R-DEVICE[1]: An Object-Oriented Knowledge Base System for RDF Data

User and Installation Manual

Nick Bassiliades

Dept. of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

nbassili@csd.auth.gr

Introduction

R-DEVICE is a deductive object-oriented knowledge base system, which transforms RDF triples into objects and uses a deductive rule language for querying and reasoning about them. More specifically, R-DEVICE imports RDF data into the CLIPS production rule system as COOL objects. The main difference between the RDF data model and our object model is that properties are treated both as first-class objects and as attributes of resource objects. In this way properties of re-sources are not scattered across several triples as in most other RDF storage and querying systems, resulting in increased query performance due to less joins. The descriptive semantics of RDF data may call for dynamic redefinitions of resource classes, which are handled by R-DEVICE.

R-DEVICE features a powerful deductive rule language which is able to express arbitrary queries both on the RDF schema and data, including generalized path expressions, stratified negation, aggregate, grouping, and sorting, functions, mainly due to the second-order syntax of the rule language which is efficiently translated into sets of first-order logic rules using metadata. R-DEVICE rules define views which are materialized and incrementally maintained. Finally, users can use and define functions using the CLIPS host language.

R-DEVICE Architecture

The R-DEVICE system consists of two major components (Figure 1): the RDF loader/translator and the rule translator. The former accepts from the user requests for loading specific RDF documents. The RDF triple loader downloads the RDF document from the Internet and uses the ARP parser to translate it to triples in the N-triple format. Both the RDF/XML and RDF/N3 files are stored locally for future reference. Furthermore, the RDF document is scanned for namespaces that have not already been imported/translated into the system. Some of the untranslated namespaces may already exist on the local disk, while others are fetched from the Internet. All namespaces (both fetched and locally existing) are recursively scanned for namespaces, which are also fetched if not locally stored. Finally, all untranslated namespaces are also parsed using the ARP parser.

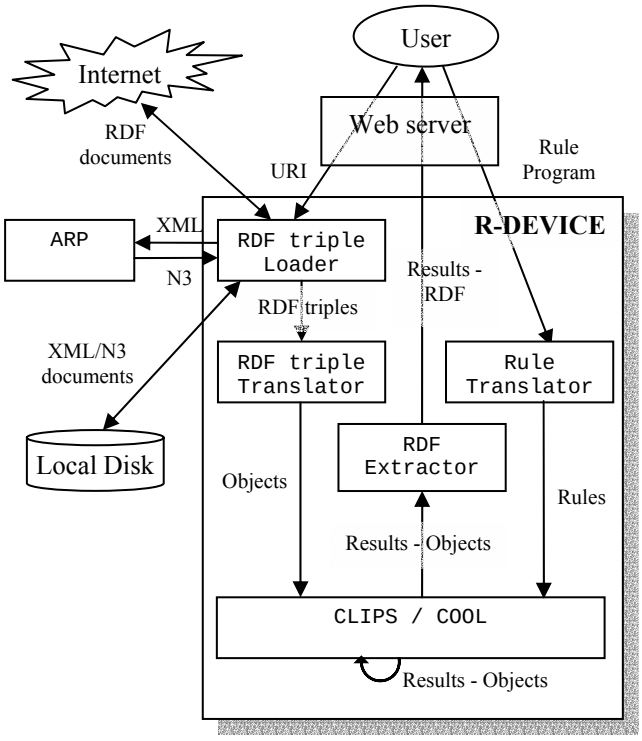

Figure 1. Architecture of the R-DEVICE system.

All N-triples are loaded into memory, while the resources that have a URI#anchorID or URI/anchorID format are transformed into a namespace:anchorID format if URI belongs to the initially collected namespaces, in order to save memory space. The transformed RDF triples are fed to the RDF triple translator which maps them

[1] `http://lpis.csd.auth.grsystems/r-device.html`

into COOL objects, according to the mapping schema that is described in the next section.

The rule translator accepts from the user a set of R-DEVICE rules and translates them into a set of CLIPS production rules. Details about the translation scheme are given in the corresponding section. When the translation ends, CLIPS runs the production rules and generates the objects that constitute the result of the initial rule program or query. Finally, the result-objects are exported to the user as an RDF document through the RDF extractor.

The R-DEVICE Deductive Rule Language

R-DEVICE has a deductive rule language that supports querying over objects and RDF data represented as objects and define materialized views over them that are maintained incrementally. The conclusions of deductive rules represent derived classes, i.e. classes whose objects are generated by evaluating these rules over the current set of objects. Furthermore, R-DEVICE supports recursion, stratified negation, path expressions over the objects, generalized path expressions (i.e. path expressions with an unknown number of intermediate steps), derived and aggregate attributes. Finally, users can call out to arbitrary built-in or user-defined functions of the implementation language, i.e. Prolog and CLIPS, respectively.

In R-DEVICE each deductive rule is implemented as a pair of CLIPS production rules: one for inserting a derived object when the condition of the deductive rule is met and one for deleting the derived object when the condition is no more met, due to base object deletions and/or slot modifications. R-DEVICE uses RETE algorithm to match production rule conditions against the objects.

Figure 1 shows the pair of production rules generated by R-DEVICE for the deductive rule `q6` found in the Appendix. We notice that the assertion of a derived object is based on a counter mechanism which counts how many derivations exist for a certain derived object, based on the values of its slots. The latter also define the identifier of the derived object. Derived objects are created only when not already exist, otherwise their counter is just increased by one. Furthermore, derived objects are deleted when their counter is one, otherwise their counter is decreased by one. Production rules that watch out for possible deletion of derived objects, have the negation of the original deductive rule condition in their condition. Finally, derived objects also keep the object identifiers of their derivators, i.e. the base objects to which they owe their existence, and the name of the rule that derived them. This is needed in order to correctly maintain the derived view.

The syntax of R-DEVICE deductive rules is a variation of the syntax for CLIPS production rules (see section R-DEVICE Rule Syntax). Although R-DEVICE uses COOL objects, the syntax of rules is as if deductive rules query over CLIPS templates, because the syntax is simpler. Specifically, each condition element follows the following format:

```
?OID <- (classname (path-expr value-expr) …)
```

where `?OID` is the (optional) object identifier (or instance name, *not* address) of an object of class `classname`, and `(path-expression value-expression)` are zero, one, or more conditions to be tested on each object that matches this pattern.

When the name of the class is unknown, a variable can be used in place of a concrete class name. Classnames can consist of a namespace prefix followed by a colon and a local part name. R-DEVICE allows the use of variables in both the place of the namespace prefix and the local part name. For example, the following condition element applies to instances of classes of the `rss` namespace.

```
(rss:?c (rss:title ?t))
```

A value expression can be a constant or a variable or a constraint or a combination of those, as defined by CLIPS rule syntax. Examples of value expressions are given below.

A path expression is an extension of CLIPS's single ground slot expression. Specifically, in R-DEVICE a path expression can be one of the following:

- A single slot of the class `classname`. For example, the following single condition element of query `q5` in the Appendix queries slots `rss:title` and `rss:link` of objects of class `rss:item`:

```
(rss:item (rss:title ?t) (rss:link ?l))
```

- A single variable denoting any slot of class classname. For example the following condition element searches for a resource object with an unknown slot whose value is "Smith".

```
(rdfs:Resource (?s "Smith"))
```

- A ground path that consists of a list of multiple slots surrounded by brackets. Query q7 in the Appendix shows an example of such a path in the first condition element:

```
((dcq:RFC1766 dc:language) ?language)
```

The right-most slot should be a slot of the "departing" class. Moving to the left, slots belong to classes that represent the range of the predecessor slots. The value expression in such a pattern (e.g. variable `?language`) actually describes a value of the left-most slot of the path.

```
(defrule gen80
  (declare (salience (calc-salience result)))
  (run-deductive-rules)
  (object (name ?gen79) (is-a rss:item) (rss:title ?title) (rss:link ?link))
  (test (str-index "RDQL" ?title))
  (not (object (name ?DO&:(eq ?DO (symbol-to-instance-name (sym-cat result ?link)))) (is-a result)
                                                    (link ?link) (derivators $? +++ ? ?gen79 +++ $?)))
  =>
  (bind ?oid (symbol-to-instance-name (sym-cat result ?link)))
  (if (instance-existp ?oid)
     then
     (slot-insert$ ?oid derivators 1 +++ gen80 ?gen79 +++)
     (send ?oid put-counter (+ (send ?oid get-counter) 1))
     else
     (make-instance ?oid of result (link ?link) (derivators +++ gen80 ?gen79 +++))))

(defrule gen81
  (declare (salience 1000))
  (run-deductive-rules)
  (object (name ?derived-object) (is-a result) (link ?link) (counter ?old-counter)
                                                    (derivators $?DER-B +++ gen80 ?gen79 +++ $?DER-A))
  (or (test (not (all-instance-existp (create$ ?gen79))))
      (and (object (name ?gen79) (is-a rss:item))
           (not (and (object (name ?gen79) (is-a rss:item) (rss:title ?title) (rss:link ?link))
                (test (str-index "RDQL" ?title))))))
  =>
  (bind ?new-counter (- ?old-counter 1))
  (if (= ?new-counter 0)
     then
     (send ?derived-object delete)
     else
     (modify-instance ?derived-object
        (counter ?new-counter)
        (derivators $?DER-B $?DER-A))))
```

Figure 1. CLIPS production rules generated for the deductive rule `q6` (see Appendix).

- A path that contains one or more single-field variables, i.e. a path whose length is known but some of the steps are not. The above ground path can be turned into such a path:

```
((dcq:RFC1766 ?x) ?language)
```

- A path that contains one or more multi-field variables, i.e. variables that their value is a list. These non-ground paths have an unknown length. The path below can have at least two steps and at most four (given the specific example):

```
((dcq:RFC1766 dc:language $?p) ?x)
```

- A path that contains an encapsulated recursive sub-path, i.e. a sub-path that is traversed an unknown number of times. The following path contains the recursive sub-path (dcq:references) which recursively follows resources that reference each other:

```
((dc:title (dcq:references)) ?t)
```

Recursive paths can be used to express transitive closure queries. For example, the following query collects all resources (pages) recursively referenced by a certain resource.

```
(deductiverule collect_refs
      (? (uri "http://www.csd.auth.gr/~lpis") ((uri (dcq:references)) ?uri))
  =>
      (result (uri ?uri)))
```

Notice that URIs that are reachable following many paths will only be included once in the result and that infinite loops will be avoided, due to the counter mechanism (see above).

Recursive sub-paths can be implicitly included in a path of unknown length. For example in the following path, the multifield variable `$?p` can represent both linear and recursive sub-paths:

```
((dc:title $?p) ?t)
```

Multifield variables can also occur at the place of value expressions, since all RDF properties are treated as multislots. For example, the following pattern retrieves in a list `$?l` all the values for the `rss:link` property of a resource object:

```
(rss:link $?l)
```

On the other hand, if we know that a resource object has many values for one property and we want to iterate over them, the pattern should be:

```
(rss:link $? ?l $?)
```

which means that variable `?l` will eventually become instantiated with all the values of the property `rss:link`. This retrieval pattern is so common that a shortcut is provided which expands to the above pattern during a macro expansion phase.

```
(rss:link ??l)
```

When the value of a specific variable is of no interest then an anonymous variable '`?`' can be used, which is replaced by a singleton system-generated variable during the macro expansion phase.

Selection conditions can be placed inside value expressions or as individual `test` condition elements, as in CLIPS. For example, the following pattern retrieves the family name in a variable and, at the same time, tests if the slot value does not equal "Smith":

```
(vcard:Family ?last&~"Smith")
```

The same condition can also be expressed as:

```
(vcard:Family ?last)
(test (neq ?last "Smith"))
```

Conditions can also express disjunction and negation. Only stratified negation is allowed.

Rule conclusion can also contain a set of function calls that calculate the values to be stored at the slots of the derived object. Such calls are placed inside a `calc` construct before the derived class template. For example, the following variation of rule `q2` retrieves the given and family name of a resource object and using a CLIPS function concatenates them into a single string that is stored in the slot `full-name` of the derived objects of class `person`.

```
(deductiverule q2-variation
      (? (vcard:Family ?f) (vcard:Given ?v))
  =>
      (calc (bind ?full (str-cat ?v " " ?f)))
      (person (full-name ?full)))
```

Finally, R-DEVICE supports aggregate functions and grouping in the form of aggregate attribute rules. These rules express how values are accumulated and combined into attributes of existing objects. For example, the following rule iterates over all resources and generates one object for each distinct creator, which holds in the `URIs` slot all the resources that he/she has created.

```
(deductiverule ex1-aggregate
      (? (dcq:creator ?c) (uri ?uri))
  =>
      (pages (author ?c) (URIs (list ?uri))))
```

Function `list` is an aggregate function that just collects values in a list. There are several other aggregate functions, such as `sum`, `count`, `avg`, etc. Notice in the above example that grouping is performed when the conclusion contains any slot (`author`) other than the aggregate one (`URIs`).

Installation Instructions

- Unzip `r-device.zip` in `C:\Program Files\R-DEVICE` directory.
  - If you want R-DEVICE installed in another directory, then after unzip, edit `r-device.bat` and change the directory at line 1.
- Install LibWWW binaries at `c:\Program Files\Libwww` (or change directory at line 1 of file `arp.bat`).
  - Binaries can be downloaded from `http://www.idm.ru/libwww.htm`.
- Install ARP RDF parser at `c:\Program Files\arp` and Xerces XML parser at `c:\Program Files\xerces` (or change directories at line 2 of file `arp.bat` and line 1 of file `arp-only.bat`).
  - The above parsers can be downloaded from:
    - ARP: `http://www.hpl.hp.com/semweb/arp.html`
    - Xerces: `http://xml.apache.org/xerces-j/index.html`
- Install CLIPS v.6.20.
  - Binaries can be downloaded from http://www.ghg.net/clips/CLIPS.html.
- Run CLIPS and `File` → `Load Batch…` → `r-device.bat`

Running the ODP example

After loading `r-device.bat` …

- Load the batch file `run-test.bat` in the `test/` directory.
  - To run different query examples change the filename `questionN.clp` at line 5.

R-DEVICE commands

- `(set verbose on|off)`

  Set R-DEVICE to display information when importing RDF triples and/or translating deductive rules.
- `(load-rdf` *<filename>* `local)`

  Load in R-DEVICE an RDF file that is stored locally in *<filename>*`.rdf`.
- `(load-rdf` *<filename>* *<address>*`)`

  Load in R-DEVICE an RDF file that can be found at URL *<address>* and also store it locally in *<filename>*`.rdf`.
- `(import)`

  Translate all the loaded RDF triples into COOL objects.
- `(r-device` *<filename>*`)`

  Load and translate the R-DEVICE rules found in *<filename>*.
- `(go)`

  Run the loaded R-DEVICE rules.
- `(export_rdf` *<filename>* *<classes>*`)`

  Export CLIPS *<classes>* in RDF/XML format to file *<filename>*.

R-DEVICE Rule Syntax

```
<r-device-rule> ::=
      <deductive-rule> | <derived-attribute-rule> | <aggregate-attribute-rule>

<deductive-rule> ::=
      (deductiverule [<rule-name>]
            <conditional-element>*
      =>
            <conclusion>)

<derived-attribute-rule> ::=
      (derivedattrule [<rule-name>]
            <conditional-element>*
      =>
            <derived-attribute-conclusion>)

<aggregate-attribute-rule> ::=
      (aggregateattrule [<rule-name>]
            <conditional-element>*
      =>
            <aggregate-attribute-conclusion>)

<conditional-element> ::=
      <pattern-CE> | <assigned-pattern-CE> |
      <not-CE> | <and-CE> | <or-CE> | <test-CE>

<pattern-CE> ::=
      <class-pattern-CE>

<assigned-pattern-CE> ::=
      <single-field-variable> <- <pattern-CE> | <instance-name> <- <pattern-CE>

<not-CE> ::=
      (not <conditional-element>)

<and-CE> ::=
      (and <conditional-element>+)

<or-CE> ::=
      (or <conditional-element>+)

<test-CE> ::=
      (test <function-call>)

<class-pattern-CE> ::=
      (<class-expr> <LHS-slot>*)

<class-expr> ::=
      <class-name> | <svar-expr> | <namespace>':'<class-name> |
      <svar-expr>':'<class-name> | <namespace>':'<svar-expr>

<LHS-slot> ::=
      <single-field-LHS-slot> | <multifield-LHS-slot>

<single-field-LHS-slot> ::=
      (<path-expr> <constraint>)

<multifield-LHS-slot> ::=
      (<path-expr> <constraint>*)

<path-expr> ::=
      <slot-expr> | (<path-item>+)

<slot-expr> ::=
      <slot-name> | <svar-expr>

<svar-expr> ::=
```

```
      <single-field-variable> | '?'

<path-item> ::=
      <slot-expr> | <multifield-variable> | (<slot-name>+)

<constraint> ::=
      '?' | '$?' | <connected-constraint>

<connected-constraint> ::=
      <single-constraint> | <single-constraint> '&' <connected-constraint> |
      <single-constraint> '|' <connected-constraint>

<single-constraint> ::=
      <term> | ~<term>

<term> ::=
      <constant> | <single-field-variable> | <multifield-variable> |
      <single-field-variable-multifield-expression> |
      ':'<function-call> | '='<function-call>

<single-field-variable> ::=
      '?'<variable-symbol>

<multifield-variable> ::=
      '$?'<variable-symbol>

<single-field-variable-multifield-expression> ::=
      '??'<variable-symbol>

<constant> ::=
      <symbol> | <string> | <integer> | <float> | <instance-name>

<function-call> ::=
      (<function-name> <expression>*)

<conclusion> ::=
      [(calc <function-call>+)]
      (<RHS-class-expr> <RHS-slot>*)

<RHS-class-expr> ::=
      <class-name> | <single-field-variable> | <namespace>':'<class-name> |
      <single-field-variable>':'<class-name> |
      <namespace>':'<single-field-variable>

<RHS-slot> ::=
      <simple-assign-expr> | <aggregate-assign-expr>

<simple-assign-expr> ::=
      (<RHS-slot-expr> <value>)

<RHS-slot-expr> ::=
      <slot-name> | <single-field-variable>

<value> ::=
      <single-field-variable> | <multifield-variable> | <constant>

<aggregate-assign-expr> ::=
      (<RHS-slot-expr> <aggregate-function-expr>)

<aggregate-function-expr> ::=
      (<aggregate-function> <single-field-variable>)

<derived-attribute-conclusion> ::=
      [(calc <function-call>+)]
      <single-field-variable> <- (<RHS-class-expr> <simple-assign-expr>)

<aggregate-attribute-conclusion> ::=
      [(calc <function-call>+)]
```

```
      <single-field-variable> <- (<RHS-class-expr> <aggregate-assign-expr>)
```

`<rule-name> ::=`
A symbol which represents the name of a rule

`<variable-symbol> ::=`
A symbol beginning with an alphabetic character.

`<function-name> ::=`
Any symbol which corresponds to a system or user defined function, a deffunction name, or a defgeneric name

`<class-name> ::=`
A valid defclass name

`<slot-name> ::=`
A valid defclass slot name

`<aggregate-function> ::=`
A valid aggregate function name

Appendix

This appendix contains examples of R-DEVICE rules for sample RDF queries that have been obtained from "RDF Query and Rule languages Use Cases and Examples survey[2]".

```
(deductiverule q1
      ?x <- (? (email:message-id '123456@example.com'))
  =>
      (result (email ?x)))

(deductiverule q2
      ?x <- (? (vcard:N ?y))
      ?y <- (? (vcard:Family "Smith") (vcard:Given ?v))
  =>
      (person (name ?v)))

(deductiverule q3
      data:x <- (? (?property ?value))
      ?property <- (rdf:Property (rdfs:range $? ?t $?))
  =>
      (result (property ?property) (value ?value)(type ?t)))

(deductiverule q5
      (rss:item (rss:title ?title) (rss:link ?link))
  =>
      (result (title ?title) (link ?link)))

(deductiverule q6
      (rss:item (rss:title ?title) (rss:link ?link))
      (test (str-index "RDQL" ?title))
  =>
      (result (link ?link)))

(deductiverule q7
      ?x <- (? (dc:title ?tt) (dc:description ?dd) ((etbthes:ETBT dc:subject) ?ss2)
            (dc:identifier ?identifier) ((dcq:RFC1766 dc:language) ?language))
      ?tt <- (? (rdf:value ?t_val) ((dcq:RFC1766 dc:language) ?t_lang))
      ?ss2 <- (? (rdf:value ?subject_val) ((dcq:RFC1766 dc:language) ?subj_lang))
      ?dd <- (? (rdf:value ?desc_val) ((dcq:RFC1766 dc:language) ?desc_lang))
  =>
      (result (title_value ?t_val) (title_language ?t_lang) (subj_val ?subject_val)
            (subj_lang ?subj_lang) (desc_value ?desc_val) (desc_lang ?desc_lang)
            (language ?language)(identifier ?identifier)))
```

[2] `http://rdfstore.sourceforge.net/2002/06/24/rdf-query/`