RDFTL : An Event-Condition-Action Language for RDF

George Papamarkos, Alexandra Poulovassilis, Peter T. Wood

Abstract

RDF is becoming a core technology in the Semantic Web. Providing the ability to describe metadata information that can be easily navigated, and the ease of storing it in existing relational database systems, have made RDF a very popular way of expressing and exchanging metadata information. However, the use of RDF in dynamic applications over distributed environments that require timely notification of metadata changes raises the need for mechanisms for monitoring and processing such a changes. Event-Condition-Action (ECA) rules are a natural candidate to fulfill this need. In this paper, we study ECA rules in the context of RDF metadata. We give a detailed description of a language to define ECA rules on RDF repositories. We specify the syntax and semantics of the language, and we illustrate its use by examples. We also describe the architecture of a system implementing this language, both for centralised and distributed environments.

1 Introduction

In this paper we describe RDFTL (RDF Triggering Language), an event-condition-action rule language providing reactive functionality over RDF metadata stored in RDF repositories. RDF is one of the technologies proposed to realise the vision of the Semantic Web, and it is being increasingly used in distributed web-based applications. Many such applications need to be *reactive*, i.e. to be able to detect the occurrence of specific events or changes in the RDF descriptions, and to respond by automatically executing the appropriate application logic. *Event-condition-action* (ECA) rules are one way of implementing this kind of functionality. An ECA rule has the general syntax

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

The event part specifies when the rule is *triggered*. The condition part is a query which determines if the information system is in a particular state, in which case the rule *fires*. The action part states the actions to be performed if the rule fires. These actions may in turn cause further events to occur, which may in turn cause more ECA rules to fire.

There are several advantages in using ECA rules to implement this kind of functionality, rather than implementing it directly in application code. Firstly, ECA rules allow an application's reactive functionality to be specified and managed within a rule base rather than being encoded in diverse programs, thus enhancing the modularity, maintainability and extensibility of applications. 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 behaviours, in contrast to application code that is typically specialised to a particular kind of reactive scenario.

The work presented here has largely been motivated by our work in the SeLeNe project (see `http://www.dcs.bbk.ac.uk/selene/`). The primary goal of the SeLeNe project is to investigate techniques for managing evolving RDF repositories of educational metadata and for providing a wide variety of services over such repositories, including syndication

and personalisation services. Peers in a SeLeNe (Self e-Learning Network) will store RDF/S descriptions relating to learning objects (LOs) registered with the SeLeNe and also RDF/S descriptions relating to users of the SeLeNe. Peers may also store system-related RDF/S descriptions. A SeLeNe may be deployed in a centralised or in a distributed environment. 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 [8]:

- 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, e.g. propagating changes in the description of a LO to the description of any composite LOs defined in terms of it.

Studying the use of ECA rules for RDF in such a large scale distributed application was a major motivation for the evolution of our RDFTL language, allowing investigation of the way the system implementing the language operates in a distributed environment in cooperation with a distributed query processor and local RDF repositories.

One precursor of the work presented here is the XML ECA Language described in [2, 12]. This XML ECA language uses a fragment of XPath for querying the XML documents and an XML update language for performing the actions. The implementation of this language was in a centralised environment.

Outline of this paper: Section 2 discusses the path expression sub-language used in all parts of RDFTL rules for navigating through RDF graphs. Section 3 discusses the syntax of RDFTL rules and gives some examples of its use. Section 4 specifies the execution semantics of RDFTL rules. The architecture supporting RDFTL in both centralised and distributed environments is described in Section 5. Finally, we conclude in Section 6 with future work and further challenges.

2 RDFTL Path Expressions

RDFTL operates over RDF Graphs and thus complies with current RDF standards of syntax, semantics and datatypes. When defining an ECA rule in RDFTL, it is necessary to specify the portion of metadata that each part of the rule deals with: for example, the RDF nodes that will be affected by an event, or the value of an RDF literal used to evaluate a condition. In order to deal with this, RDFTL uses a path-based query sub-language for defining queries over an RDF graph. In this section we describe the way this path-based query sub-language operates over RDF graphs. We first describe the built-in functions used by the sub-language for navigating around an RDF graph. We then present the abstract syntax and denotational semantics of the sub-language, following the approach of [14, 15].

The built-in functions used to perform basic navigation operations within an RDF graph and to relate the RDF datatypes to one another are as follows:

The *resource* function takes a URI as its argument and returns a singleton containing the RDF resource described by the URI, or all the resources in the graph when the URI is equal to the empty string.

The *sources* function takes an RDF *Predicate* and an RDF *Object* as arguments and returns the set S of RDF *Subject*s such that, for each $x \in S$, $(x, Predicate, Object)$ is a triple in the RDF graph.

The *targets* function takes an RDF *Predicate* and an RDF *Subject* as arguments and returns the set S of RDF *Object*s such that, for each $x \in S$, $(Subject, Predicate, x)$ is a triple in the RDF graph.

The *element* function returns the i^{th} element of an RDF collection if passed the integer i as an argument, or returns all the elements of the collection if no argument is supplied.

The *value* function returns the value of a given RDF resource in the form of a string.

An extra function that checks whether a node is the root of an RDF collection is defined, exploiting the functionality of the functions above. The $isCollection(x)$ function returns true if and only if the RDF node x is an RDF resource and the node returned by the *targets* function with predicate $rdf : type$ and subject x as parameters is one of the RDF classes $rdf : Bag$, $rdf : Seq$ or $rdf : Alt$. $rdf : type$ is an instance of the $rdf : Property$ type and is used to state that a resource is an instance of a class. In the case of an RDF collection it denotes that a node (the root of the RDF collection) is an instance of the RDF class $rdf : Bag$, $rdf : Seq$ or $rdf : Alt$. The $rdf : Bag$, $rdf : Seq$ and $rdf : Alt$ classes are all subclasses of the $rdf : Container$ class. Formally they are no different to each other and are used only to make the RDF files more readable by humans, indicating that a collection is intended to be unordered ($rdf : Bag$), numerically ordered ($rdf : Seq$) or that its typical use is the selection of one of its members ($rdf : Alt$). The predicate $rdf : \_i$, where $i \in \mathbf{N}$, is used to relate an RDF collection node to its i^{th} member.

Having defined all the functions that are needed in order to navigate around an RDF graph, we give below the abstract syntax of RDFTL's path expressions, where $uri \in URI$, $arc\_name \in Predicate$, $i \in Number$, $s \in String$ $qry \in Query$, $p \in Path$, and $q \in Qualifier$:

$$
\begin{array}{rcl}
qry & ::= & \text{"}resource(\text{"}uri\text{"})\text{"}\ (\text{"}/\text{"}p)? \\
p & ::= & p\text{"}/\text{"}p \mid p\text{"}[\text{"}q\text{"}]\text{"} \mid \text{"}target(\text{"}arc\_name\text{"})\text{"} \mid \text{"}source(\text{"}arc\_name\text{"})\text{"} \mid \\
 & & \text{"}element(\text{"}i\text{"})\text{"} \mid \text{"}element()\text{"} \\
q & ::= & q\ \text{"}and\text{"}q \mid q\ \text{"}or\text{"}\ q \mid \text{"}not\text{"}\ q \mid p \mid p\ \text{"} = \text{"}\ s \mid p\text{"} \neq \text{"}\ s
\end{array}
$$

Based on this abstract syntax and the data model defined earlier, we now give the denotational semantics of RDFTL's path expressions. We write $S[\![p]\!]x$ to indicate the set of nodes selected by path expression p starting from the node x as context node, and we write $Q[\![q]\!]x$ to denote whether the qualifier q is satisfied when the context node is x :

$$
\begin{array}{lcl}
S & : & Expression \rightarrow Node \rightarrow Set(Node) \\
S\ [\![resource(uri)]\!]\ x & = & \{x_1 \mid value(x_1) = uri\} \\
S\ [\![p_1/p_2]\!]\ x & = & \{x_2 \mid x_1 \in S\ [\![p_1]\!]\ x\ ,\ x_2 \in S\ [\![p_2]\!]\ x_1\} \\
S\ [\![p[q]]\!]\ x & = & \{x_1 \mid x_1 \in S\ [\![p]\!]\ x\ ,\ Q\ [\![q]\!]\ x_1\ \} \\
S\ [\![target(arc\_name)]\!]\ x & = & \{x_1 \mid x_1 \in\ targets(arc\_name, x)\} \\
S\ [\![source(arc\_name)]\!]\ x & = & \{x_1 \mid x_1 \in\ sources(arc\_name, x)\} \\
S\ [\![element()]\!]\ x & = & \textbf{if}\ isCollection(x) \\
 & & \textbf{then}\ \{x_1 \mid x_1 \in\ (S\ [\![target()]\!]\ x - S\ [\![target(rdf:type)]\!]\ x)\}\ \textbf{else}\ error; \\
S\ [\![element(i)]\!]\ x & = & \textbf{if}\ isCollection(x)\ \textbf{and}\ targets(rdf:type, x) = rdf:Seq \\
 & & \textbf{then}\ \{x_1 \mid x_1 \in\ S\ [\![target(rdf:\_i)]\!]\ x\}\ \textbf{else}\ error; \\
Q & : & Qualifier \rightarrow Node \rightarrow Boolean \\
Q\ [\![q_1\ and\ q_2]\!]\ x & = & Q\ [\![q_1]\!]\ x \wedge Q\ [\![q_2]\!]\ x \\
Q\ [\![q_1\ or\ q_2]\!]\ x & = & Q\ [\![q_1]\!]\ x \vee Q[\![q_2]\!]\ x \\
Q\ [\![not\ q]\!]\ x & = & \neg\ Q\ [\![q]\!]\ x \\
Q\ [\![p]\!]\ x & = & S\ [\![p]\!]\ x \neq \emptyset \\
Q\ [\![p = s]\!]\ x & = & \{x_1 \mid x_1 \in\ S\ [\![p]\!]\ x,\ value(x_1) = s\} \neq \emptyset \\
Q\ [\![p \neq s]\!]\ x & = & \{x_1 \mid x_1 \in\ S\ [\![p]\!]\ x, value(x_1) \neq s\} \neq \emptyset
\end{array}
$$

3 The RDFTL Language

Having described the path expressions RDFTL uses for querying RDF metadata, we now proceed to describe the RDFTL ECA language as a whole. RDFTL allows the definition of event-condition-action rules over RDF metadata, operating directly on the graph/triple representation of the RDF. An early draft of this language was described in [12]. RDFTL has evolved considerably from that early draft and now matches more closely the RDF data model. RDFTL rules consist of three parts, the event part specifying the event that will trigger the rule, the condition part specifying the condition that must hold for the rule to fire, and the action part specifying the actions to be taken whenever the rule fires. We consider each of these parts of a rule in turn below.

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 expressed in the sub-language described in Section 2, 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 again 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 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 rule is a boolean-valued expression which may reference the `$delta` variable. This expression may 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 → 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 Learning Object metadata illustrated in Figure 1 and to the fragment of a user's personal metadata illustrated in Figure 2. In Figure 2, ext1 is the IMS-LIP schema and ext3 is SeLeNe's User Profile schema (see [7] for details of these schemas).

Suppose 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:

```
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

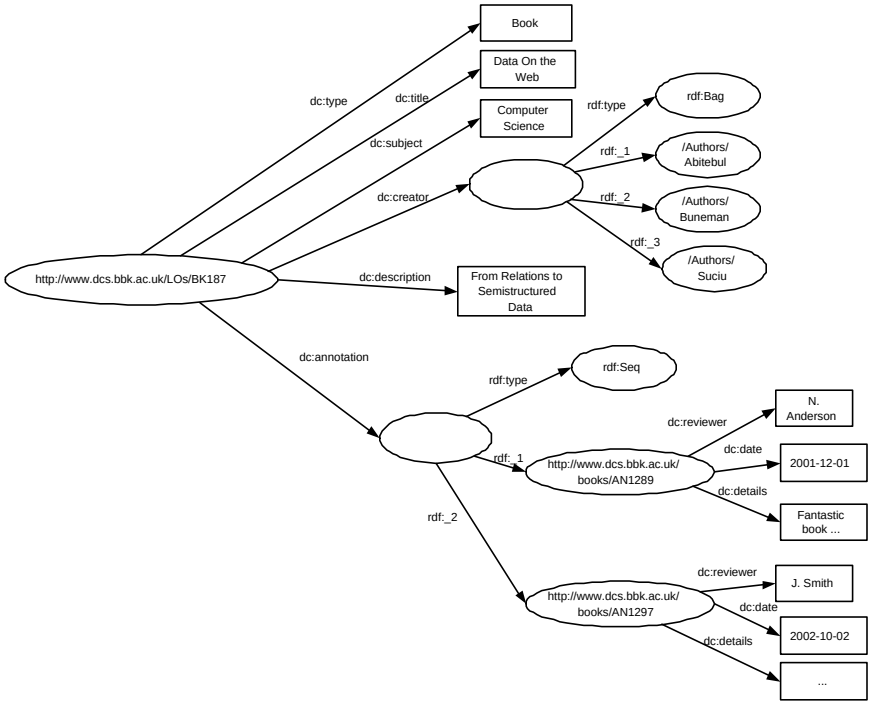

Figure 1: Example of LO Metadata

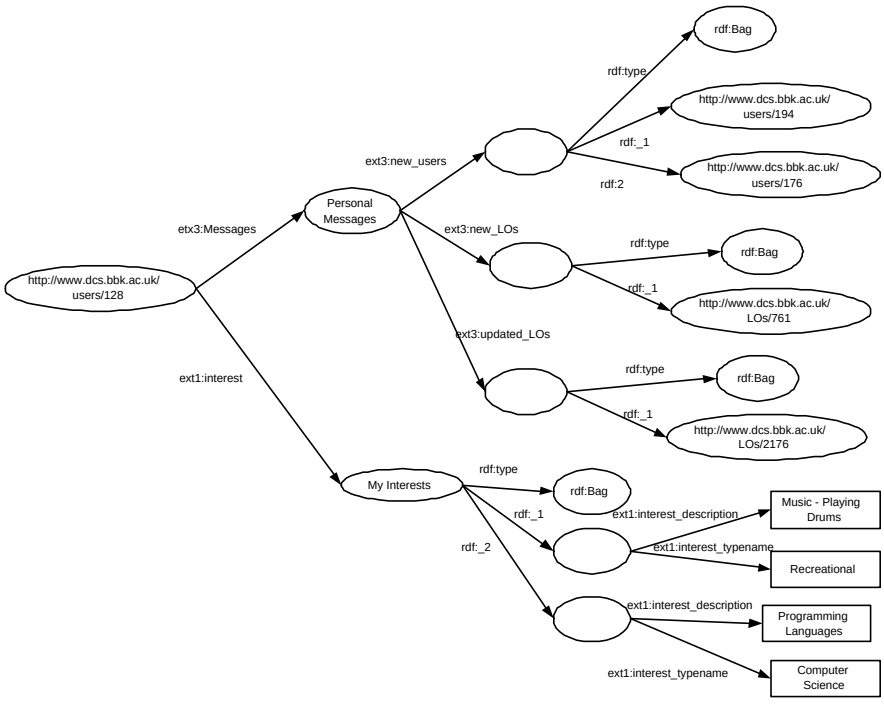

Figure 2: Example of User Metadata

`$new_los` to the newly inserted LO (we use the syntax `seq++` to indicate an increment in the collection's element count).

As another 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);;
```

4 RDFTL Rule Execution Semantics

In this section we describe the rule execution semantics of RDFTL, i.e. the way that ECA rules are executed in response to a triggering event, and the resulting RDF graph. Our ECA rule execution semantics are expressed as a recursive function, *execSched*, which takes as input an *RDF graph* and a *schedule*. The schedule consists of a sequence of *updates* which are to be executed on the RDF graph, an update having the same syntax as a rule action except that there are no occurrences of the `$delta` variable within it. The execution of an update may cause events to occur. These may cause rules to fire, modifying the schedule with new sub-sequences of updates. The rule execution continues in this fashion until the schedule becomes empty.

The events detectable by our system are determined by the syntax of the event parts of our ECA rules as described in Section 3, where we also specified for each kind of event when a rule is deemed to have been *triggered*, and what is its set of instantiations for the `$delta` variable[1].

The condition and action parts of an RDFTL rule may or may not contain occurrences of the `$delta` variable. If neither the condition nor the action part contain occurrences of `$delta`, then the rule is a *set-oriented rule*, otherwise it is an *instance-oriented rule*. A set-oriented rule *fires* if it is triggered and its condition evaluates to $True$. An instance-oriented rule fires if it is triggered and its condition evaluates to $True$ for some instantiation of `$delta`.

A rule's action part consists of one or more actions. If a set-oriented rule fires, then one copy of its action part is prefixed to the current schedule. If an instance-oriented rule fires then one copy of its action part is prefixed to the current schedule for each value of `$delta` for which the rule's condition evaluates to true, in each case substituting all occurrences of `$delta` within the action part by one specific instantiation for `$delta`; the ordering of these multiple copies of the rule's action part is arbitrary[2].

All rules have 'immediate' *coupling mode*, meaning that if a rule fires then the updates generated by its actions are prefixed to the current schedule (as, for example, in the SQL3 trigger standard [10]). If multiple rules fire as a result of an event occurrence, then the updates of higher-priority rules precede those of lower-priority ones on the schedule. We thus require that there is a total ordering imposed on the set of ECA rules (as, for example, in SQL3 [10]). We also assume that all rules have the same *binding mode*, whereby any

[1] We note that our system supports *semantic* rather than *syntactic* triggering — syntactic triggering happens if instances of an event occur, while semantic triggering happens if instances of an event occur and make changes to the RDF graph.

[2] We assume that instance-oriented rules are well-defined, in the sense that the same final RDF graph will result when rule execution terminates irrespective of the order in which copies of a rule's actions are scheduled.

occurrences of the `$delta` variable appearing in a rule's condition or action parts are bound to the state of the RDF graph in which the rule's condition is evaluated[3].

Below we specify our rule execution semantics as a recursive function *execSched* which takes an RDF graph and schedule, and repeatedly executes the update at the head of the schedule, amending the schedule with the updates generated by rules that fire along the way. If *execSched* terminates, it outputs the final RDF graph and the final, empty, schedule. In this specification, [] denotes the empty list, $(x : y)$ a list with head x and tail y, and ++ is the list append operator. We also assume the following standard function for 'left folding' a binary function f into a list:

```
foldl f a []     = a
foldl f a (x:xs) = foldl f (f x a) xs
```

The function *exec u gr* executes an update u on the current RDF graph gr, and returns the new RDF graph, gr, together with the set of nodes and arcs, *changes*, inserted or deleted by u. Each rule's instantiations for its `$delta` variable will subsequently be extracted from this *changes* set.

We assume that ECA rules are identified by unique identifiers of type *RuleId*. The expression *deltas ch r* denotes the set of instantiations of the `$delta` variable for rule r, given the current set of overall changes *ch*. So r is triggered when *deltas ch r* is non-empty. *condition r* returns the rule's condition query and *actions r* its list of actions. *isSetOriented r* returns whether or not r is a set-oriented rule.

The function *triggers* takes an update, and returns a list comprising the identifiers of the rules that may be triggered by that update, in decreasing order of the rules' priority. *triggers* does this by performing a syntactic analysis of updates and rule event parts, and is conservative in the sense that if *triggers u* does not return a rule identifier, then there is no RDF graph in which execution of u can trigger that rule.

The function *schedRules* applies the function *schedRule* to each rule that may be triggered by u, in decreasing order of these rules' priority. *schedRule* determines whether a rule has indeed triggered in which case the function *updateSched* is called. This determines if a rule has fired, and if so calls *updateSched* to update the schedule's prefix. Within *updateSched*, *eval q gr* evaluates a query q with respect to an RDF graph gr, and *substitute e d* replaces any occurrences of `$delta` within e by d:

```
execSched : (RDFGraph,Schedule) -> (RDFGraph,Schedule)
execSched (gr,[])  = (gr,[])
execSched (gr,u:s) =
    let (changes,gr) = (exec u gr) in
        execSched (schedRules (changes,gr,(u:us)))

schedRules : (Changes,RDFGraph,Schedule) -> (RDFGraph,Schedule)
schedRules (ch,gr,u:s) =
    let (ch,gr,prefix) = (foldl schedRule (ch,gr,[]) (triggers u)) in
        (gr,prefix++s)

schedRule : RuleId -> (Changes,RDFGraph,Schedule,Schedule) ->
                      (Changes,RDFGraph,Schedule,Schedule)
schedRule r (ch,gr,prefix) =
    if (deltas ch r) = {}
    then (ch,gr,prefix)
    else updateSched (ch,gr,deltas ch r,r,prefix)
```

[3]Our rules could be enriched to handle a greater variety of coupling modes and binding modes, but this is an area of future work. A detailed description of the coupling and binding possibilities for ECA rules can be found in [13].

```
updateSched (ch,gr,deltas,r,prefix) =
    if (isSetOriented r)
    then if (eval (condition r) gr)
         then (ch,gr,prefix ++ (actions r))
         else (ch,gr,prefix)
    else (ch,gr,prefix ++ [u | d<-deltas; eval (substitute (condition r) d) gr];
                              u<-substitute (actions r) d]
```

5 System Architecture

As part of our research there is in progress the implementation of a system for processing RDFTL rules in both centralised and distributed environments. The main component of our system is the *RDFTL ECA Engine* (see below). This provides an 'active' wrapper over a 'passive' RDF repository, which exploits the query, storage and update functionality of such a repository. We thus assume the availability of a Query Service and an Update Service provided by the RDF repository. In our current implementation, the RDF repository is RDFSuite [1, 5] from ICS-FORTH.

In a centralised environment, 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 enables both event detection and update processing at that server.

In a distributed environment, an ECA rule generated at one site of the network might be triggered, evaluated, and executed at different sites. There may thus be an installation of the ECA Engine at several network sites. The architecture that we are implementing is illustrated in Figure 3. 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 in the network. At each super-peer there is installed one ECA Engine operating as a Web Service.

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.

In the dynamic applications that we envisage, ECA rules are likely not to be handcrafted but automatically generated by higher-level presentation and application services. Within the event, condition and action parts of ECA rules there might or might not be references to specific RDF resources i.e. ECA rules may be resource-specific or generic. We currently 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 hold true for the SeLeNe system, but generalising our techniques and architure to support distributed event detection and action execution is an area of future work.

Each ECA engine consists of several sub-components:

- The *RDFTL Language Interpreter* takes as input an RDFTL rule definition and registers it in a Rule Base. The Language Interpreter consists of a *Parser* which verifies the syntactic correctness of the rule, and a *Translator* which translates any path queries within the rule into the query syntax of the underlying RDF repository. In our present implementation, RDFTL path expressions are translated into RQL [6] which is the query language supported by RDFSuite.
- The *Event Detector* detects the occurrence of events within the local RDF repository and the triggering of ECA rules registered within the local rule base. The Event

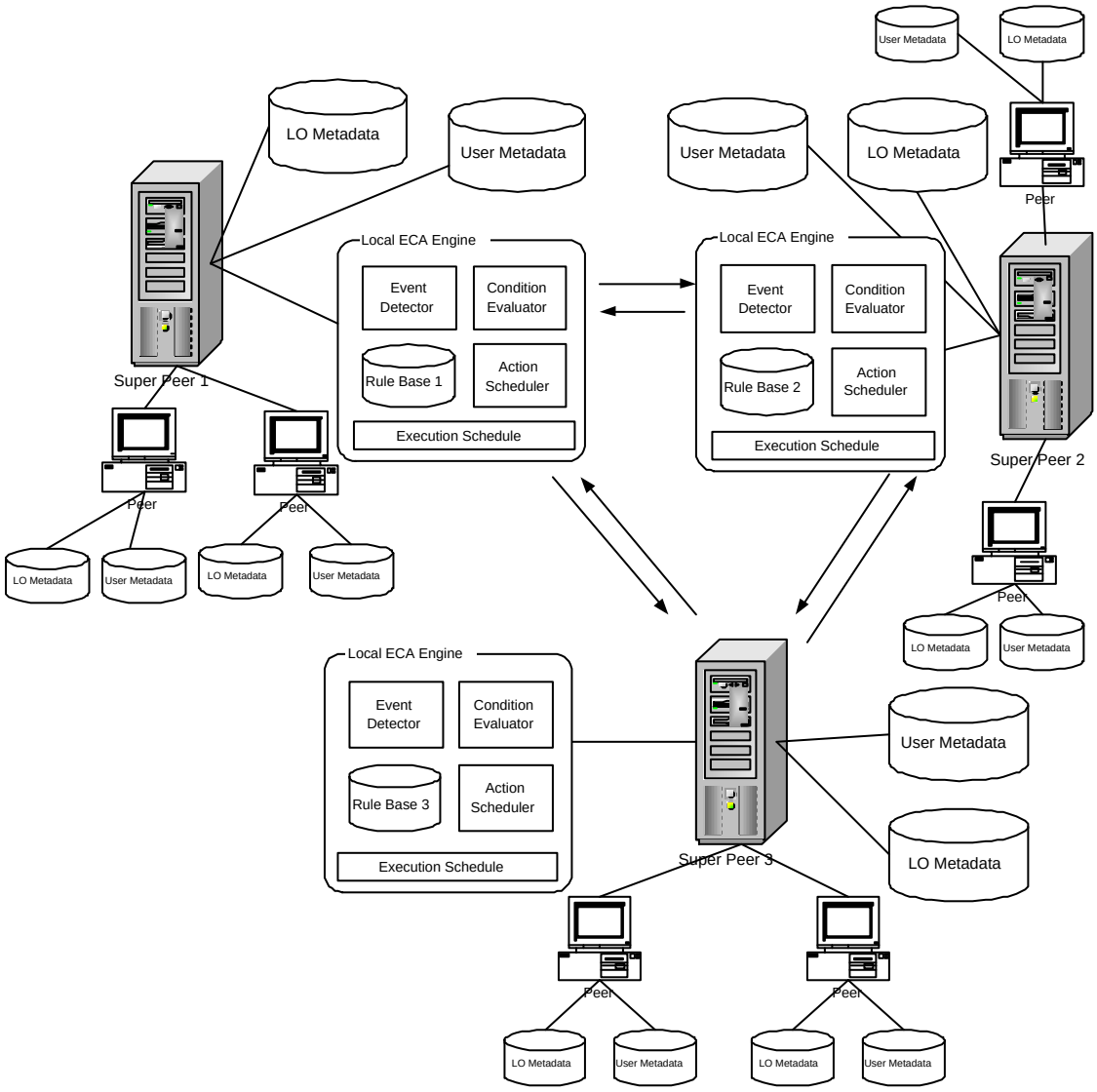

Figure 3: Distributed System Architecture

Detector determines which rules have been triggered by the most recent update to the local RDF repository by invoking the Query Service to evaluate the event queries of rules that may have been triggered.

- The *Condition Evaluator* determines which of the triggered rules should fire. It does this by submitting appropriate queries to the Query Service to etermine which of the triggered rules' conditions are true. This may require distributed query processing across a number of peers, which is provided by the Query service.
- The *Action Scheduler* generates from the action parts of rules that have fired a list of updates to be prefixed to the appropriate execution schedules, and generates the appropriate messages to update these execution schedules. These execution schedules may be either the local execution schedule, or any number of remote execution schedules managed by remote ECA Engines.
- The *Peer Connection Manager* establishes connection for data transfer and message passing between the local ECA Engine and the local RDF repository, in case of a centralised environment, and in addition between different peers' ECA Engines in the case of a distributed environment.

5.1 Operating in a centralised environment

In a centralised environment, one ECA Engine and one RDF repository manager are installed at one server. All update requests to the RDF repository are passed through the ECA Engine which, in coordination with the repository's Update Service, enables both event detection and update processing at the server. Whenever a new ECA rule is registered, the RDFTL language interpreter performs syntax validation and translation of path expressions to the repository's query language. The rule is then stored in the rule base. At run-time, the ECA engine monitors all update requests to the repository. Whenever an update occurs that triggers one or more ECA rules, the ECA engine evaluates the condition parts of these rules, by submitting the appropriate queries to the repository's Query Service. The actions of any rules that have fired are then placed onto the local execution schedule, as described in Section 4 above.

5.2 Operating in a distributed environment

There are two possible scenarios for the way in which super-peers may 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 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. These schemas and mappings may evolve with time, but this will be driven by changes to the local or global RDF Schemas.

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.

5.3 P2P Scenario

Whenever a new ECA rule r is registered at a peer P, it will be sent to P's super-peer for storage. 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. 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 (see for example [3, 11, 9, 4]), and this is our 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, and so need to provide an RDF Schema mapping 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*.

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 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.

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.

5.4 Mediated Scenario

In this case the schemas and mappings may again evolve with time, but this will be driven by changes to the local or global RDF Schemas, under the control of the overall application. 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.

6 Concluding Remarks and Future Work

In this paper we have described a language for defining ECA rules on RDF repositories, including its syntax, semantics and the architecture of a system implementing the language, both for centralised and distributed environments. For the immediate future, we plan to explore more deeply the expressiveness RDFTL — it is straight-forward to show that RDFTL is computationally complete but we wish to investigate also its query and update expressiveness. We will also finish the implementation of both the centralised and

distributed sytems over the ICS-FORTH RDFSuite repository, evaluate our implementations in the context of the SeLeNe project, and determine empirically their performance and scalability characteristics.

More generally, there is as yet no accepted standard query or update language for RDF. If ECA rules are to be supported on RDF repositories, then whatever standards eventually emerge, there is also the parallel issue of designing the event language to match up with the update language. In this paper we have seen how this was done in the context of our particular RDF ECA language. 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 RDF. Another important area is combining ECA rules with transactions and consistency maintenance in RDF repositories.

References

[1] 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.

[2] J. Bailey, A. Poulovassilis, and P. T. Wood. Analysis and Optimisation for Event-Condition-Action Rules on XML. *Computer Networks*, 39(3), 2001.

[3] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS 2002*, 2002.

[4] L. Galanis, Y. Wang, S. R. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB'03*, 2003.

[5] ICS-FORTH. RDFSuite. See `http://139.91.183.30:9090/RDF/`.

[6] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative Query Language for RDF. In *Proc. WWW'2002*, 2002.

[7] 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`, 2003. SeLeNe Deliverable 4.2.

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

[9] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT'03*, 2003.

[10] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer, 1999.

[11] 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.

[12] 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*, September 2003.

[13] N. Paton. *Active Rules in Database Systems*. Springer, 1999.

[14] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies 99*, 1999.

[15] P. Wadler. Two Semantics for XPath. In *Markup Technologies 2000*, 2000.