

RDFTL : An Event-Condition-Action Language for RDF

George Papamarkos, Alexandra Poulouvasilis, 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.

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/seleene/>). 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 [3]:

- 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. One precursor of the work presented here is the XML ECA Language described in [1, 6]. This XML ECA language uses a fragment of XPath for querying the XML documents and an XML update language for performing the actions.

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. We conclude in Section 5 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 [8, 9].

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 *Subjects* 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 *Objects* 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 $q \in Qualifier$:

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

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 :

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

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 [6]. 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 *namespace*]

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* → *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 [2] 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)
```

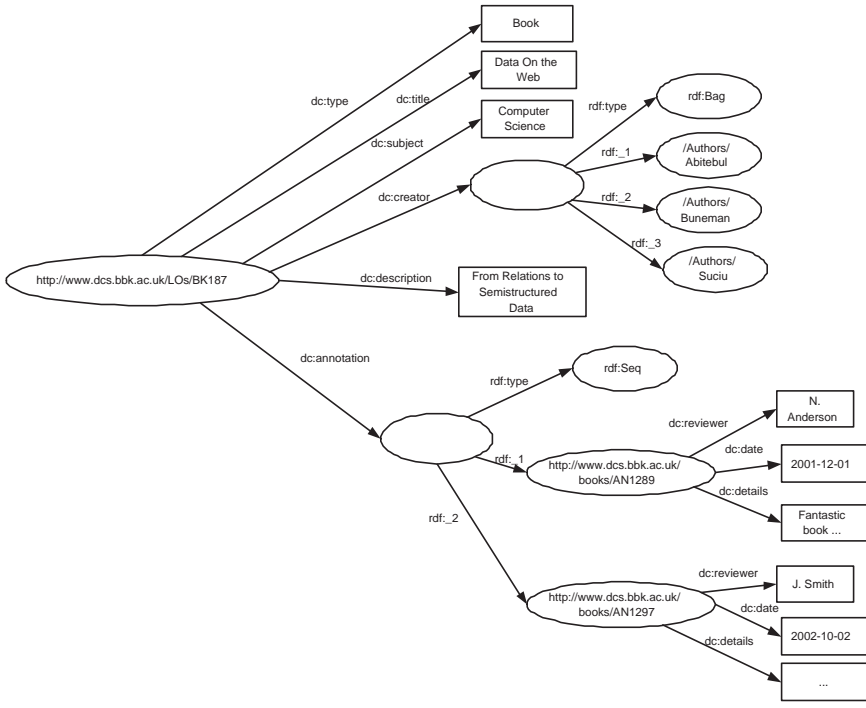


Figure 1: Example of LO Metadata

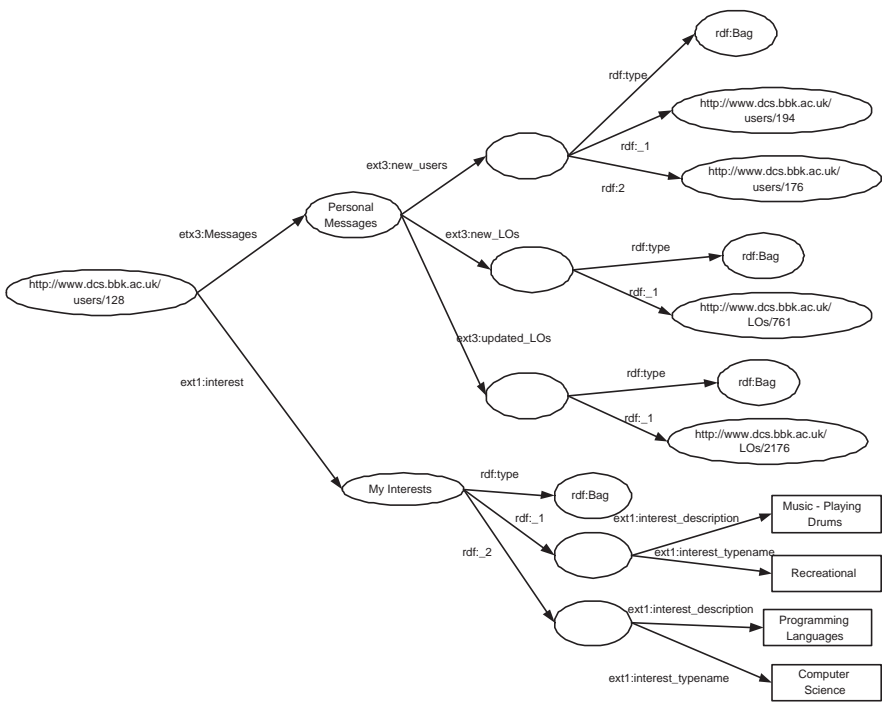


Figure 2: Example of User Metadata

```

        /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 (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);;

```

As a third example, 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).

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 $\$delta$ variable¹.

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 arbitrary².

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 [4]). 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 [4]). We also assume that all rules have the same *binding mode*, whereby any 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 evaluated³.

¹We 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.

²We 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.

³Our 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

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
```

[7].

```

      (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)

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 Concluding Remarks and Future Work

In this paper we have described the RDFTL language for defining ECA rules on RDF repositories, including its syntax and execution semantics. We refer the reader to [5] for a description of the architecture of the system implementing the language, both for centralised and distributed environments. In Appendix A is listed the BNF of the RDFTL language.

For the immediate future, we plan to explore more deeply the expressiveness RDFTL — it is straight-forward to show that RDFTL is computationally complete (see Appendix B) but we wish to investigate also its query and update expressiveness. We will also finish the implementation of both the centralised and distributed systems 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] J. Bailey, A. Poulouvasilis, and P. T. Wood. Analysis and Optimisation for Event-Condition-Action Rules on XML. *Computer Networks*, 39(3), 2001.
- [2] K. Keenoy, M. Levene, and D. Peterson. Personalisation and Trails in Self e-Learning Networks. See <http://www.dcs.bbk.ac.uk/selene/reports/De142.pdf>, 2003. SeLeNe Deliverable 4.2.
- [3] 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.
- [4] 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.
- [5] G. Papamarkos, A. Poulouvasilis, and P. T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proc. 3rd Web Dynamics Workshop, at WWW'2004 (to appear)*, May 2004. See <http://www.dcs.bbk.ac.uk/webdyn3>.
- [6] G. Papamarkos, A. Poulouvasilis, 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.
- [7] N. Paton. *Active Rules in Database Systems*. Springer, 1999.
- [8] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies 99*, 1999.
- [9] P. Wadler. Two Semantics for XPath. In *Markup Technologies 2000*, 2000.

A BNF of RDFTL

```
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

namespace  ::= string

URI        ::= string

```

B Computational Completeness of RDFTL

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: **s₁; s₂; ...; 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 RDFTL 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 $s_1; s_2; \dots; s_m$ be all the statements appearing in P . Assume there is a class **Counter** of resources labelled $flag_1, flag_2, \dots, flag_m$, 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 $s_1; s_2; \dots; s_n$:

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 statement s_1 .

The sequential composition of the statements $s_1; s_2; \dots; s_n$ 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 $x, y, z \dots$ over natural numbers:

Variables $x, y, z \dots$ are represented as resources of a class **Variable**, and are labelled $x, y, z \dots$

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')
```