The Resource Action Language: Towards Designing Reactive RDF Stores

Jean-Yves Vion-Dury and Nikolaos Lagos

Xerox Research Centre Europe
{jean-yves.vion-dury, nikolaos.Lagos}@xrce.xerox.com

Keywords: Streaming, RDF query, context, ReAL.

Abstract. In an interconnected world such as the one envisioned by pervasive computing, systems should be able to react to stimuli received from the environment in a streaming fashion. Reactions may include not only performing local updates, but also sending and asking for information from other systems, waiting for responses, and requesting for changes. In this paper we give a short introduction to the main principles of a language we are developing to achieve that, ReAL. Key elements of ReAL in that context include the introduction of explicit operators to deal with concurrency, nested transactions, and streams.

1 Introduction

In an interconnected world such as the one envisioned by pervasive computing, systems should be able to react to stimuli received, for instance thanks to sensors, from the environment. Reactions may include not only sending information but also asking for information from other systems, waiting for responses, and requesting for changes in a continuous, streaming fashion. Streaming extensions to the standard Semantic Web query language (SPARQL) have been developed for dealing with continuous data flows [1-5], with the most interesting in our context being EP-SPARQL [3] that uses events as triggers of query execution. However, we observed that interaction with other systems, and the effects on the design of a corresponding query language, have not been explicitly considered up to now.

In this paper we give a short introduction to the main principles of a language we are developing to bridge this gap, **ReAL** (**Re**source **A**ction **L**anguage). The overall objective is to provide a means for describing the dynamic behavior of RDF stores in a streaming fashion, i.e. handle queries within a specific execution context, perform (potentially transformative) actions on the store itself, and allow interaction with external services. The design principles include.

- Offer an explicit mechanism of (nested) transactions, thus allowing the execution context to be clearly defined at query time.

- Use a concurrency model to allow coordination with other services – we do that based on a "Triple Space" derived from "Tuple Space" as formerly done in coordination languages like Linda [6].
- Follow a streaming execution model to enumerate solutions one by one, thus propagating solutions as soon as possible.
- Allow a synthesis of query and production-rule languages (to define actions and their impacts within the query).
- Aim for modular and highly compositional programming structures (procedures).

In this paper we don't target exhaustivity. In particular, many general purpose primitive actions are missing, as we essentially focus on some of the most interesting features of ReAL in our context. Other higher level actions (e.g. time-oriented and memory-protection-oriented primitives) are work in progress.

We have to note here that ReAL can be seamlessly linked to the LRM upper level ontology [7], being developed in the PERICLES project[1]. An example of such integration will be described in the paper. The LRM OWL ontology has been designed to address dynamicity in the digital preservation field, with a focus on change management through sophisticated model to handle intentional dependencies, versioning mechanisms and reflexive metadata modeling. If ReAL is designed as a "natural infrastructure" to support LRM based services, we do believe that its more fundamental qualities are not bound to any particular data model.

2 Matching, Bindings and Basic Actions

Triples are represented through a syntax similar to the one adopted by the abstract syntax of SWRL [8], using functional notation like *predicate(subject, object)*. where any of the three components can be an IRI using a prefixed form or a variable *?name*. The object component can additionally be a string like "3.1416", a decimal/integral number, or a symbol like true, false. Note that triples extended with language tags or typing IRI are captured by an additional argument (separated by "|"), e.g.:

```
rdf:label(test:c1,"my class" | en)
ex:weight(test:c1,"0.12456" | xsd:decimal)
```

Based on this notation, we introduce next the most basic primitive constructs to perform reading and writing in the RDF store. They constitute what we like to call *basic actions*.

Simple reading. The following reading expression (illustrative)

```
rdf:type(?sub, ?class)              (1)
```

will succeed if at least one solution can be read in the triple store. Solution here designates all triples matching the expression. The result is of the form <boolean, Bind-

[1] http://pericles-project.eu/

ing>, where boolean (true or false) denotes whether a solution is found, and Binding denotes the set[2] of pairs (variable, term[3]). Failing queries always return <false,{}>. A new Binding is streamed whenever a matching solution is found, and can be defined as a mapping relating all variables (e.g. ?sub) to subterms such that the filtering terms are made equal to the matching terms. In other words, a Binding represents the substitutive solution that equates the filter to the instance. The substitution operation of an expression e using a binding B is a new expression noted B(e). Note that the expression (1) above, if changed into e.g.

```
rdf:type(ex:nantes-triptych, ex:Artwork)          (1b)
```

could stream a unique solution (an empty binding {}) in a context where the triple is indeed present in the RDF store.

Destructive reading. To express that you want not only to filter-out the RDF store, but also to withdraw the matching solutions, you may use a "-" operator as a prefix.

```
- rdf:type(?sub, ex:Artwork)              (2)
```

Note that the store is immediately modified, so that unforeseen "side effects" may occur when such an instruction is combined with others, even if those do not return any solution eventually (this is one reason why nested transactions are relevant in ReAL, as we will see later). Destructive reading may fail if the triple is write-protected (such protection mechanisms will not be presented here).

Explicit inference invocation. When one needs to extract more complex information from the store, he may use inference to stream solutions, thanks to the "!" prefix.

```
! rdf:type(?sub, ex:Artwork)              (3)
```

The type of inference is dependent on the context and on the configuration of the corresponding infrastructure, but typically, it could exploit a background taxonomy or ontology. For instance, provided that relations like rdfs:subClassOf (ex:VideoArt, ex:Artwork) and rdfs:subClassOf (ex:SoftwareBasedArt, ex:Artwork) are included in the underlying knowledge base, corresponding inference (based on the rdfs:subClassOf) could be used to infer that the instances of ex:VideoArt and ex:SoftwareBasedArt are solutions of query (3).

Writing triples. In order to insert a new triple inside the store, one may use the "+" prefix:

```
+ rdf:type(ex:nantes-Triptych, ex:Artwork)          (4)
```

[2] The set can be empty if the pattern does not involve any variable, or only jokers, noted *?*

[3] Here the term's syntax is defined by the non-terminal BItem of the formal grammar provided in the appendix.

When one wants to write a triple into the store, the operation might fail if: the triple already exists; the triple is not well formed (remaining unbound variables, bad element organization, such as a string placed at the subject position) - this is an error case; the triple is write-protected (individual triples, or families of triples can be write-protected by a lock - not developed here); the store forbids the writing of such triples (similarly, one can specify access rights; not developed here). If the operation is successful, it will stream a unique solution: the empty binding.

3 Blocking Actions

Reading and writing actions can be suspended until completion when specified with the WAIT primitive. It means that the ReAL process will be suspended until the action can be fulfilled in the current context. This is a powerful way to synchronize and communicate information between concurrent ReAL processes through the intermediary of the RDF triple store (à la LINDA [6]). As an illustration, the three expressions below

```
WAIT rdf:type(ex:nantes-triptych, ex:Artwork)
WAIT -rdf:type(ex:nantes-triptych, ex:Artwork)
WAIT +rdf:type(ex:nantes-triptych, ex:Artwork)
```

will wait for the triple if not initially present at evaluation time. For the last one, a writing action, the waiting process will not start if the problem is linked to a badly formed triple issue (the action will just fail).

4 Stream-based Logical Connectors

Considering that an expression becomes "true" if at least one solution exists, we propose to consider our set of combinators (as described below) as being dual, each of them being both a logical combinator and a stream-based composition operator as well.

AND. The binary combinator "AND" allows combining solutions from both operands. An (e1 AND e2) expression first looks for solutions of e1; for each corresponding binding B1, it is applied to e2 (applying a binding means doing a substitution: if e2 shares variables with e1, they will be instantiated) and then the operator looks for solutions for B1(e2) and streams them as results. As an illustration,

```
rdf:type(?sub,ex:ArtWork) AND ex:creator(?sub,ex:BillViola)   (5)
```

will stream the subject IRI for all art works by Bill Viola explicitly known in the RDF store. Now, if we want to do a more powerful operation, for instance replacing the "ex:BillViola" IRI with another one (where the IRI is more abstract and does not mention a name), and specifying the artist's name through the rdfs:label property:

```
$iri AND
+rdfs:label(?iri,"Bill Viola") AND
rdf:type(?sub, ex:ArtWork) AND                Example (6)
-ex:creator(?sub, ex:BillViola) AND
+ex:creator(?sub, ?iri)
```

The notation $iri is a syntactic sugar for FRESH(?iri), a primitive that streams fresh and unique IRIs bound to the variable ?iri.

OR. This binary combinator propagates only the left substream if any. Otherwise, it propagates the right one, if any.

UNION. This binary combinator propagates first the left substream if any. Afterwards, it propagates the right substream if any (meaning that it fails if both substreams fail). Note that like the OR primitive, no junction is done between left and right terms.

NO. This combinator streams the empty binding if no solution is found for the subexpression, fails otherwise. Usage example:

```
NO rdf:type(?x,rdfs:Class)
```

FIRST. this unary operator evaluates its subexpression, and just streams the first solution if any. Albeit the RDF store is not ordered, streams can be ordered, especially when yielded by inference based queries.

LAST. Same behavior than FIRST, except that only the last solution is found. Note that (i) it cannot work with infinite streams, and (ii) the substream is delayed until completion since only the last solution is streamed.

REPEAT. Evaluate the subexpression but do not propagate any solutions. Fails if the subexpression fails, returns the empty binding when the stream terminates. Usage example:

```
REPEAT (rdf:type(?x,ex:Book) AND +rdfs:label(?x,"scanned"))
```

Repeat can be parameterized by a counting parameter i.e. number of solutions. If the number cannot be reached, the action will fail. For instance:

```
REPEAT 1 (rdf:type(?x,rdfs:Class) AND + rdfs:label(?x,"scanned"))
```

performs only one action. It is not equivalent to FIRST because REPEAT is opaque. The expression below, will perform as many times as possible the actions of the subexpression, and will return a binding giving the value of ?count, i.e. the number of solutions.

```
REPEAT ?count (rdf:type(?x,rdfs:Class) AND +rdfs:label(?x,"scanned"))
```

TRUE. Streams the empty binding.

FALSE. Always fails (do not stream anything).

CALL *some-IRI (i0 ... ik >> o0 ... on).* This operator executes the actions defined by some-IRI; parameters i0… ik are passed to the target environment; outputs o0 … on, when defined, are used to rename the bindings streamed by the action if any.

SPAWN *some-IRI (i0 ... ik).* This action is similar to the CALL action, except that the action will be executed in a concurrent micro-thread, and cannot stream any solution (one must use synchronized triples to exchange data). This is therefore an asynchronous call, as opposed to CALL which is synchronous.

STOP *some-IRI.* This action stops a process (designated by some-IRI) but fails if the process is not found or is not active anymore.

STOP *some-IRI (msg-IRI).* Same as the previous combinator, but a message will be associated (msg-IRI, should be an IRI of a lrm:Message instance) to the lrm:ActivityStopped event that will be attached to the RDF activity descriptor (aka some-IRI).

5 Transactions and Sandboxes

Example (6) may raise a problem when the store does not contain any artwork by Bill Viola. In that case, the global action will fail (not returning any solution/binding) when evaluating the third operand rdf:type(?subject, ex:Artwork) but however the store will be eventually modified: a triple rdfs:label(_:b1,"Bill Viola") will be inserted as a side-effect. Indeed, the writing action (as specified by the second operand) is done immediately, as explained in previous sections. One very obvious solution is to reorder the operands:

```
rdf:type( ?sub, ex:ArtWork) AND
- ex:creator(?sub, ex:BillViola) AND
$iri  AND
+ rdfs:label(?iri,"Bill Viola") AND
+ ex:creator(?sub, ?iri)
```
(6b)

Another more generic solution is to use a transaction: all transformative actions are committed at each streamed solution, if any. If no solution is yielded, the transaction is aborted and the store stays unchanged. The transaction is specified by enclosing square brackets […], and transactions can be nested. A transaction is transparent (i.e. it always propagates the substream).

```
$iri  AND [
 + rdfs:label(?iri,"Bill Viola") AND
```
(6c)

```
 rdf:type( ?sub, ex:ArtWork) AND
 - ex:creator(?sub, ex:BillViola) AND
 + ex:creator(?sub, ?iri)
]
```

A similar mechanism, called the sandbox, allows to confine all transformative actions into a temporary substore which will be forgotten after evaluation, be it a success or not (so it behaves like a transaction that is always aborted). It is denoted by enclosing brackets {…} and like for transactions, it is transparent (it always propagates the substream).

6 Handling Graphs

Graphs can be viewed as a way to modularize RDF stores. We propose two combinators to work with graphs: ON and IN. Their behavior is defined according to a dedicated execution structure, namely, a stack of contexts (an RDF graph for instance can be considered as a context). At the bottom of the stack, there is always the default context (i.e. the context stack is never empty), and transformative actions (addition and deletion of triples) are always performed in the context lying on the top of the stack.

```
ON <iri> or <var> {action}
```

If the first parameter is an IRI, it must designate an existing graph. If the parameter is a variable, the graph will be created, and in extension to the standard RDF 1.1 semantics, a triple `rdf:type(iri, rdf:Graph)` will be created inside the top context[4]. This (new) graph will be pushed on the stack, and will become the new active context. The action associated with the ON operation will be undertaken and solutions streamed up. Note that transformative actions (insertions and deletions) will only affect the top context, however reading actions will explore the whole context stack in the top-down direction.

```
IN <iri> or <var> {action}
```

The semantics are pretty much the same, except that IN builds a stack of one unique context, the one given as parameter of the action. Therefore, transformative and reading actions associated with the IN operations are all confined to the same unique graph (in that sense, it is much more restrictive: it locally behaves like if the default store and other graphs do not exist).

7 Summary and Ongoing Work

We have presented the main design principles of a query language for RDF stores based on the notion of actions. We have presented several combinators to handle con-

[4] Actually, all named graphs will be associated with such a triple.

currency, enable interaction with external services, and define the context of execution via the notion of nested transaction.

Currently we are working on:

- Experimenting the most innovative operators, especially the transactional and graph related combinators (we expect the former to simplify greatly concurrent modification and the latter to provide means for simple and efficient safety control mechanisms).
- Decoupling completely ReAL from LRM. The current version of ReAL is still dependent for some operators on the LRM ontology (they are both being developed in the context of the same project, PERICLES). For instance, in the combinator *CALL some-IRI (i0 ... ik >> o0 ... on)* the reference some-IRI must designate today an instance of a specific LRM class (lrm:Action) which, by design, defines a unique predicate lrm:body where the ReAL code describing the actions is inserted. Fig. 1 shows an example of such an instance which is used to invoke a certification service for the versioning of an entity.

```
235  lrm:askCertification  a  lrm:Action;
236       rdfs:label  "lrm:askCertification";
237       lrm:input-signature  "rsc, kind, agent"^^real:Signature;
238       lrm:output-signature  "certif "^^real:Signature;
239       lrm:body  [  real:code  """
240            rdf:type ( ?ctf, lrm:CertificationService)   AND
241             $cr AND
242            +rdf:type ( ?cr, lrm:CertificationRequest)   AND
243            +lrm:ofResource ( ?cr,?rsc)   AND
244            +lrm:requestedService ( ?cr,?ctf)   AND
245            +lrm:byAgent  ( ?cr,  ?agent)   AND
246            +lrm:versionKind  ( ?cr,  ?kind)   AND
247            CALL primitive:service ( ?cr)   AND
248            NO lrm:error ( ?cr, ?)   AND
249            lrm:certificate ( ?cr, ?certif)
250       """^^real:Code  ]
251       .
252
```

Fig. 1. Example of ReAL and LRM integration

Line 237 in Fig. 1 defines the input signature, which must be matched with the input parameters (order and cardinal of the list are both significant and must match; also true for the output) given by the caller; the line 238 defines the output signature: these parameters will be streamed back to the caller if solutions are found.

- Defining the formal semantics for ReAL.
- Analyzing the relation of SPARQL (and streaming variants) to ReAL.

The results of the above three actions will be made available in the near future.

Acknowledgments. This work takes place in the framework of the PERICLES project which received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 601138. We thank our colleagues and partners for the fruitful exchanges we shared. We also thank Mehreen Ikram and Stéphane Jean for their valuable collaboration into bridging formally the semantics of the above language with the one of SPARQL.

8 References

1. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF Streams with C-SPARQL. SIGMOD Record 39(1): 20-26 (2010)
2. Calbimonte, J.-P., Jeung, H., Corcho, Ó., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. Int. J. Semantic Web Inf. Syst. 8(1): 43-63 (2012)
3. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning, WWW 2011. Proceedings of the Twentieth International World Wide Web Conference, India (2011)
4. Dehghanzadeh, S., Dell'Aglio, D., Gao, S., Della Valle, E., Mileo, A., Bernstein, A.: Approximate Continuous Query Answering over Streams and Dynamic Linked Data Sets. In: 15th International Conference on Web Engineering (ICWE 2015), Rotterdam, The Netherlands. pp. 307–325. Springer (2015)
5. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL Semantics: a Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. IJSWIS 10(4), 17–44 (2015)
6. Wells, G.: Coordination languages: Back to the future with linda. In Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05) 87-98 (2005).
7. Vion-Dury, J.-Y., Lagos, N., Kontopoulos, E., Riga, M., Mitzias, P., Meditskos, G., Waddington, S., Laurenson, P. and Kompatsiaris, I.: Designing for Inconsistency - The Dependency-based PERICLES Approach. In T. Morzy, P. Valduriez, L. Bellatreche (Ed.), New Trends in Databases and Inf. Systems, 539, 458-467. Springer Berlin Heidelberg (2015)
8. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member submission 21, 79 (2004)
9. PERICLES European project. http://www.pericles-project.eu/

Appendix: EBNF Grammar

```
ReAL ::=  '[' ReAL ']'    |
     '{' ReAL '}'    |
     'IN' Pattern '{' ReAL '}'    |
     'ON' Pattern '{' ReAL '}'    |
     ReAL 'AND'  ReAL  |
     ReAL 'UNION'  ReAL  |
     ReAL 'OR'  ReAL  |
     'NO'  ReAL  |
```

```
     'FIRST'  ReAL     |
     'LAST'  ReAL     |
     'REPEAT'  ReAL     |
     'REPEAT'  Item ReAL     |
     '(' ReAL ')'     |
     'TRUE'  |
     'FALSE' |
     Action

Action  ::=
     'CALL' CallPattern      |
     'SPAWN' CallPattern      |
     'STOP'  Pattern      |
     'STOP' CallPattern      |
     'EXPAND' CallPattern    |
     '!' TriplePattern      |
     Iri '!' TriplePattern      |
     '$' <symbol>      |
     'WAIT' BasicAction     |
     BasicAction

BasicAction    ::=
     '+' TriplePattern    |
     '++' TriplePattern    |
     '-' TriplePattern    |
     TriplePattern

TriplePattern  ::=
     Pattern '(' Pattern ',' Item ')'  |
     Pattern '(' Pattern ',' Item, '|' Pattern ')'  |
     Pattern '(' Pattern ',' BTree  ')'

CallPattern  ::=
     Iri '(' ItemList?  ')' |
     Iri '(' ItemList? '>>' ItemList? ')'

Item       ::= Pattern |  Atom
Pattern    ::= Iri  |  Var
Iri        ::=  <symbol> ':' <symbol>
Var        ::=  '?'  |   '?' <symbol>
Atom       ::= <string> |  <number> |  <symbol>
ItemList   ::=  Item ',' ItemList  |  Item
BTree      ::= '[ ' BItem   BTreeList ']'
BTreeList  ::= BItem  BTreeList   |  BItem
BTree      ::= '{ '  BList '}'
BList      ::= BItem  "," BList   |
               BItem  BList   |
               '|' BItem  |  BItem
BItem      ::= BTree | Item
```