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

George Papamarkos, Alexandra Poulovassilis, Peter T. Wood
{gpapa05,ap,ptw}@dcs.bbk.ac.uk

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 also develop conservative tests for determining when the order of execution of pairs of rules, or instances of the same rule, is immaterial to the final state of the RDF metadata after rule execution terminates. We describe an architecture for supporting such rules in P2P environments, and 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 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

<p align="center">on event <code>if</code> condition <code>do</code> actions</p>

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 [Pat99] for a general discussion of ECA rules in databases. There are several advantages in using ECA rules to implement this kind of functionality, and again we refer the reader to [Pat99] for a general discussion of these. 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 been motivated by our work in the SeLeNe project (see `http://www.dcs.bbk.ac.uk/selene/`). The aim 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, 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, and it is with this scenario that we are concerned in this paper. SeLeNe's reactive functionality includes features such as: automatic propagation of changes in the description of one resource to the descriptions of other, related resources; automatic notification to users of the registration of new LOs of interest to them; and automatic notification of changes in the description of resources of interest to users.

Outline of this paper: Section 2 describes RDFTL (RDF Triggering Language), an event-condition-action rule language providing reactive functionality over RDF metadata stored in RDF repositories. As well as describing the language, we also develop conservative tests for determining when the order of execution of pairs of rules, or instances of the same rule, is immaterial to the final state of the RDF metadata after rule execution terminates. Section 3 the describes our architecture for supporting RDFTL in P2P environments. We describe how rules are registered at peers and propagated through the network. We also discuss rule execution in P2P environments, including techniques for relaxing the isolation and atomicity requirements of transactions. We give our concluding remarks in Section 4.

2 The RDFTL Language

2.1 RDFTL Syntax

RDFTL operates over RDF graphs and thus complies with current RDF standards of syntax, semantics and datatypes. RDFTL assumes that RDF graphs conform to one or more RDFS schemas, in the sense that:

(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.

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, i is a number and s is a string:

$e ::= "resource("uri")" ("/"p)?$

$p ::= p"/"p \mid p"["q"]" \mid "target("arc\_name")" \mid "source("arc\_name")" \mid$
$\qquad "element("i")" \mid "element()"$

$q ::= q "and" q \mid q "or" q \mid "not" q \mid p \mid p " = " s \mid p" \neq " s$

The above syntax is similar to that for XPath [W3C99], except for the following. $resource(uri)$ 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, while $source(arc\_name)$ returns the set of *subject* nodes related by the predicate arc_name to the set of *object* nodes given by the context. $element(i)$ is used to return the ith element of a collection, while $element()$ returns all elements of a collection. A denotational semantics for the language is given in [PPW04].

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 described 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 the *class*, if specified[1]. 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.

---

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

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

3. UPDATE `upd_triple`

   This detects updates of arcs. *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 `$delta` has as its set of instantiations the triples which have triggered the rule. The individual components of one these triples can be obtained by `$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 and 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, according to the RDFS 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.

The triples in the case of arc manipulation have the same form as in the event sublanguage. 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 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 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.

2.2 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 rules as defined in Section 2.1 above. In Section 2.1 we also specified for each kind of event when a rule is considered to have been triggered, and what is its set of instantiations for the `$delta` variable.

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. The position at which it is added to the schedule depends on the rule's **coupling mode**, which must be specified for each rule. The coupling mode may be one of the following:

- ○ **Immediate**. In this case, the copy of the rule's action part is executed before the rest of T executes, as a subtransaction of T.
- ○ **Deferred**. In this case, the copy of the rule's action part is executed after the rest of T executes, again as a subtransaction of T.
- ○ **Detached**. In this case, the copy of the rule's action part is scheduled for execution independently of T, as a new transaction.

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`. Again, the position on the schedule where these multiple instances of a rule's action part are added depends on the rule's coupling mode. 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, up to ordering of elements in bag and set collections, irrespective of the order in which instances of the rule's actions part are scheduled See Section 2.4 below for a discussion of conservative tests for verifying this property for RDFTL rules.

It is in general possible that many rules with the same coupling mode 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 with the same coupling mode 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 with the same coupling mode 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, up to ordering of elements in bag and set collections, irrespective of the order of scheduling of r_i and r_j See Section 2.5 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[2].

2.3 Transaction Dependencies

Each time a copy of a rule's actions part is added to the schedule, a new subtransaction of the current transaction (in the case of Immediate or Deferred coupling modes) or a new transaction (in the case of Detached coupling mode) is generated.

In the former case, a failure in the subtransaction will necessitate rolling back the entire current transaction. In the latter case, there are a number of alternative **abort dependencies** between the current transaction, T_o, and the new transaction, T_r:

- ○ **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.

The abort dependency needs to be specified for each rule that has Detached mode.

---

[2] A detailed description of the general coupling and binding possibilities for ECA rules can be found in [Pat99].

This raises the question of how to reverse, if necessary, transactions that have already committed. This can be achieved by means of compensating transactions [GMS87,KLS90]. 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.

There is also the question of what happens to transactions that have read from committed transactions which subsequently need to be reversed. One solution is to use the techniques similar to those discussed in Sections 2.4 and 2.5 below to allow this only for transactions which commute with the committed transaction so that they do not need to be reversed if the committed transaction is reversed. Another possibility is to carry out a cascade of compensations.

2.4 Well-definedness of Action Instances

We stated earlier that instance-oriented rules are **well-defined** if the same final RDF graph will result when rule execution terminates, up to ordering of elements in bag and set collections, 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, up to ordering of elements in bag and set collections.

(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 `$delta` then I1 holds. If the rule's action part does have occurrences of `$delta`, then I1 holds if all actions that refer to `$delta` are either `INSERT` or `DELETE` actions, and moreover if none of them may insert or delete elements of list collections. Otherwise, i.e. if any `UPDATE` actions refer to `$delta` or if elements may be added to or deleted from list collections, then I1 may not hold.

We next discuss a conservative test guaranteeing I2 above for an RDFTL rule. We start by defining sets of RDFS schema nodes and triples that are 'matched' by RDFTL rule actions and events. Throughout we assume that we always work with the *closure* of the RDF graph and of the RDFS schema, as defined in [W3C04]. This means, for example, that when a resource s is inserted into class c (i.e., the triple $(s, \mathtt{rdf : type}, c)$ is inserted), s is also inserted into all superclasses of c. Likewise when triple (u, p, v) is inserted into an RDF graph, a triple (u, q, v), where p is a subproperty of q, is also inserted for all superproperties q.

For simplicity in the following definitions (and without loss of generality), we assume that each *source_node*, *target_node* and path expression in an

event or action of an RDTFL rule is a simple variable, defined by a *let-expression*. 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 instances would be returned by evaluating v on graph G. Note that if $nodes(v, S)$ contains class c, then it will also contain all superclasses of c, except that we assume that it does not contain the class `rdfs:Resource`.

For example, for an action or event t given by either `INSERT` v `AS INSTANCE OF` c or `DELETE` v `AS INSTANCE OF` c (we leave out the *let-expressions* that provide the definition of v), $nodes(v, S)$ includes c and all superclasses of c (apart from `rdfs:Resource`).

Similarly, for each triple $t = (u, p, v)$ used in a rule, where u and v are variables or the wildcard '_' and p is a property (arc name) or '_', we can identify the set $triples(t, S)$ of triples in S whose instances would be returned by 'evaluating' t on G. When used as the first or third component of a triple, the wildcard '_' matches any node in S; when used as the second component, it matches any property name in S. 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,

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

A rule r_i **may trigger** a rule r_j if some action of r_i may generate an event which triggers r_j. The **triggering graph** 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.

Consider 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. Then a may generate e if $nodes(v_1, S) \cap nodes(v_2, S) \neq \emptyset$. Action a may also generate an event e of the form `INSERT` $(u,$ `rdf:type,` $w)$ if $nodes(v_1, S) \cap nodes(w, S) \neq \emptyset$. Consider 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 generate e if $triples(a, S) \cap triples(t, S) \neq \emptyset$. If some triple comprising a is of the form $(u, \text{rds} : \text{type}, v)$, then action a may generate an event of the form `INSERT` x `AS INSTANCE OF` y if $nodes(u, S) \cap nodes(x, S) \neq \emptyset$ and $nodes(v, S) \cap nodes(y, S) \neq \emptyset$.

Analogous tests apply to actions and events that correspond to deletions of resources, deletions of triples, and updates of triples.

Example 1. Consider RDF schema S formed by merging S_1 from Figure 1 and S_2 from Figure 2 based on their common node labelled `LO`. Suppose that we have rule r_1 with its action part containing the following action a

```
INSERT resource(http://www.dcs.bbk.ac.uk/LOs/BK187)
       AS INSTANCE OF LO
```
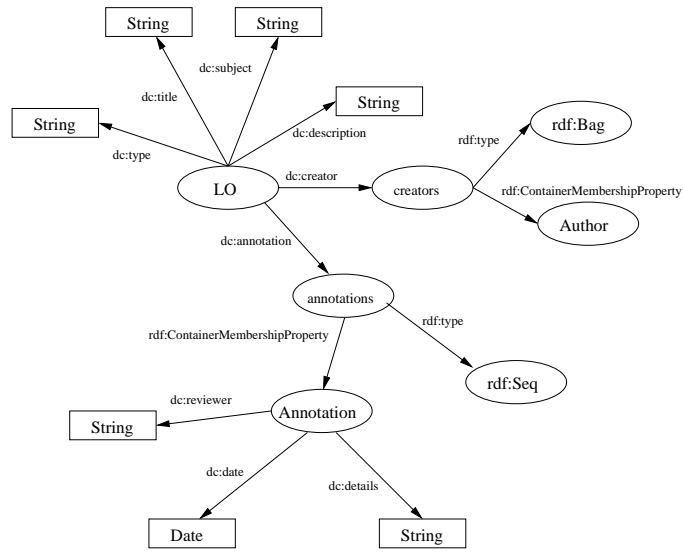
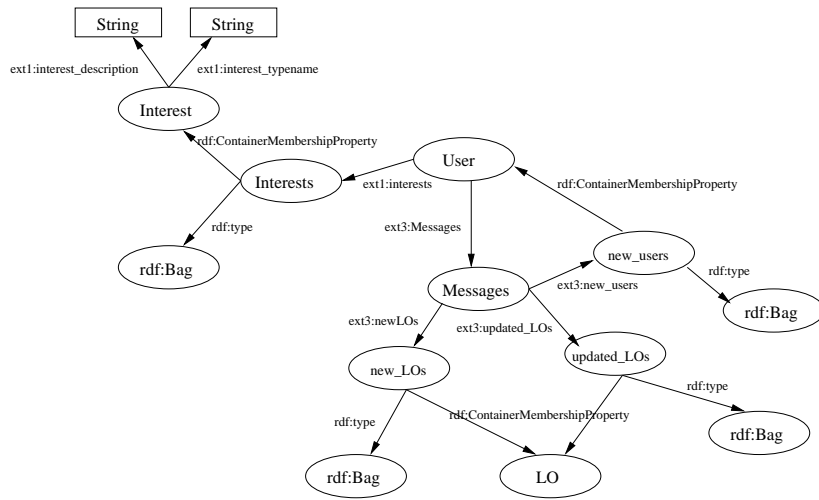**Fig. 1.** RDF schema describing learning objects.

Fig. 2. RDF schema describing users.

and a rule r_2 with event part e

```
INSERT resource()/target(ext3:Messages)/target(ext3:updated_LOs)
                 /element() AS INSTANCE OF LO
```

We have that $nodes(a, S) \cap nodes(e, S) \neq \emptyset$, since both sets of nodes include LO. Hence, r_1 may trigger r_2.

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. In order to shorten the explanation, we assume that all updates are to triples, rather than resources, but the treatment extends easily to resources also. Let R be a set of RDFTL rules. For an action a, we can determine the events of rules in R that may be generated by a, and 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 set of triples 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. For rule r, $triplesQueried(r, S)$ is the union of the sets $triplesQueried(p, S)$ for each path expression p in r. For action a, $triggeredTriplesQueried(a, S)$ is then the union of the sets $triplesQueried(r, S)$ 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.

Now, an action a is independent of an action b if

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

and

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

Example 2. Consider RDF schema S formed by merging S_1 from Figure 1 and S_2 from Figure 2 based on their common node labelled LO. Now consider 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 ext3 http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/lo
LET $updated_los := resource(http://www.dcs.bbk.ac.uk/users/sys)
                 /target(ext3:messages)/target(ext3:updated_LOs)
ON INSERT (resource(),dc:type,_)
IF True
DO INSERT ($updated_los,seq++,$delta);;
```

Here, the syntax `seq++` denotes an increment to a collection's current element count, in order to insert a new element.

Then $triggeredTriplesQueried(a, S)$ comprises the 3 triples (`LO`, `dc:type`, `String`), (`User`, `ext3:messages`, `Messages`) and (`Messages`, `ext3:updated_LOs`, `updated_LOs`), while $triggeredTriplesUpdated(a, S)$ comprises the single triple

(`updated_LOs`, `rdfs:ContainerMembershipProperty`, `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

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

$$\cap$$

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

2.5 Rule Commutativity

Two rules r_i and r_j with the same coupling mode that may be triggered at the same time are said to **commute** if the same final RDF graph will result when rule execution terminates, up to ordering of elements in bag and set collections, irrespective of the order in which the 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, up to the ordering of elements in bag and set collections;

(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 s_i be the set of RDFS schema triples whose extents may be accessed by the action part of r_i and s_j the set of RDFS schema triples whose extents may be accessed by the action part of r_j. Then, a conservative test for C1 is that s_i and s_j do not intersect.

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

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

$$\cap$$

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

3 RDFTL Rules in P2P Environments

We are in the process of implementating a system for processing RDFTL rules in P2P environments. The main component of our system is the *RDFTL ECA Engine* (see below). This provides an 'active' wrapper over a 'passive' RDF/S repository, exploiting the query, storage and update functionality of such a repository. In our current implementation, the RDF/S repository is RDFSuite [ACK$^+$01] from ICS-FORTH.

The architecture that we are implementing is illustrated in Figure 3. Each 'superpeer' server shown in that picture may be coordinating a group of further 'peer' servers, which we term its 'peergroup', as well as itself hosting a fragment of the global RDF/S descriptions in the network. 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.

At each superpeer there is installed one ECA Engine operating as a Web Service. 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 coordinating superpeer of any updates to their local RDF/S repository. Peers may dynamically join or leave the network.

In order to control access to its peergroup's data, 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 access privileges are also allowed on specific RDF resources and properties. These facilities allow a superpeer to specify which data can be shared with other superpeers outside its peergroup.

In the dynamic applications that we envisage, ECA rules are likely not to be hand-crafted 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 $\texttt{on } e \texttt{ if } c \texttt{ do } a_i$ and $r_1 \; prec \; r_2 \; prec \; \ldots \; prec \; r_n$. Note that it is also possible to relax
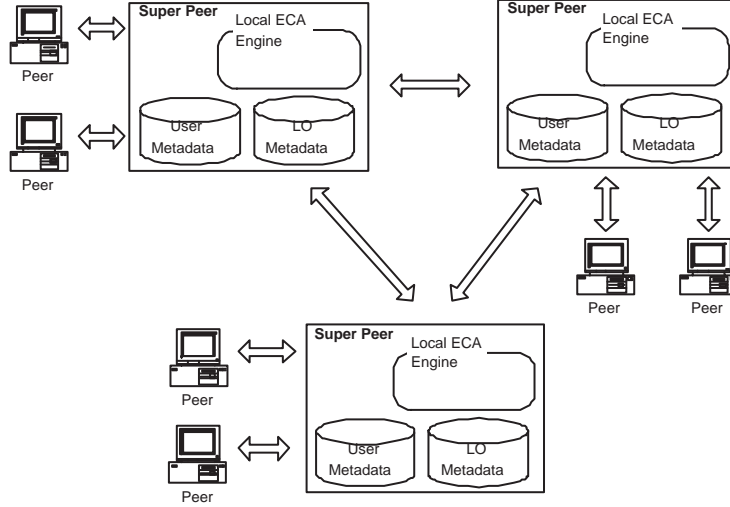
Fig. 3. P2P System Architecture

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 architiure 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 in the event part of r 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 in 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 superpeer is e-relevant, c-relevant or a-relevant to a rule r if r is e-, c- or a-relevant, respectively, to the superpeer's RDFS schema.

Example 3. Consider the following RDFTL rule r:

```
USING NAMESPACE dc http://purl.org/dc/elements/1.1/
USING NAMESPACE ext1 http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/user
USING NAMESPACE ext3 http://www.dcs.bbk.ac.uk/~gpapa05/semapeer/lo
LET $new_los := resource(http://www.dcs.bbk.ac.uk/users/128)
                /target(ext3:messages)/target(ext3:new_LOs)
ON INSERT resource() AS INSTANCE OF LO
IF $delta/target(dc:subject)
   = resource(http://www.dcs.bbk.ac.uk/users/128)
     /target(ext1:interests)/element()/target(ext1:interest_typename)
DO INSERT ($new_los,seq++,$delta);;
```

and the RDF schemas S_1 and S_2 given in Figures 1 and 2, respectively. Then r is *e-relevant* to both S_1 and S_2, since its event part can be evaluated on both schemas. It is *c-relevant* to both as well, since, for example, `dc:subject` occurs in S_1 and `ext1:interests` occurs in S_2. Finally, r is *a-relevant* to S_2 since the newly added property between the collection represented by `$new_los` and the newly inserted `LO`, represented by `$delta`, belongs to `rdf:ContainerMembershipProperty` class that is contained in S_2.

Each ECA engine at each superpeer, SP, consists of several subcomponents:
- The *RDFTL Language Interpreter* takes as input an RDFTL rule definition, generated by some peer of SP's peergroup, and registers it in SP's Rule Base. The Language Interpreter consists of a *Parser* which verifies the syntactic correctness of the rule, and a *Translator* which translates any path queries within the rule into the query syntax of the underlying RDF repository. In our present implementation, RDFTL path expressions are translated into RQL [KAC+02] which is the query language supported by RDFSuite.
- The *Event Detector* detects 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 Detector determines which rules have 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. It does this by submitting appropriate queries to the SP's Query Service to determine which of the triggered rules' conditions

are true. This may require distributed query processing across a number of peers and superpeers, which we assume is coordinated by the Query service.

– 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 sent to all other superpeers over the network. At any superpeer, whether these updates are or are not added to the local execution schedule depends on whether:

(a) they are *a-relevant* with respect to the local schema and

(b) the local schema allows update rights to all the resources affected by the updates.

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

3.1 Registering a new ECA rule

Whenever a new ECA rule r is registered at a peer P, it is sent to P's superpeer for syntax validation, translation of path expressions to the local repository's query 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.

Peer and SuperPeer indexes. Determining the e-, c- and a-relevance of a particular ECA rule to a superpeer involves comparing path expression(s) in that part of the rule against the superpeer's RDFS schema. In order to aid this comparison, a simple *index* can be kept at each peer and superpeer. There are a number of possibilities for doing this [CGM02] [NWS+03] [KP03] [GWJD03], and this is our solution:

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

This information is also propagated to the peer's coordinating superpeer. This superpeer maintains a combined RDFS schema which is annotated so that each node shows the set of peers in its own peergroup that manage data of this type (a set of peer IDs).

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

Registering a new ECA rule. When a new rule is generated at a peer, it is sent to the peer's coordinating superpeer for storage in its local rule base. The superpeer annotates the event, condition and action parts of the rule with the local peers that are relevant to each part (a list of peer IDs). This can be determined by matching each part of the rule against the superpeer's annotated RDFS schema S and/or its resource index I — the former is useful if no resource is specified in this part of the rule and the latter is useful if a resource is specified.

In particular, let i be a peer ID and let S_i denote the subgraph of S induced by nodes whose annotation includes i. The event part of r is

annotated with peer ID i if r is e-relevant to S_i and any specific resource mentioned in the event part of r is in resource index I. 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 and every resource mentioned in the action part of r is in I.

Using the *Routing Service*, the new rule is also propagated to all other superpeers of the network and it is stored at those superpeers that are e-relevant to it. At such a superpeer, the process of matching each part of the rule against the superpeer's annotated RDFS schema is repeated, and the resulting annotated rule is stored in the superpeer's rule base. The final result is a replica of the rule at each superpeer which is e-relevant to the rule, annotated with local information about which peers 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 or assigning higher priority to more specific rules).

Over time, a peer 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). These changes are propagated to P's coordinating superpeer.

Changes in a superpeer's RDFS schema require the annotations of each part of each rule in its rule base to be updated. Any rules that are no longer relevant to the superpeer can be deactivated. Conversely, if the RDFS schema at a superpeer SP has changed from having no data associated with a particular RDFS schema node to now having such data, 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.

3.2 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, it will notify its coordinating superpeer SP. This 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 P r's event query to evaluate.

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

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 data in order to decide whether each update 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 coupling mode and 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.

In theory, any distributed concurrency control protocol could be adapted to a P2P environment. For example, the AMOR system adopts optimistic concurrency control [HSS03]. 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 can 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 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.

We assume as the default that a parent transaction (or subtransaction) and its immediate subtransactions can commit independently of each other, and so an **abort dependency** now needs to be specified for each rule with an Immediate or Deferred coupling mode — not just for rules with Detached coupling mode. The possible abort dependencies are as described in Section 2.3, with T_o being the parent (sub)transaction and T_r the child subtransaction in this case.

This raises the question of what happens to transactions that have read from committed subtransactions which are subsequently reversed? One solution is to only allow this for transactions which commute with the committed transaction, so that they do not need to be reversed if the committed subtransaction has been reversed (as in the open nested transaction model [WS92]). Another solution is to allow it for all transactions, but then carry out cascaded compensations if necessary.

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 generated as a result of the firing of Immediate or Deferred rules, and between transactions and other transactions generated as a result of the firing of Detached rules.

The abort graph is distributed amongst the superpeers that participate in any subtransaction of a top-level transaction. This graph is built dynamically step-by-step 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 at 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 we hold all the necessary information in order to initiate a compensating transaction, at any transaction level.

Figure 3.2 below 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).

Level 1

T0 –> {SP1}

Level 2

T1–>{SP2}   T2–>{SP1}   T3–>{SP3}   T4–>{SP4}

Level 3

T5–>{SP2}   T6–>{SP5}   T7–>{SP3}   T8–>{SP4}   T9–>{SP2}

Abort Dependencies

where:

Tparent ⟶ Tchild : ParentChild

Tparent ⟵ Tchild : ChildParent

Tparent ⟷ Tchild : Mutual

Tparent   Tchild : Independent

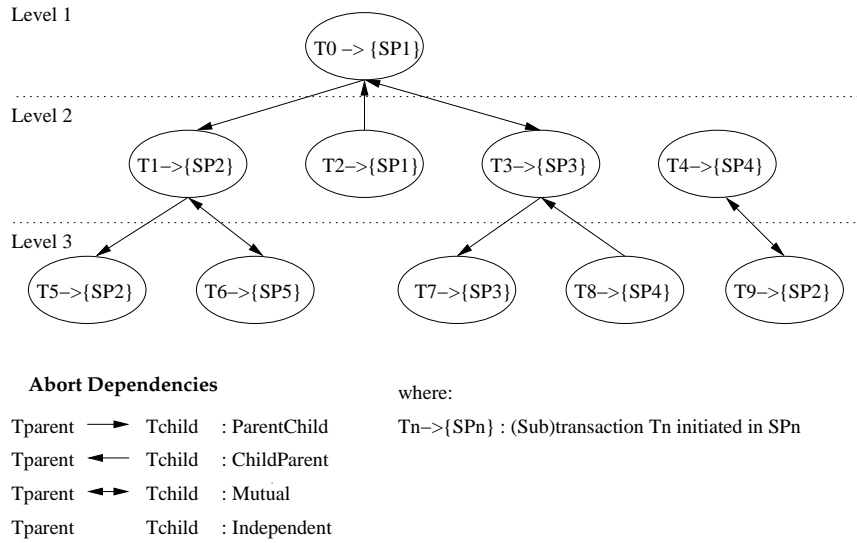Tn–>{SPn} : (Sub)transaction Tn initiated in SPn

Fig. 4. Abort graph example

4 Concluding remarks

We have described a language for defining ECA rules on RDF metadata, including its syntax and execution semantics. We have developed conservative tests for determining when the order of execution of pairs of rules, or instances of the same rule, is immaterial to the final effects of rule execution on the RDF metadata. We have described an architecture for supporting such rules in P2P environments, and have discussed techniques for relaxing the isolation and atomicity requirements of transactions. We are currently implementing this architecture, deploying the system and evaluating its performance. We are also developing more sophisticated tests for determining rule commutativity, which we will report on in a forthcoming paper. Future work will consist of implementing

the transaction management techniques discussed here, and evaluating their effects on system behaviour and performance.

References

[ACK⁺01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plex-ousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *2nd. Int. Workshop on the Semantic Web (SemWeb 2001)*, 2001.

[CGM02] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS 2002*, 2002.

[GMS87] H. Garcia-Molina and H. Salem. Sagas. In *SIGMOD '87*, pages 249–259, 1987.

[GWJD03] L. Galanis, Y. Wang, S. R. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB'03*, 2003.

[HSS03] K. Haller, H. Schuldt, and H.J. Schek. Transctional peer-to-peer information processing: The amor approach. In *4th Int. Conf. on Mobile Data Management*, pages 356–362. Springer-Verlag, 2003.

[KAC⁺02] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plex-ousakis, and M. Scholl. RQL: a declarative Query Language for RDF. In *WWW 2002*, 2002.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

[KP03] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT'03*, 2003.

[NWS⁺03] 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 *WWW 2003*, pages 536–543, 2003.

[Pat99] N. Paton. *Active Rules in Database Systems*. Springer, 1999.

[PPW04] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *Proc. 3rd Int. Workshop on Web Dynamics (in conjunction with WWW2004)*, 2004.

[W3C99] W3C. XML Path Language (XPath), 1999.

[W3C04] W3C. RDF Semantics, W3C Recommendation 10 February 2004, 2004.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of mul-tilevel transactions and open nested transactions. In *Database transaction models for advanced applications*, pages 515–553. Morgan Kaufmann Publishers Inc., 1992.