Event-Condition-Action Rules on RDF Metadata in P2P Environments

George Papamarkos, Alexandra Poulovassilis, Peter T. Wood

{gpapa05,ap,ptw}@dcs.bbk.ac.uk

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

Abstract

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. The use of RDF in applications that require timely notification of metadata changes raises the need for mechanisms for monitoring and processing such changes. Event-Condition-Action (ECA) rules are a natural candidate to fulfill this need. In this paper, we study ECA rules on RDF metadata in P2P environments. We describe a language for defining ECA rules on RDF metadata, including its syntax and execution semantics. We develop conservative tests for determining the termination and confluence of sets of such ECA rules. We describe an architecture supporting such rules in P2P environments, and our current implementation of this architecture. We also discuss techniques for relaxing the isolation and atomicity requirements of transactions.

1 Introduction

This paper concerns the support of Event-Condition-Action rules on RDF metadata in peer-to-peer environments. RDF [54, 56] 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 in e-learning, e-business, e-science, e-government etc. 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. We refer the reader to [58, 47] for a general discussion of ECA rules in databases, where they are more commonly known as *triggers* and where they are used for activities such as incremental maintenance of materialised views and replicas, constraint enforcement, and maintaining audit trails and database usage statistics. More broadly, ECA rules

have also been used in workflow management, network management, personalisation, and publish/subscribe technology [3, 16, 17, 21, 48, 52, 31, 18].

There are several advantages in using ECA rules to implement an application's reactive functionality, rather than encoding it in application code. Firstly, ECA rules allow this functionality to be specified and managed within a rule base rather than being dispersed 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 been motivated by our work in the "SeLeNe: Self e-Learning Networks" project (see `http://www.dcs.bbk.ac.uk/selene/`). The aim of this 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, notification and personalisation services. Peers in a SeLeNe (Self e-Learning Network) store RDF/S descriptions relating to learning objects registered with the SeLeNe, and also RDF/S descriptions relating to users of the SeLeNe. A SeLeNe may be deployed in a centralised or in a distributed environment. In a centralised environment, there is just one 'peer' server which manages all of the RDF/S descriptions relating to learning objects (LOs) and users. In a distributed environment, each peer manages some fragment of the overall RDF/S descriptions.

SeLeNe's reactive functionality provides the following aspects of the user requirements discussed in [35]:

- 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.

Outline of this paper: Section 2 gives an overview of related work in ECA rule languages. Section 3 describes RDFTL, our ECA rule language for providing reactive functionality over RDF metadata stored in RDF repositories. Section 4 develops conservative tests for determining the termination and confluence of sets of RDFTL rules. Section 5 then describes our architecture for supporting RDFTL in P2P environments. We describe how rules are registered at peers and propagated through the network, and we discuss rule execution in P2P environments. Section 6 describes our current implementation of this architecture. It also discusses techniques for relaxing the isolation and atomicity requirements of transactions. We give our concluding remarks in Section 7.

2 Related Work

Developing ECA rule support for RDF in a large-scale distributed application such as SeLeNe 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 [9, 10]. This language uses a fragment of XPath for querying XML documents within the event

and condition parts of rules, and an XML update language for specifying the rule actions. Techniques are developed in [9, 10] for determining the triggering and activation relationships between pairs of rules, which can be 'plugged into' existing frameworks for ECA rule analysis. This XML ECA language could be used for operating on RDF which has been serialised as XML. The RDFTL language that we describe below on the other hand operates directly on the graph/triple representation of RDF and also takes explicit advantage of the available RDFS schema information. To our knowledge, RDFTL is the first ECA rule language developed specifically for RDF/S.

A number of other ECA rule languages for XML have been proposed: Reference [16] discusses extending XML repositories with ECA rules in order to support e-services. Reference [17] discusses a more specific application of this approach to push technology where rule actions are methods that cannot update the repository, and hence cannot trigger other rules. Reference [15] defines an active rule language for XML whose syntax is based on the SQL3 triggers standard. This language is more complex than that of [9, 10] as it allows full XPath in the event parts of rules, and full XQuery in the condition and action parts. However, analysing the behaviour of ECA rules expressed in this more complex language is not considered, and there is in general a trade-off between complexity of an ECA language on the one hand and the ease of analysing rules expressed in it on the other.

ARML [23] provides an XML-based rule description for rule sharing among different heterogeneous ECA rule processing systems; conditions and actions are defined abstractly as XML-RPC methods which are later matched with system-specific methods. GRML [57] is a multi-purpose rule markup language for defining integrity, derivation and ECA rules; it uses an abstract syntax based on RuleML, leaving the mapping to a real language for each underlying system implementation.

Other related work is [2, 46] which discusses monitoring and subscription in Xyleme, an XML warehouse supporting subscription to web documents. A set of *alerters* monitor simple changes to web documents. A *monitoring query processor* then performs more complex event detection and sends notifications of events to a *trigger engine* which performs the necessary actions, including creating new versions of XML documents. The focus of this reactive functionality is highly tuned to this specific application.

Active XML [1] provides similar functionality to that provided by XML ECA rules by embedding calls to web services within XML documents via special tags. When a web sevice in an Active XML document is invoked, the document is expanded with the results returned. The aim is to integrate distributed data and distributed computation in a peer-to-peer architecture.

In the commercial arena, triggers on XML data are now supported by all the major relational DBMS vendors and also by some native XML repository vendors. However, this is confined to document-level triggering, and only events concerning the insertion, deletion or update of an XML document can be caught. In relational DBMS it is however possible to decompose XML documents into a set of relational tables, potentially allowing developers to exploit existing relational triggering functionality in order to define finer-grain triggers over XML data.

An extensive survey of other candidate techniques for providing reactivity in Web applications and the Semantic Web can be found in [49], including a discussion of logics for reasoning about state changes and updates, update languages for the Web, and rule-based agent frameworks. Other related work includes [34] which discusses using ECA rules in P2P systems in order to encode policies for the exchange of data between the peer databases.

A complementary technology to ECA rules are content-based publish/subscribe systems [26]. In such systems, publishers are information providers, or producers of events,

subscribers express their interest in particular events, and the system notifies subscribers of every event occurrence matching their criteria. P2P networks that support publish/subscribe, such as [22] for example, support more sophisticated distributed event definition and detection than the ECA rules approach we propose in this paper. On the other hand, our approach allows the definition and execution of more complex actions than just simple notifications.

3 The RDFTL Language

3.1 RDFTL Syntax

RDFTL (*RDF Trigger Language*) operates over RDF graphs and complies with the current RDF standards of syntax, semantics and datatypes [54, 55, 56]. RDFTL assumes that RDF graphs conform to one or more RDFS schemas, as follows:

(a) every resource in the RDF graph belongs to an RDFS class (in addition to belonging to the default `rdfs:Resource` class);

(b) every property in the RDF graph is declared in the RDFS schema, along with domain and range constraints;

(c) the subject and object of every property in the RDF graph are of the declared subject and object type of the property in the RDFS schema.

3.1.1 RDFTL Path Expressions

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. RDFTL uses a path-based query sublanguage for defining queries over an RDF graph. The abstract syntax of this path-based sublanguage is as follows, where e is a query, p is a path expression, q is a qualifier, *uri* is a URI, *arc_name* is a predicate and s is a string:

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

The above syntax is similar to that for XPath [53], except that: $resource(uri)$ matches the resource given by *uri* in the RDF graph being queried; $target(arc\_name)$ returns the set of *object* nodes related by the predicate *arc_name* to the set of *subject* nodes given by the context; and $source(arc\_name)$ returns the set of *subject* nodes related by the predicate *arc_name* to the set of *object* nodes given by the context. The path expressions of RDFTL also resemble those of RDFPath [37] except that the graph navigation functions in RDFPath are `child` and `parent` instead of `target` and `source`.

We give below 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[1]. In the denotational specification below, the *value* function returns the value of its argument URI in the form of a string. The *targets* function takes an RDF predicate p and an RDF subject x as arguments and returns the set S

[1]By convention, when specifying the denotational semantics of a language, arguments of the semantic functions which are expressions in the language are delimited by $[\![$ and $]\!]$.

of RDF objects such that, for each $y \in S$, (x, p, y) is a triple in the RDF graph. The argument p may be the wildcard symbol _ in which case $targets(\_, x)$ returns the set of RDF objects y such that (x, p, y) is a triple in the RDF graph for any p. Similarly, the *sources* function takes an RDF predicate p and an RDF subject x as arguments and returns the set S of RDF objects such that, for each $y \in S$, (y, p, x) is a triple in the RDF graph. The argument p may be the wildcard symbol _ in which case $sources(\_, x)$ returns the set of RDF objects y such that (y, p, x) is a triple in the RDF graph for any p.

$$
\begin{array}{lcl}
S & : & Expression \rightarrow Node \rightarrow Set(Node) \\
S\ [\![resource(uri)]\!]\ x & = & \{y \mid value(y) = uri\} \\
S\ [\![p_1/p_2]\!]\ x & = & \{z \mid y \in S\ [\![p_1]\!]\ x\ ,\ z \in S\ [\![p_2]\!]\ y\} \\
S\ [\![p[q]]\!]\ x & = & \{y \mid y \in S\ [\![p]\!]\ x\ ,\ Q\ [\![q]\!]\ y\ \} \\
S\ [\![target(arc\_name)]\!]\ x & = & \{y \mid y \in\ targets(arc\_name, x)\} \\
S\ [\![source(arc\_name)]\!]\ x & = & \{y \mid y \in\ sources(arc\_name, x)\}
\end{array}
$$

$$
\begin{array}{lcl}
Q & : & Qualifier \rightarrow Node \rightarrow Boolean \\
Q\ [\![q_1\ and\ q_2]\!]\ x & = & Q\ [\![q_1]\!]\ x \wedge Q\ [\![q_2]\!]\ x \\
Q\ [\![q_1\ or\ q_2]\!]\ x & = & Q\ [\![q_1]\!]\ x \vee Q [\![q_2]\!]\ x \\
Q\ [\![not\ q]\!]\ x & = & \neg\ Q\ [\![q]\!]\ x \\
Q\ [\![p]\!]\ x & = & S\ [\![p]\!]\ x \neq \emptyset \\
Q\ [\![p = s]\!]\ x & = & \{y \mid y \in\ S\ [\![p]\!]\ x, value(y) = s\} \neq \emptyset \\
Q\ [\![p \neq s]\!]\ x & = & \{y \mid y \in\ S\ [\![p]\!]\ x, value(y) \neq s\} \neq \emptyset
\end{array}
$$

3.1.2 RDFTL Rule Syntax

Having described the path expressions RDFTL uses for querying RDF metadata, we now describe the RDFTL ECA language as a whole, considering in turn the event part, condition part and action part of a rule.

There is an optional preamble to each rule. This preamble may contain one or more clauses of the form `USING NAMESPACE` *name uri* which associate a local name with a namespace URI. The preamble may also contain a set of *let-expressions* of the form `let` *variable* `:=` *e* which associate a variable name with a path expression *e*.

The event part of a rule is an expression of one of the following three forms:

1. `(INSERT | DELETE)` *e* `[AS INSTANCE OF` *class*`]`

   This detects insertions or deletions of resources specified by the expression *e*. *e* is a path expression, which evaluates to a set of nodes. Optionally, *class* is the name of the RDFS schema class to which at least one of the nodes identified by *e* must belong in order for the rule to trigger.

   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 *class*, if specified[2]. 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. `(INSERT | DELETE)` *triple*

[2]Note that RDFTL supports *semantic* rather than *syntactic* triggering: rule triggering occurs if instances of an event occur and make changes to the RDF graph. In contrast, with syntactic triggering, rule triggering would occur if instances of an event occurred irrespective of whether the RDF graph were changed, e.g. an attempt to insert an existing resource.

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 `$delta` has as its set of instantiations the triples which have triggered the rule. The individual components of one these triples can be obtained by `$delta.source`, `$delta.arc_name` or `$delta.target`.

3. `UPDATE` *upd_triple*

   This detects updates of arcs specified by *upd_triple*, which has the form *(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 `$delta` has as its set of instantiations the triples which have triggered the rule. The individual components of one these triples can be obtained by `$delta.source`, `$delta.arc_name`, `$delta.old_target` or `$delta.new_target`.

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. `INSERT` *e* `AS INSTANCE OF` *class*

   `DELETE` *e* `[AS INSTANCE OF` *class*`]`

   for expressing insertion or deletion of a resource.

2. `(INSERT | DELETE)` *triple* `(','` *triple*`)*`

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

3. `UPDATE` *upd_triple* `(','` *upd_triple*`)*`

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

The `AS INSTANCE OF` keyword classifies 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. The semantics of this optional class specification in deletions is as follows: when the name of the class is specified, the RDF resource(s) returned by the path expression *e* that are instances of this class are deleted; in the case that no class is specified, all the RDF resources returned by *e* are deleted, regardless of their classification.

The triples in the case of arc manipulation have the same form as in the event sublanguage. The wildcard '`_`' may appear inside triples in the action sublanguage, 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

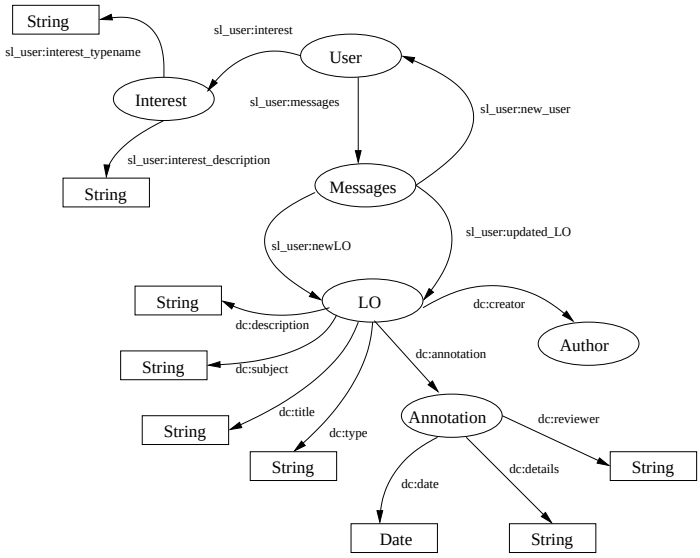

Figure 1: RDF schema describing users and learning objects.

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.

Example 1 *The following examples refer to the Learning Object metadata and to the fragment of a user's personal metadata illustrated in Figure 1 (see [36] for details of these schemas and their use in the SeLeNe system).*

Suppose a Learning Object (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 to user 128's personal messages:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/user
LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
             /target(sl_user:messages)
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO INSERT ($msgs,sl_user:newLO,$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 defines the variable* `$msgs` *to be user 128's messages. Finally, the* `INSERT` *clause inserts a new arc from* `$msgs` *to the newly inserted LO.*

Example 2 *As another example, if the description of a LO whose subject is the same as one of user 128's areas of interest changes, the following rule inserts a new arc from*

user 128's `Messages` *to the modified LO:*

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/user
LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(sl_user:messages)
ON UPDATE (resource(),dc:description,_->_)
IF $delta.source/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(sl_user:interest)/target(sl_user:interest_typename)
DO INSERT ($msgs,sl_user:updated_LO,$delta.source);;
```

3.2 RDFTL Rule Execution

RDFTL rule execution 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, where the syntax of an update is the same as the syntax of a rule action described above, except that there are no occurrences of the `$delta` variable within it. The updates on the schedule belong to concurrently executing transactions, and are scheduled for execution according to the concurrency control policy being employed.

Execution of the update at the head of the schedule may cause events to occur. These events may cause rules to fire, modifying the schedule with new subsequences of updates to be executed. The events detectable by the system are determined by the syntax of the event parts of RDFTL, as defined above. We also specified above for each kind of event when a rule is considered to have been triggered, and what its set of instantiations for the `$delta` variable is.

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 as a result of the execution of an update belonging to a transaction T, then a copy of the rule's action part is added to the current schedule and executed after the rest of T executes, as a subtransaction of T. This is known as Deferred rule coupling [47] — other coupling modes are also possible for ECA rules, but our current implementation of RDFTL supports only this one.

If an instance-oriented rule fires then one copy of its action part is added 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 instances of the rule's action part is arbitrary. Thus, 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 instances of the rule's actions part are scheduled. See Section 4.2 below for a discussion of conservative tests for verifying this property for RDFTL rules.

It is in general possible that many rules may fire as a result of an event occurrence. The set of rules is partially ordered by a **rule precedence** relationship, *prec*, which is specified by the user or application (and which may be empty). If two rules r_i and r_j fire and r_i *prec* r_j then the updates generated by r_i precede on the schedule the updates generated by r_j. If two rules r_i and r_j fire and they are not related by *prec* then the

updates generated by r_i may precede or may follow the updates generated by r_j. We assume that rules r_i and r_j **commute** in such cases, i.e. that the same final RDF graph will result when rule execution terminates irrespective of the order of scheduling of r_i and r_j. See Section 4.3 below for a conservative test for verifying this property for RDFTL rules.

We finally 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 evaluated. A detailed description of the general coupling and binding possibilities for ECA rules is beyond the scope of this paper and we refer the reader to [47].

4 Analysing RDFTL Rule Behaviour

One of the key recurring themes regarding the successful deployment of ECA rules in systems is the need for techniques and tools for analysing their run-time behaviour [19, 41]. Analysis of ECA rules in active databases is a well-studied topic, with a number of approaches appearing in the literature, e.g. [4, 5, 7, 8, 11, 12, 13, 14, 20], mostly in the context of relational databases. In recent work we also explored analysis of ECA rules for XML data [9, 10]. In this section we develop conservative tests for determining two properties of RDFTL rules: *termination* and *confluence*. A set of ECA rules is said to be *terminating* if for any initial event and any initial database state, the rule execution terminates. In Section 4.1 we first develop conservative tests for when one RDFTL rule may trigger another; acyclicity of the resulting triggering graph between rules then implies definite termination of rule execution. A rule set is said to be *confluent* if the same database state results (if rule execution terminates) irrespective of the order of execution of the instances of actions of instance-oriented rules, or the actions of rules which are not related by the precedence (*prec*) relationship. Several techniques have been developed for detecting confluent ECA rule sets in the context of relational databases, e.g., [5], and in Sections 4.2 and 4.3 we develop analogous tests for sets of RDFTL rules.

4.1 Triggering Relationships and Rule Termination

Triggering relationships between rules can been used to determine whether a set of ECA rules is terminating: A rule r_i *may trigger* a rule r_j if the action of r_i may generate an event which triggers r_j. The *triggering graph* [4, 5] represents each rule as a vertex, and there is a directed arc from a vertex r_i to a vertex r_j if r_i may trigger r_j. Acyclicity of the triggering graph implies definite termination of rule execution.

In order to determine triggering relationships between our RDFTL rules, we need to be able to determine whether an action of some rule may trigger the event part of some other rule. This can be done by defining the sets of RDFS schema nodes and triples that are *matched* by RDFTL rule actions and events. Before doing this, we need to make our definition of an RDFS schema graph more precise, particularly as there are some differences with the standard definition of an RDFS schema.

An RDFS schema graph S comprises nodes representing the built-in classes, such as `rdfs:Resource`, `rdfs:Class`, `rdf:Property` and `rdfs:Literal`, as well as any user-defined classes and properties. These nodes are connected by arcs labelled with built-in properties, such as `rdf:type` and `rdfs:subClassOf`. We require that all user-defined properties have domain and range constraints specified. Consider property p with domain d and range r. Rather than this being represented in S by a pair of triples

$(p, \texttt{rdfs:domain}, d)$ and $(p, \texttt{rdfs:range}, r)$, we instead use the single triple (d, p, r). Examples of this representation are given in Figure 1. One advantage of this representation is that RDFTL path expressions can be evaluated on an RDFS schema graph as outlined below.

Throughout we assume that we always work with the *closure* of the RDF graph and of the RDFS schema, as defined in [54]. This means, for example, that when a resource s is inserted into a class c (i.e. the triple $(s, \texttt{rdf:type}, c)$ is inserted into the RDF graph), s is also inserted into all superclasses of c. Likewise when a triple (u, p, v) is inserted into an RDF graph for any property p, then a triple (u, q, v) is also inserted for all superproperties q of p.

For simplicity in the following definitions (and without loss of generality), we assume that each *source_node*, *target_node* and path expression appearing in the event or action part of an RDTFL rule is a simple variable, defined by a *let-expression* in the rule's preamble. Let S be the RDFS schema to which the rules conform, and G be the RDF graph on which the rules will operate.

For each variable v used in an event part or action part of a rule, we can identify the set $nodes(v, S)$ of nodes in S whose extents may be accessed by evaluating v on graph G. Assume that variable v is defined by path expression e. We compute $nodes(v, S)$ by evaluating e on S as follows.

Recall the syntax and semantics of RDFTL path expressions given in Section 3. In order to evaluate an expression e on a schema graph S rather than an RDF graph, we modify the semantics as follows: (i) expressions $resource(x)$ and $resource()$ both return all nodes in S, and (ii) $p = s$ and $p \neq s$ both return the same set of nodes as p. Note that, because we are querying the closure of S, if $nodes(v, S)$ contains a class c, then it will also contain all superclasses of c, except that we assume that it does not contain the class `rdfs:Resource`.

Given a way of identifying the nodes in a schema matched by a variable defined by a path expression, we can now identify those schema triples matched by a triple in a rule. For each triple $t = (u, p, v)$ appearing in a rule, where u and v are variables or the wildcard '_' and p is a property or '_', $triples(t, S)$ is the set of triples in S whose extents may be accessed by evaluating t on G. If u (resp. v) is '_', then $nodes(u, S)$ (resp. $nodes(v, S)$) is the set of all nodes in S. If p is '_', then let $properties(p)$ be all properties in S; otherwise, let $properties(p) = \{p\}$. Now $triples(t, S)$ is given by

$$\{(x, q, y) \in S \mid x \in nodes(u, S), y \in nodes(v, S), q \in properties(p)\}$$

Note that if $(a, q, b) \in triples(t, S)$, then all triples (c, r, d), where a is a subclass of b, q is a subproperty of r, and b is a subclass of d, will also be in $triples(t, S)$. As above, we assume that no triple in $triples(t, S)$ includes `rdf:Property` or `rdfs:Resource`.

For an action a of the form `INSERT` $t_1, \ldots, t_n$ or `DELETE` $t_1, \ldots, t_n$, where $t_1, \ldots, t_n$ are triples,

$$triples(a, S) = triples(t_1, S) \cup \cdots \cup triples(t_n, S).$$

Clearly, `INSERT` actions can only trigger `INSERT` events, `DELETE` actions can only trigger `DELETE` events, and `UPDATE` actions can only trigger `UPDATE` events.

Consider first an action a given by `INSERT` v_1 `AS INSTANCE OF` c_1 along with an event e given by `INSERT` v_2 `AS INSTANCE OF` c_2. For a class c, we define $nodes(c, S)$ to be the set of nodes in S that includes c and all its superclasses, excluding `rdfs:Resource`. Then action a may trigger event e if $nodes(v_1, S) \cap nodes(v_2, S) \neq \emptyset$ and $nodes(c_1, S) \cap nodes(c_2, S) \neq \emptyset$. Action a may also trigger an event e of the form `INSERT (`u`, rdf:type,` w`)` if $nodes(v_1, S) \cap nodes(u, S) \neq \emptyset$ and $nodes(c_1, S) \cap nodes(w, S) \neq \emptyset$.

Example 3 *Consider the RDF schema S from Figure 1. Suppose that we have rule* r_1 *with its action part containing the following action*

```
INSERT $u AS INSTANCE OF LO
```

where `$u` *is defined by*

```
LET $u := resource(http://www.dcs.bbk.ac.uk/LOs/BK187)
```

and a rule r_2 *with event part*

```
INSERT $v AS INSTANCE OF LO
```

where `$v` *is defined by*

```
LET $v := resource()/target(sl_user:messages)/target(sl_user:updated_LO)
```

We have that $nodes(u, S) \cap nodes(v, S) \neq \emptyset$*, since both sets of nodes include* `LO`*. Clearly we also have that the sets of classes in S matched by* `LO` *in each case intersect. Hence,* r_1 *may trigger* r_2*.*

Consider next an action a which is an `INSERT` of triples along with an event e given by `INSERT` t, where t is a triple. Then a may trigger e if $triples(a, S) \cap triples(t, S) \neq \emptyset$. If some triple comprising a is of the form $(u, \texttt{rdf} : \texttt{type}, v)$, then action a may trigger an event of the form `INSERT` x `AS INSTANCE OF` y if $nodes(u, S) \cap nodes(x, S) \neq \emptyset$ and $nodes(v, S) \cap nodes(y, S) \neq \emptyset$.

`DELETE` actions can be analysed in a similar way to `INSERT` actions. An action a, given by `DELETE` v_1 `AS INSTANCE OF` c_1, may trigger an event e, given by `DELETE` v_2 `AS INSTANCE OF` c_2, if $nodes(v_1, S) \cap nodes(v_2, S) \neq \emptyset$ and $nodes(c_1, S) \cap nodes(c_2, S) \neq \emptyset$. An event e of the form `DELETE (`u`,rdf:type,`w`)` may also be triggered by a if $nodes(v_1, S) \cap nodes(u, S) \neq \emptyset$ and $nodes(c_1, S) \cap nodes(w, S) \neq \emptyset$.

Similarly, an action a that deletes triples, may trigger an event e given by `DELETE` t, where t a triple, if $triples(a, S) \cap triples(t, S) \neq \emptyset$. If some triple comprising a is of the form $(u, \texttt{rdf} : \texttt{type}, v)$, then action a may trigger an event of the form `DELETE` x `AS INSTANCE OF` y if $nodes(u, S) \cap nodes(x, S) \neq \emptyset$ and $nodes(v, S) \cap nodes(y, S) \neq \emptyset$.

Finally, considering `UPDATE` actions, an action a given by `UPDATE` $(u_1, p_1, v_{11} \rightarrow v_{12})$ may trigger an event e of the form `UPDATE` $(u_2, p_2, v_{21} \rightarrow v_{22})$ if $triples(t_{11}, S) \cap triples(t_{21}, S) \neq \emptyset$ and $triples(t_{12}, S) \cap triples(t_{22}, S) \neq \emptyset$, where $t_{11} = (u_1, p_1, v_{11})$, $t_{21} = (u_2, p_2, v_{21})$, $t_{12} = (u_1, p_1, v_{12})$ and $t_{22} = (u_2, p_2, v_{22})$.

We note that more sophisticated approaches than just using triggering relationships between rules have been also developed for determining rule termination: using activation relationships [13], using a combination of triggering and activation relationships [11], and using abstract interpretation [7]. Extending our RDFTL rule analysis to utilise these approaches is an area of future work.

4.2 Well-definedness of Action Instances

We stated earlier that instance-oriented rules are considered to be **well-defined** if the same final RDF graph will result when rule execution terminates irrespective of the order in which copies of the rule's actions part are scheduled.

A condition guaranteeing this is that:

(I1) for all possible pairs of instances U_i, U_j of the rule's action part, the sequence of updates U_i; U_j has the same effect as the sequence of updates U_j; U_i on any RDF graph;

(I2) all possible pairs of instances U_i, U_j of the rule's action part are **independent**, i.e. the queries that are evaluated following the execution of U_i are independent of the updates that are generated following the execution of U_j.

Regarding I1, if the rule's action part has no occurrences of `$delta` then I1 holds provided the rule's action part is idempotent, which is indeed the case for actions defined in our RDFTL actions syntax. If the rule's action part does have occurrences of `$delta`, then I1 holds if all actions that refer to `$delta` are either `INSERT` or `DELETE` actions. Otherwise, if there is both an `INSERT` action and a `DELETE` action that refer to `$delta`, or any `UPDATE` actions refer to `$delta` then I1 may not hold.

For condition I2 above, we need to establish whether the queries generated following the execution of one update are independent of the updates generated following the execution of another update.

Let R be a set of RDFTL rules. For an action a, we can determine, using the techniques described in Section 4.1, the rules in R that may be triggered by a. Hence, using the triggering graph, we can determine the set of all rules that may be recursively triggered by a. We denote this set of rules by $triggered(a)$.

For each rule r in $triggered(a)$, we need to find the sets of triples and nodes in the schema S whose extents may be accessed during the evaluation of r. None of these extents must be updated if independence is to hold. For each path expression p used in r, we determine $triplesQueried(p, S)$, the triples in S that correspond to the triples in the RDF graph that may be accessed during the evaluation of p. We also determine $nodesQueried(p, S)$, the nodes in S that correspond to the nodes in the RDF graph that may be accessed during the evaluation of p. These sets can be computed by evaluating p on S, as described in Section 4.1, and marking all nodes and triples that contribute to complete matches of p in S. For rule r, $triplesQueried(r, S)$ (resp. $nodesQueried(r, S)$) is the union of the sets $triplesQueried(p, S)$ (resp. $nodesQueried(p, S)$) for each path expression p in r. For action a, $triggeredTriplesQueried(a, S)$ (resp. $triggeredNodesQueried(a, S)$) is then the union of the sets $triplesQueried(r, S)$ (resp. $nodesQueried(r, S)$) for each rule r in $triggered(a)$.

In a similar fashion, we can determine $triggeredTriplesUpdated(a, S)$, the set of triples in S that correspond to triples in the RDF graph that may be inserted, deleted or updated by rules triggered by action a. We can also determine $triggeredNodesUpdated(a, S)$, the set of nodes in S that correspond to nodes in the RDF graph that may be inserted or deleted by rules triggered by action a.

Now, an action a is independent of an action b if the following four conditions hold:

$$triggeredTriplesQueried(a, S) \cap triggeredTriplesUpdated(b, S) = \emptyset$$

$$triggeredTriplesQueried(b, S) \cap triggeredTriplesUpdated(a, S) = \emptyset$$

$$triggeredNodesQueried(a, S) \cap triggeredNodesUpdated(b, S) = \emptyset$$

$$triggeredNodesQueried(b, S) \cap triggeredNodesUpdated(a, S) = \emptyset$$

Example 4 *Consider the RDF schema S in Figure 1 along with the action a given by*

```
INSERT (resource(http://www.dcs.bbk.ac.uk/LOs/BK187),dc:type,'Book')
```

and assume that the only rule that may be triggered (directly or indirectly) by a is the following:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE sl_user http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/user
LET $msgs := resource(http://www.dcs.bbk.ac.uk/users/sys)
```

```
            /target(sl_user:messages)
ON INSERT (resource(),dc:type,_)
IF True
DO INSERT ($msgs,sl_user:updated_LO,$delta.source);;
```

Then $triggeredTriplesQueried(a, S)$ *comprises the 2 triples* (`LO`, `dc` : `type`, `String`) *and* (`User`, `sl_user` : `messages`, `Messages`)*, while* $triggeredTriplesUpdated(a, S)$ *comprises the single triple* (`Messages`, `sl_user` : `updated_LO`, `LO`).

Finally, a conservative test for the independence of two instances of a rule's action part $a_1, \ldots, a_n$, i.e. for condition I2 above, is that

$$(triggeredTriplesQueried(a_1, S) \cup \ldots \cup triggeredTriplesQueried(a_n, S)) \cap$$

$$(triggeredTriplesUpdated(a_1, S) \cup \ldots \cup triggeredTriplesUpdated(a_n, S)) \;=\; \emptyset$$

and

$$(triggeredNodesQueried(a_1, S) \cup \ldots \cup triggeredNodesQueried(a_n, S)) \cap$$

$$(triggeredNodesUpdated(a_1, S) \cup \ldots \cup triggeredNodesUpdated(a_n, S)) \;=\; \emptyset$$

4.3 Rule Commutativity

Two rules r_i and r_j that may fire at the same time are said to **commute** if the same final RDF graph will result when rule execution terminates irrespective of the order in which r_i and r_j are scheduled for execution.

A condition guaranteeing this is that:

(C1) for all possible pairs of instances U_i and U_j of the action part of r_i and r_j, respectively, the sequence of updates U_i; U_j has the same effect as the sequence of updates U_j; U_i on any RDF graph;

(C2) all possible pairs of instances U_i, U_j of the action part of r_i and r_j are independent i.e. the queries that are evaluated following the execution of U_i are independent of the updates that are generated following the execution of U_j, and vice versa.

Regarding C1, let $triplesInserted(r, S)$, $triplesDeleted(r, S)$ and $triplesUpdated(r, S)$ be the sets of triples in schema S whose extents may be subject to `INSERT`, `DELETE` and `UPDATE` actions, respectively, by the action part of a rule r, as well as by the action part of any rule that may be triggered by r. In addition, let $triplesModified(r, S)$ be

$$triplesInserted(r, S) \cup triplesDeleted(r, S) \cup triplesUpdated(r, S)$$

Then, assuming that all update actions are to triples, a conservative test for C1 is that

$$triplesInserted(r_i, S) \cap triplesDeleted(r_j, S) = \emptyset$$

$$triplesDeleted(r_i, S) \cap triplesInserted(r_j, S) = \emptyset$$

$$triplesUpdated(r_i, S) \cap triplesModified(r_j, S) = \emptyset$$

and

$$triplesModified(r_i, S) \cap triplesUpdated(r_j, S) = \emptyset$$

A similar test can be made for updates to nodes and for action parts that are combinations of updates to triples and nodes.

Regarding condition C2, if $a_1, \ldots, a_n$ is the action part of r_i and $b_1, \ldots, b_m$ is the action part of r_j, then a conservative test for C2 is that

$$(triggeredTriplesQueried(a_1, S) \cup \ldots \cup triggeredTriplesQueried(a_n, S)) \cap$$

$$(triggeredTriplesUpdated(b_1, S) \cup \ldots \cup triggeredTriplesUpdated(b_m, S)) \ = \ \emptyset$$

and

$$(triggeredTriplesQueried(b_1, S) \cup \ldots \cup triggeredTriplesQueried(b_m, S)) \cap$$

$$(triggeredTriplesUpdated(a_1, S) \cup \ldots \cup triggeredTriplesUpdated(a_n, S)) \ = \ \emptyset$$

once again assuming that updates involve only triples. The test can easily be extended to include updates to nodes as well.

5 RDFTL Rules in P2P Environments

We have implemented a system for processing RDFTL rules in P2P environments. We discuss our implementation further in Section 6 below. The rule processing functionality in our system is provided by a set of services that constitute the *RDFTL ECA Engine*. This set of services acts as an 'active' wrapper over a distributed set of 'passive' RDF/S repositories, exploiting their query, storage and update functionality.

Our system is an example of a schema-based P2P system, having been inspired by the superpeer-based architecture of Edutella [44]. Other schema-based P2P systems include ICS FORTH SQPeer [38] and Piazza [51]. Similar to our system, the metadata distribution in an Edutella network allows hybrid fragmentation with possible replication between peers. SQPeer also does not impose any particular data fragmentation or replication policy; each superpeer integrates a set of peers that support the same schema, although a peer may belong to more than one peergroup if it supports more than one schema; this is contrast to our approach where each peer is connected to one superpeer only and all peers support the same RDF schema. Piazza focuses on the semantic integration and global querying of heterogenous data distributed over a P2P network, where each peer supports its own schema.

The architecture of our system is illustrated in Figure 2. Each superpeer shown in that figure may be supervising a group of further peers, which we term its peergroup, as well as itself hosting a fragment of the global RDF/S descriptions in the network. At each superpeer there is one ECA Engine installed. We assume that each peer or superpeer hosts a fragment of an overall global RDFS schema, and that each superpeer's RDFS schema is a superset of its peergroup's individual RDFS schemas. Although this is sufficient for the SeLeNe project, in general superpeers, and indeed peers, may hold heterogeneous RDFS schemas, and there is a need for an RDFS schema mapping service. The techniques discussed in [24, 43] could be used as the basis for such a service, and this is an area of future work.

The fragment of the global RDFS schema stored at a peer may change as a result of changes in the peers' RDF/S descriptions. Peers notify their supervising superpeer of any updates to their local RDF/S repository. Peers may dynamically join or leave the network.

Each superpeer defines access privileges over the classes and properties in its RDFS schema. These privileges may be read-only, read-write or private, describing the corresponding access level to the instances of each class and property. More fine-grained

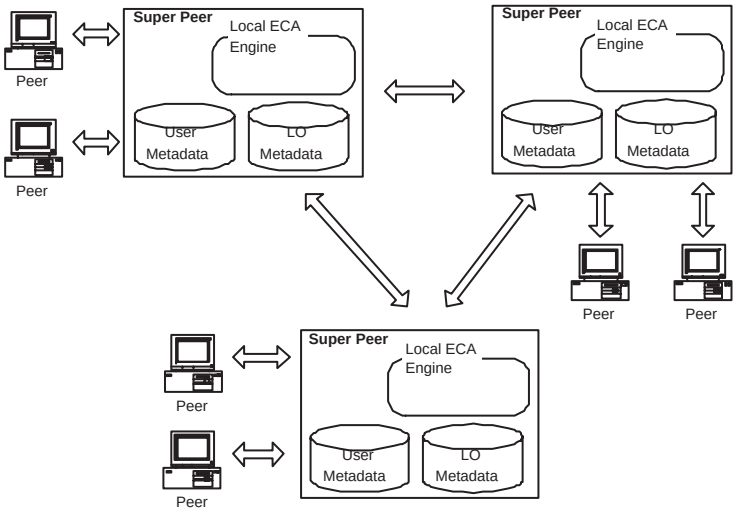

Figure 2: P2P System Architecture

access privileges are also allowed on specific RDF resources and properties. These facilities allow a superpeer to specify which metadata can be shared with other superpeers outside its peergroup.

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. An ECA rule generated at one site of the network might be triggered, evaluated, and executed at different sites. 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.

Whenever a new ECA rule r is generated at a peer P, it will be sent to P's superpeer for storage. From there, r will also be forwarded to all other superpeers, and a replica of it will be stored at those superpeers where an event may occur that may trigger r's event part, i.e. those superpeers that are **e-relevant** to r (see below). A rule r has a globally unique identifier of the form $SP_i.j$, where SP_i is the originating superpeer identifier and j a locally unique identifier for the rule in SP_i's rule base.

We assume that at run-time rules are triggered by events occurring within a single peer's local RDF repository. We also assume that each particular copy of a rule's action part executes within a single peer's RDF repository. If there is a need to distribute a sequence of updates across a number of peers in reaction to some event, then rather than specifying one rule of the form

$$\texttt{on } e \texttt{ if } c \texttt{ do } a_1, \ldots, a_n$$

instead, n rules $r_1, \ldots, r_n$ can be specified, where each r_i is `on` e `if` c `do` a_i and r_1 *prec* r_2 *prec* ... *prec* r_n. Note that it is also possible to relax the total ordering of $r_1, \ldots, r_n$ into a partial ordering, or no ordering at all. However, there is still the limitation that a copy of each a_i will only execute on one peer.

In summary, we assume that there is no need for distributed event detection or distributed update execution (although the evaluation of rules' condition parts may be distributed). These assumptions hold true for the SeLeNe system, but generalising our techniques and architure to support distributed event detection and distributed update

execution are areas of future work.

Given an RDF schema S and an RDFTL rule r, we now define what it means for r to be relevant to S. There are three types of relevance:

- r is **e-relevant** to S if each of the path expressions that either appear in the event part of r or are used by the event part through variable references, can be evaluated on S, i.e., each step in each path expression exists in S. (This is because we assume that there is no distributed event detection.)
- r is **c-relevant** to S if some step in one of the path expressions referenced by the condition part of r can be evaluated on S. (This is because we assume that conditions may be evaluated at multiple sites.)
- r is **a-relevant** to S if all actions in the action part of r are **a-relevant** to S. (This is because we assume that there is no distributed update execution.) An individual action is a-relevant to S if it satisfies one of the following:
  - If it is a deletion or insertion of resources that uses `AS INSTANCE OF` *`class`*, then *`class`* must be in S.
  - If it is a deletion of resources that does not use `AS INSTANCE OF` *`class`*, then we determine the most specific class of resources that the path expression in the deletion would return. This class must be in S.
  - If it is an action over triples that uses a property p, then p must be in S. If it is a deletion of triples that uses the wildcard '_' instead of a property (the only action allowed to do this), then the classes of resources returned by the path expressions involved in the deletion must exist in S. Note that use of the wildcard '_' instead of the source or target node of a triple would return all resources.

We say that a peer or superpeer is e-relevant, c-relevant or a-relevant to a rule r if r is e-, c- or a-relevant, respectively, to the peer or superpeer's RDFS schema.

The ECA engine at a superpeer, SP, provides several services:

- The *Rule Registration Service* takes as input an RDFTL rule definition, generated by some peer in SP's peergroup, and registers it in SP's Rule Base. It invokes an *RDFTL Language Interpreter* to verify the syntactic correctness of the rule and to translate any path queries and updates within the rule into the query and update syntax of the underlying RDF repository.
- The *Event Handler* handles the occurrence of events within the RDF repositories of SP's peergroup, and the triggering of rules registered within SP's rule base. The Event Handler determines which rules have actually been triggered by an update to a local repository by invoking that repository's 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. This may require distributed query processing across a number of peers and superpeers, invoking their local query services.
- The *Action Scheduler* generates from the action parts of rules that have fired a list of updates to be considered for execution at SP and also to be sent to all other superpeers over the network. At any superpeer, whether these updates are added to the local execution schedule or not depends on whether:
  (a) they are *a-relevant* to the local schema, and
  (b) the local schema allows read-write privileges to all the resources affected by the updates.

- The *Routing Service* keeps a list of the immediate neighbours of SP (peers in its peergroup and other superpeers) and hence maintains the message transmission paths in the network.

5.1 Registering a new ECA rule

Whenever a new ECA rule r is registered at a peer P, it is sent to P's supervising superpeer for syntax validation, translation into the local repository's query and update language, and storage. ¿From there, r will also be sent to all other superpeers, and a replica of it will be stored at those superpeers that are e-relevant to r.

Determining the e-, c- and a-relevance of a particular ECA rule to a superpeer involves comparing the path expression(s) used by that part of the rule against the superpeer's RDFS schema. In order to aid this comparison, an *index* can be kept at each superpeer. There are a number of possibilities for doing this [25, 45, 39, 27] and we describe our approach in Section 6.

Using the *Routing Service*, a new rule is propagated to all superpeers of the network and it is stored at those superpeers that are e-relevant to it. Each such superpeer matches each part of the rule against its index, and annotates the event, condition and action parts of the rule with the IDs of local peers which may be affected by each part of the rule.

Each superpeer SP_i is responsible for specifying the precedence relationship $prec_i$ between the rules generated by itself or its local peergroup. As rules are propagated from superpeer to superpeer, local decisions are made at each superpeer regarding the precedence of the rules originating from other superpeers compared with its own rules, and a local precedence scheme is applied (e.g. timestamp order, assigning higher priority to local rules, assigning higher priority to more specific rules, or combinations thereof).

Changes in a superpeer's index (caused by changes in its peergroup's or its own RDF/S metadata) require the annotations of each part of each rule in its rule base to be updated. Any rules that are no longer e-relevant to the superpeer can be deactivated. Conversely, if a superpeer's RDFS schema changes from having no metadata associated with a particular schema node to now having such metadata, this change is notified to SP's neighbouring superpeers. If any of these neighbours have ECA rules which may have been made e-relevant by the new change at SP, they send these ECA rules to SP. These superpeers also request from their neighbours (other than SP) their current set of ECA rules which are potentially e-relevant to the change, and they forward these rules on to SP. This process repeats until all the potentially e-relevant ECA rules throughout the network have been sent to SP.

5.2 P2P Rule Execution

In a P2P environment, the RDF graph is partitioned amongst the peers and each superpeer manages its own local execution schedule.

Each local schedule at a superpeer is a sequence of updates constituting fragments of global transactions which are to be executed on the fragment of the global RDF graph which is stored at the superpeer or its local peergroup.

Each superpeer coordinates the execution of transactions that are initiated by that superpeer, or by any peer in its local peergroup.

Whenever an update u is executed at a peer P, P will notify its supervising superpeer SP. SP will determine whether u may trigger any ECA rule whose event part is annotated with P's ID. If a rule r may have been triggered, then SP will send r's event query to P 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 distributed evaluation of the condition is coordinated by SP's Condition Evaluator.

If a condition evaluates to true, SP will send each instance of r's action part (there will be only one instance if r is a set-oriented rule, and one or more instances if r is an instance-oriented rule) to its local peers, according to the annotations on r's action part made during r's registration. The instances of r's actions part will also be sent to all neighbouring superpeers and from there in turn to all other superpeers of the network. All superpeers that are a-relevant to r will consult the access privileges on their metadata in order to decide whether the updates they have received can be scheduled and executed on their local peergroup.

In summary therefore, local execution of the update at the head of a local schedule may cause events to occur. These events may cause rules to fire, modifying the local schedule or remote schedules with new subtransactions to be executed.

Because of the assumption that instance-oriented rules are well-defined, if different instances of an instance-oriented rule's action part are executed by different superpeers, then the order of execution of these different instances is immaterial and the coordinating superpeer does not have to enforce any particular ordering. Moreover, the resulting subtransactions do not have to be executed in isolation from each other.

Similarly, because of the commutativity assumption for rules that have the same precedence, the coordinating superpeer does not have to enforce any particular order of execution of such rules and the resulting subtransactions do not have to be executed in isolation from each other.

6 Implementation

Our system implements a set of services over the JXTA P2P framework [33]. JXTA is a platform-independent framework, containing a set of open protocols that enable service-based communication in a P2P manner, between any kind of device in a network. JXTA provides a full range of services necessary for building a peer-to-peer network. These services include peer discovery, message exchange, resource sharing, security and authentication. In our system we have exploited this functionality and have built our system-specific services using the JXTA API.

Each peer of our P2P network implements a set of core services that provide the basic functionality necessary to participate in the system: local RDF/S event detection, local RDF/S indexing, RDF repository connectivity, and messaging. In order for a host to become a member of the network, it has to support this set of core services. Superpeers in addition support RDFTL rule registration and rule processing, superpeer RDF/S indexing, and manage connectivity with the peers in their peergroup as well as with their neighbouring superpeers.

Our implementation of these services is flexible, allowing a peer to dynamically extend its set of services and become a superpeer, or a superpeer to dynamically shed its extra services and become a simple peer. Below we describe the most important services, and their role in the overall operation of the system.

6.1 Core Services

Event Detection Service. This is responsible for detecting metadata modification events at a peer. It notifies the *Event Handler* service at its superpeer of each event

occurrence, including the type of the event, the metadata affected and the time the event occurred. It also notifies its own *Peer Indexing* service to update its indexes according to the changes, if necessary.

Peer Indexing Service. This maintains indexes on the RDF metadata stored at the peer and provides a simple query interface over these indexes. After notification of a metadata change by the Event Detection service, the Peer Indexing service updates, if necessary, the local peer indexes. If necessary, it also sends a notification to the Superpeer Indexing Service (see below) at its supervising superpeer in order for it to update its superpeer indexes.

In more detail, as the RDF metadata stored at a peer P change over time, P maintains an RDFS schema which shows for each node in the schema whether or not there is RDF metadata of this type stored at P (a '0' or '1' bit). This information is also propagated to P's supervising superpeer SP, which maintains a combined RDFS schema annotated so that each node shows the set of peers in its peergroup that manage metadata of this type (a set of peer IDs). Over time, the metadata stored at P may change so that its RDFS schema 'shrinks' (i.e. one or more '1' annotations become '0' annotations) or 'grows' (i.e. one or more '0' annotations become '1' annotations) and such changes are also propagated to SP.

As well as this annotated RDFS schema, each peer also keeps for each node annotated with a '1' in this schema a list of the RDF resources of this type that its RDF metadata references — we call these lists of RDF resources the *resource index*. Each peer also keeps a list of the properties that its RDF metadata references — which we call the *property index*. Each superpeer keeps a consolidated resource index and property index for its entire peergroup.

Repository Connection Service. This manages the connection with the underlying RDF repository. It consists of three subcomponents: the *Connection Manager* that provides connection pooling, the *Update Manager* that interprets and passes RDF metadata update requests to the repository, and the *Query Manager* that establishes communication with the query engine of the repository and retrieves query results. In the current version of our system we are using ICS-FORTH RSSDB [6] as the RDF repository. For the future we plan also to support Jena2 [32].

Messaging Service. This is responsible for all message exchange between the peer and its supervising superpeer. It undertakes to wrap or unwrap the outgoing or incoming messages, respectively, and pass them to the appropriate service.

6.2 Superpeer Services

RDFTL Rule Processing Services. This set of services includes the *Event Handler*, *Condition Evaluator* and *Action Scheduler* already mentioned earlier.

The *Event Handler* is passed information concerning an event occurrence by the Event Detection service each time that an update takes place at some peer of the peergroup. Using this information, the Event Handler contacts the *Rule Base Indexer* (see below) to retrieve all rules that may be triggered by the event. The Event Handler then determines which rules have actually been triggered by the event by invoking the peer's Query Manager to evaluate the event queries of rules that may have been triggered.

The *Condition Evaluator* evaluates the condition part of rules that have been triggered. Our current implementation assumes that all conditions can be evaluated within the local peergroup and does not support distributed query processing across a number of superpeers. Subqueries of the condition part of a triggered rule are dispatched to the appropriate peers in the peergroup for evaluation by their Query Managers. A super-

peer can use the annotations on a rule's condition part to determine to which local peers subqueries of the condition should be dispatched for evaluation. If the `$delta` variable is present in the condition, it will have been instantiated and the superpeers' indexes can be consulted for more precise information about which local peers are relevant to subqueries of the instantiated condition. The subquery results are subsequently merged by the Condition Evaluator.

The *Action Scheduler* generates from the action parts of rules that have fired a list of updates to be considered for execution at the superpeer. A copy of this list of updates is also sent, using the *Routing Service*, to all neighbouring superpeers and from there in turn to all other superpeers.

The Action Scheduler maintains the local execution schedule where all updates scheduled for local execution are placed. Each time an update reaches the head of this schedule, the Action Scheduler consults the peer ID annotations on the update and dispatches the update to the appropriate peers within the peergroup for execution.

In the current version of the system, no transaction management has been implemented. So each time the Action Scheduler dispatches an update for execution somewhere in the network, we assume that this execution is successful. Future versions of the system will, however, support transaction management as discussed in Section 6.3 below. To detect possibly non-terminating rule executions, a maximum number of recursive rule firings is allowed.

Rule Base Management Services. This set of services is dedicated to maintaining the local rule base, including indexing of its contents and providing simple query and update functionality over it.

The *Rule Registration* service receives a new RDFTL rule and undertakes to register it in the rule base. The *RDFTL Language Interpreter* is first invoked to translate RDFTL path expressions to the corresponding query expressions of the underlying RDF repository, and RDFTL rule actions to update-API function calls. The *Superpeer Indexing* service is then contacted in order to construct the list of peers that are affected by each part of the rule, using the annotated superpeer RDFS schema and the consolidated resource and property indexes at the superpeer. In particular, let i be a peer ID and let S_i denote the subgraph of the superpeer schema S induced by nodes whose annotation includes i. The event part of a new rule r is annotated with peer ID i if r is e-relevant to S_i, any property mentioned in the event part of r is in i's property index and any resource mentioned in the event part of r is in i's resource index. The condition part of r is annotated with i if r is c-relevant to S_i. The action part of r is annotated with i if r is a-relevant to S_i, every property mentioned in the action part of r is in i's property index and every resource mentioned in the action part of r is in i's resource index. The annotated rule is then stored in the rule base.

The translated, but not yet annotated rule, is also sent to the neighbouring superpeers, using the Routing Service, and from there to all superpeers in the network. It will be stored in the rule base of all superpeers that are e-relevant to it, after it has been appropriately annotated.

The *Rule Base Indexer* creates and maintains rule-specific indexes on the contents of a rule base, aiming to speed up the discovery of rules that may be triggered by an event. Whenever the rule base is updated (i.e. a rule is added or deactivated) this service undertakes to perform the appropriate updates to the rule indexes.

Routing Service. This keeps a list of the neighbouring peers and superpeers in order to maintain the message transmission paths in the network. This service is called each time that another service of the superpeer needs to transmit a message to one or more of its neighbouring peers or superpeers.

Superpeer Indexing Service. This is responsible for the creation and main-

tenance of the consolidated indexes operating at the peergroup level, including the combined RDFS schema and the consolidated resource and property indexes. Each time a change occurs to a peer's RDF metadata, this service is notified, if necessary, by the corresponding Peer Indexing service in order to update also these peergroup-level indexes.

6.3 Concurrency Control and Recovery

Our current implementation does not yet support any concurrency control or recovery mechanisms, but we briefly discuss here possible implementations of such mechansisms.

In theory, any distributed concurrency control protocol could be adapted to a P2P environment. For example, the AMOR system adopts optimistic concurrency control [30]. The serialisation graph is distributed amongst those peers responsible for transaction coordination, which are analogous to our superpeers. The AMOR system assumes that conflicts are only possible between those transactions that are accessing a particular 'region' of resources (analogous to our peers) and thus subgraphs of the global serialisation graph are stored and replicated amongst those coordinators which service a particular region. However, the regions are not static and these subgraphs are dynamically merged and replicated as transactions execute and regions evolve. We could use similar techniques to merge and replicate subgraphs of the global serialisation graph between our superpeers.

In the classical approach to distributed transactions, global transactions hold on to the resources necessary to achieve their ACID (Atomicity, Consistency, Isolation and Durability) properties until such time as the whole transaction commits or aborts. In a P2P environment this may not be feasible: the resources available at peers may be limited, peers may not wish to cooperate in the execution of global transactions, and peers may disconnect at any time from the network, including during the execution of a global transaction in which they are participating. The cascaded triggering and execution of ECA rules will cause longer-running transactions which may further exacerbate these problems. It is therefore necessary to relax the Atomicity and Isolation properties of transactions.

In particular, subtransactions executing at different peers may be allowed to commit or abort independently of their parent transaction committing or aborting, and parent transactions may be able to commit even if some of their subtransactions have failed. Subtransactions that have committed ahead of their parent transaction committing can be reversed, if necessary, by executing compensating subtransactions [28, 40]. These are generated as transactions execute and they reverse the effects of a transaction by compensating each of the transaction's updates in reverse order of their execution. Generating compensating updates is straight-forward for RDFTL updates: the insertion of a triple is reversed by deletion of the triple, the deletion of a triple by an insertion, and an update by the restoration of the original value. If transactions have read from committed (sub)transactions which are subsequently reversed, then a cascade of compensations will result.

We assume as the default that a parent transaction (or subtransaction) and its immediate subtransactions are able to commit independently of each other, and so an explicit **abort dependency** now needs to be specified for each rule. The possible abort dependencies are as follows, with T_o being the parent (sub)transaction and T_r the child subtransaction:

- **ParentChild**: If T_o aborts then T_r is to abort.
- **ChildParent**: If T_r aborts then T_o is to abort.

- **Mutual**: If either T_o or T_r aborts then so must the other.
- **Independent**: There is no abort dependency between T_o and T_r.

For coordinating the execution of compensating transactions or subtransactions, an abort graph can be maintained that describes the abort dependencies between parent transactions and their subtransactions.

The abort graph will be distributed amongst the superpeers that participate in any subtransaction of a top-level transaction. The graph will be constructed dynamically with each new subtransaction. In particular, each time a transaction T_n at a superpeer SP_i initiates a new subtransaction T_m to be executed at a superpeer SP_j (where it may be that $i = j$) then depending on the abort dependency between T_n and T_m, the following actions are taken:

1. **ParentChild**: The identifier of T_m and the superpeer SP_j that it will execute on are transmitted to SP_i and recorded there, together with an arc $T_n \rightarrow T_m$ in the local abort graph at SP_i.
2. **ChildParent**: The identifier of T_n and the superpeer SP_i that it is executing on are transmitted to SP_j and recorded there, together with an arc $T_m \rightarrow T_n$ in the local abort graph at SP_j.
3. **Mutual**: A combination of the actions for **ParentChild** and **ChildParent** above is taken.
4. **Independent**: No local abort graph is updated.

Using the above, in case of a subtransaction failure all the necessary information is available in order to initiate a compensating subtransaction, at any level of nesting of the subtransaction.

Figure 3 gives an example of a distributed abort graph. In this figure, a failure in subtransaction T_7 at SP_3 leads to compensation of T_7 at SP_3 but leaves the rest of the transaction unaffected, while a failure in subtransaction T_6 at SP_5 initiates a compensating transaction for T_6 at SP_5, a compensating transaction for T_1 at SP_2 (due to the **Mutual** abort dependency between T_6 and T_1), and a compensating transaction for T_5 at SP_2 (due to the **ParentChild** dependency between T_1 and T_5).

7 Concluding remarks

In this paper we have described the RDFTL language for defining ECA rules on RDF metadata, including its syntax and execution semantics. To our knowledge, this is the first ECA rule language developed specifically for RDF/S. We have developed conservative tests for determining the termination and confluence of RDFTL rules. We have described an architecture for supporting RDFTL rules in P2P environments, and have described an implementation of this architecture. We have also discussed techniques for relaxing the isolation and atomicity requirements of transactions. We are currently evaluating the performance of our system in the context of the SeLeNe application domain, including the development and validation of an analytical performance model.

For the future, we plan to explore more deeply the expressiveness of RDFTL — it is straight-forward to show that RDFTL is computationally complete but we wish to investigate also its query and update expressiveness. We also plan to implement the transaction management techniques discussed here, and to evaluate their effects on system performance.

There are as yet no generally accepted query or update languages for RDF/S. A survey of current RDF query language proposals can be found in [29]. Since the design

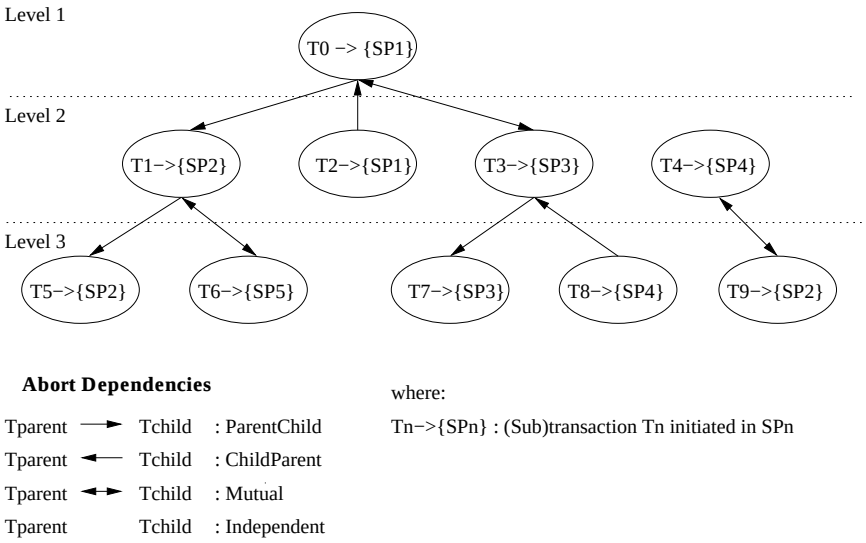

Figure 3: Abort graph example

of our RDFTL language, proposals for RDF update languages have been described [42, 50]. If ECA rules are to be supported on RDF/S repositories, then whatever query and update languages eventually emerge for RDF/S, there is also the parallel issue of designing the event, condition and action sub-languages for ECA rules. In this paper we have shown how this was done in the context of our particular 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.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: peer-to-peer data and web services integration. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 1087–1090, 2002.

[2] S. Abiteboul, S. Cluet, G. Ferran, and M.-C. Rousset. The Xyleme project. *Computer Networks*, 39:225–238, 2002.

[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 Data Bases*, pages 643–645, 2000.

[4] A. Aiken, J. Widom, and J. M. Hellerstein. Behaviour of database production rules: Termination, confluence and observable determinism. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 59–68. ACM Press, 1992.

[5] A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM TODS*, 20(1):3–41, 1995.

[6] 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*, 2001.

[7] J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proc. 7th Int. Workshop on Database Programming Languages, LNCS 1949*, pages 249–266, Kinloch Rannoch, Scotland, 1999.

[8] J. Bailey, A. Poulovassilis, and P. Newson. A dynamic approach to termination analysis for active database rules. In *Proc. 1st Int. Conf. on Computational Logic (DOOD stream), LNCS 1861*, pages 1106–1120, London, 2000.

[9] J. Bailey, A. Poulovassilis, and P. Wood. An Event-Condition-Action Language for XML. In *Proc. 11th Int. Conf. on the World Wide Web*, pages 486–495, 2002.

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

[11] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems, LNCS 985*, pages 165–181. Springer, 1995.

[12] E. Baralis, S. Ceri, and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):353–370, 1998.

[13] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 475–486, Santiago, Chile, 1994.

[14] E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM TODS*, 25(3):269–332, 2000.

[15] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. 18th Int. Conf. on Data Engineering*, pages 403–418, 2002.

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

[17] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. In *Proc. 10th Int. Conf. on the World Wide Web*, pages 633–641, 2001.

[18] A. Buchmann et al. DREAM : Distributed Reliable Event-Based Application Management. In M. Levene and A. Poulovassilis, editors, *Web Dynamics*, pages 319–352. Springer, 2004.

[19] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 254–262, 2000.

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

[21] 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 Data Bases*, pages 615–626, 1999.

[22] P.-A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/subscribe for RDF-based P2P networks. In *Proc. ESWS 2004, Heraklion, Crete*, pages 182–197, 2004.

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

[24] V. Christophides et al. The ICS-FORTH SWIM: A powerful semantic web integration middleware. In *Proc. SWDB 2003*, pages 381–393, 2003.

[25] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proc. ICDCS 2002, Vienna*, pages 23–34, 2002.

[26] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[27] L. Galanis, Y. Wang, S. R. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 874–885, 2003.

[28] H. Garcia-Molina and H. Salem. Sagas. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 249–259, 1987.

[29] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of RDF query languages. In *Proc. 3rd Int. Semantic Web Conference, Hiroshima*, pages 502–517, 2004.

[30] K. Haller, H. Schuldt, and H. Schek. Transctional peer-to-peer information processing: The AMOR approach. In *4th Int. Conf. on Mobile Data Management*, pages 356–362. Springer, 2003.

[31] J. Jacob, A. Sanka, N. Pandrangi, and S. Chakravarthy. WebVigil: An approach to just-in-time information propagation in large network-centric environments. In M. Levene and A. Poulovassilis, editors, *Web Dynamics*, pages 301–318. Springer, 2004.

[32] Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net/.

[33] JXTA Framework. http://www.jxta.org/.

[34] V. Kantere, I. Kiringa, J. Mylopoulos, A. Kemenstiestides, and M. Arenas. Coordinating peer databases using ECA rules. In *Proc. DBISP2P 2003*, pages 108–133, 2003.

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

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

[37] S. Kokkelink. Transforming RDF with RDFPath. `http://zoe.mathematik.uni-osnabrueck.de/QAT/Transform/RDFTransform.pdf`, March 2001.

[38] G. Kokkinidis and V. Christophides. Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In *Proc. EDBT Workshops*, pages 486–495, 2004.

[39] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proc. EDBT'03*, pages 29–47, 2003.

[40] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. *VLDB Journal*, pages 95–106, 1990.

[41] A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, commercial experience and beyond. In N. Paton, editor, *Active Rules in Database Systems*, pages 367–404. Springer, 1999.

[42] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. RUL: A declarative update language for RDF. In *Proc. 4th Int. Semantic Web Conference (ISWC'05), Galway, Ireland*, 2005.

[43] P. McBrien and A. Poulovassilis. Defining peer-to-peer integration using Both As View rules. In *Proc. DBISP2P 2003*, pages 91–107, 2003.

[44] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *Proc. 11th Int. Conf. on the World Wide Web*, pages 604–615, 2002.

[45] 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. 12th Int. Conf. on the World Wide Web*, pages 536–543, 2003.

[46] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 437–448, 2001.

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

[48] 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.

[49] REWERSE Project. Deliverable i5-d1: State-of-the-art on evolution and reactivity. `http://rewerse.net/deliverables/i5-d1.pdf`.

[50] A. Souzis. RxPath specification proposal. `http://rx4rdf.liminalzone.org/RxPathSpec`.

[51] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza Peer Data Management Project. *SIGMOD Record*, 32(3):47–52, 2003.

[52] M.-R. Tazari. A context-oriented RDF database. In *Proc. SWDB 2003*, pages 63–78, 2003.

[53] W3C. XML Path Language (XPath), 1999.

[54] W3C. RDF Semantics, W3C Recommendation 10 February 2004, 2004.

[55] W3C. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004, 2004.

[56] W3C. RDF/XML Syntax Specification, W3C Recommendation 10 February 2004, 2004.

[57] G. Wagner. How to design a general rule markup language? In *Invited talk at the Workshop XML Technologien für das Semantic Web (XSW 2002)*, pages 19–37, Berlin, June 2002.

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