource('flag_i');
   ```

4. A conditional statement `s_i` of the form `if x = 0 then s_j else s_k`:

   This is represented by two ECA rules:

   ```
   ON INSERT resource('flag_i')
   IF resource('x')[target(has-value)='0']
   DO INSERT resource('flag_j') AS INSTANCE OF Counter;
      DELETE resource('flag_i');

   ON INSERT resource('flag_i')
   IF not resource('x')[target(has-value)='0']
   DO INSERT resource('flag_k') AS INSTANCE OF Counter;
      DELETE resource('flag_i');
   ```

5. A while statement `s_i` of the form `while x > 0 do s_j`:

   This is represented by the following two ECA rules:

   ```
   ON INSERT resource('flag_i')
   IF not resource('x')[target(has-value)='0']
   ```

```
DO DELETE resource('flag_i');
   INSERT resource('flag_j') AS INSTANCE OF COUNTER;
   INSERT resource('flag_i') AS INSTANCE OF COUNTER;

ON INSERT resource('flag_i')
IF resource('x')[target(has-value)='0']
DO DELETE resource('flag_i')
```