Manual – RDD Checker (v2.11)

Michael Schmidt

April 14, 2014

Abstract

Although the intention of RDF is to provide an open, minimally constraining way for representing information, there exists an increasing number of applications for which guarantees on the structure and values of an RDF data set become desirable if not essential.

The RDF Data Description Language (RDD) was designed to tackle this problem. With RDDs, data maintainers can define a rich set of integrity constraints (such as keys, cardinality restrictions, path constraints, etc.) to tie RDF data sets to quality guarantees, akin to schemata of relational databases, or to DTDs in XML.

Making constraints explicit by means of an associated RDD description (which is coupled to a specific RDF data set or SPARQL endpoint) not only helps in maintaining data quality, but also eases the formulation of *precise* queries for users and system developers.

This handbook documents the RDD Checker v2.11, a tool that processes RDD descriptions and either (i) decompose them into a set of First-order Logic constraints for further investigation (e.g., they may serve as input to an FOL reasoner) or (ii) in a set of SPARQL queries which can be executed by any SPARQL 1.0 engine to verify the constraints. Further, the implementation comes with bindings for the SPARQL store Sesame, providing command line option to verify RDDs directly using Sesame against any RDF data set. The result is output as a report indicating verification times as well as constraints that succeeded and constraints that are violated in the input dataset.

Contents

1 RDD Checker – What is it?

Although the intention of RDF is to provide an open, minimally constraining way for representing information, there exists an increasing number of applications for which guarantees on the structure and values of an RDF data set become desirable if not essential. As an example, consider a software application running over an RDF database, which submits a fixed set of parametrized queries to implement the application logics. Here, it may be important to guarantee certain properties on the RDF data such that the queries return the expected results. For instance, one may one want to make sure that the queries contain no duplicate results, or avoid cross products in the result due to unexpected multi-valued properties.

The RDF Data Description Language (RDD) was designed to tackle such situations. With RDDs, one can define a rich set of integrity constraints (such as keys, cardinality restrictions, path constraints, etc.) to tie RDF data sets to quality guarantees, akin to schemata of relational databases, or to DTDs in XML. Thus, with RDDs at hand, programmers are enabled to write much more precise queries over RDF data sets, with predictable behavior.

1.1 Motivating Example

As a motivating example, assume a developer wants to write a SPARQL query that extracts information about persons in an RDF document, described by the properties `rdfs:label` (denoting the name), `foaf:age`, and `foaf:mbox` (mail address) – where every person shall be represented by exactly one row of the result table. While this sounds like a fairly trivial task, with the unconstrained RDF model it may become quite tricky, even if the schema (i.e., FOAF and RDF(S) vocabulary) is well known to the developer: without further knowledge about the *instance data*, the developer cannot be sure which predicates are present at all, which of them may be multi-valued, and which are not. Making guesses that `rdfs:label` and `foaf:age` are single-valued, the developer may finally come up with the following query:

```
SELECT ?person ?name ?age
       (GROUP_CONCAT(?mail; separator=", ")  AS ?mail)
WHERE {
   ?person rdf:type foaf:Person .
   OPTIONAL { ?person rdfs:label ?name }
   OPTIONAL { ?person foaf:age ?age }
   OPTIONAL { ?person foaf:mbox ?mail }
} GROUP BY ?name ?age
```

The OPTIONAL clauses ensure that persons with incomplete information are included in the result; to group persons with multiple email addresses, the developer used `GROUP BY` combined with `GROUP_CONCAT` in the `SELECT` clause, thus concatenating all email addresses of a single person. The crucial point here is that even this simple task leads to a quite complex query covering the "worst case scenario" anticipated by the developer, requiring the use of advanced SPARQL 1.1 constructs (which, as a matter of fact, are hard to optimize by query engines). And even this carefully designed query leads to multiple result rows for the same person in the presence of multiple labels (e.g., when the dataset contains multible labels with different language tags).

1.2 The RDD Language

RDDs allow to describe instance-level constraints that hold in a given RDF data set (or, for an associated SPARQL endpoint). RDDs are specified in a dedicated syntax, which adheres to the object-oriented character of the RDF data format. As such, they may greatly help developers in designing precise queries over the constrained data with predictable behavior.

At global scale, an RDD may consist of two (optional) parts:

1. A *class section* defines constraints for instances of RDF classes contained in the RDF data. For instance, one may want to express that a property `ex:matricNumber` forms the key for all instances of some class `ex:Student` and that this property always points to a literal typed as `xsd:integer`. As another example, one could specify the constraint that every `foaf:Person` typed instance in the RDF data graph has at most one `foaf:age` specification.

2. A *property section* defines global constraints over properties, i.e. constraints that do not depend on the context in which properties are used. One example could be a constraint asserting that every resource in the dataset (i.e., every IRI or blank node) has exactly one outgoing edge with property `rdfs:label` in the underlying RDF data.

An example RDD is given in Figure 1. At the beginning, the RDD starts with a definition of prefixes used inside the RDD, using the prefix syntax used in SPARQL queries. Next, the RDD defines the previously mentioned class and property sections, which are defined by the enclosing keywords `CLASSES` and `PROPERTIES`, respectively. The `CLASSES` section defines constraints for two individual `CLASS`es, namely `foaf:Person` and `ex:Student`.

Let us start with a description of the constraints for class `foaf:Person`.

```
PREFIX ex: <http://www.example.com#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CWA CLASSES {
  OWA CLASS foaf:Person SUBCLASS ex:Student {
    KEY rdfs:label : LITERAL;
    TOTAL foaf:mbox : LITERAL;
    PARTIAL foaf:age : LITERAL(xsd:integer);
    RANGE(foaf:Person) foaf:knows : IRI;
  }

  OWA CLASS ex:Student {
    TOTAL ex:matricNr : LITERAL(xsd:integer);
    MIN(1), RANGE(ex:Course) ex:course : RESOURCE;
    PATH(ex:course/ex:givenBy),
      RANGE(foaf:Person) ex:taughtBy : IRI;
  }
}

OWA PROPERTIES {
  TOTAL rdfs:label;
  SUBPROPERTY(foaf:knows) ex:taughtBy;
}
```

Figure 1: Sample RDD

- The `KEY` constraint specifies that the property `rdfs:label` serves as a key for persons, i.e. it uniquely identifies persons. While this, of course, may in the general case not always be true, the constraint tells a developer that in the dataset under consideration, this can be seen as a fact that is never violated. Note that by its semantics, the `KEY` implies that every instance of class `foaf:Person` has exactly one `rdfs:label`. The last part of the constraint, "`: LITERAL`", specifies that the property `rdfs:label`, when used for an instance of class `foaf:Person`, always points to a literal (i.e., never to a IRI or a blank node).

- The `TOTAL` constraint asserts that the property `foaf:mbox` is total w.r.t. class `foaf:Person` in the sense that every `foaf:Person` has *exactly one* email address specified. As for the key constraint, keyword `LITERAL` enforces that this property always points to a literal.

- The `PARTIAL` constraint asserts that the property `foaf:age` is partial w.r.t. class `foaf:Person` in the sense that every `foaf:Person` has *at most* one age specified; further, `LITERAL(xsd:integer)` enforces that this property, if present, points to a literal typed as `xsd:integer`.

- The `RANGE(foaf:Person)` constraint asserts that property `foaf:knows` always points to an instance typed as `foaf:Person`, which, in addition, must be represented by a IRI (and, e.g., not a blank node).

- The `SUBCLASS ex:Student` constraint in the header of the class definition asserts that all the four constraints mentioned before are inherited to instances of class `ex:Student`.

The constraints for class `ex:Student` are similiar in spirit. As described before, this class inherits all the constraints from class `foaf:Person`, and in addition has the following dedicated constraints specified in its `CLASS` section:

- All instances of `ex:Student` have exactly one `ex:matricNr` property, which points to an `xsd:integer`-typed Literal.

- The property `ex:course`, which always points to a `RESOURCE` (i.e., a IRI or a blank node, but not a literal) obeys two constraints, separated by the "`,`" symbol. First, the `MIN(1)` constraint enforces that it occurs at least one time (thus enforcing that every student is enrolled in at least one course); second, the `RANGE(ex:Course)` constraint specifies that this property always points to an instance that is typed as `ex:Course`.

- The constraints for property `ex:taughtBy` when used for instances of class `ex:Student` are: (i) it points to IRIs, (ii) `RANGE(foaf:Person)` implies that the teacher (to which the property points) is typed as `foaf:Person`, and (iii) the `PATH(ex:course/ex:givenBy)` implies that, all objects to which property `ex:taughtBy` points to can as well be reached when following a path along property chain `ex:couse` and `ex:givenBy`. Informally speaking, this constraint says that whenever a student is taught by someone, this someone must also be the person giving one of the courses the student is enrolled in.

The following `PROPERTIES` section contains two constraints:

- The first one, as discussed before, asserts that every resource has exactly one outgoing property labeled `rdfs:label`.
- The second constraint establishes a `SUBPROPERTY` relationship between `foaf:knows` and `ex:taughtBy`. This implies that in the RDF dataset, the subproperty relationship is materialized: whenever there is a triple (S,`ex:taughtBy`,O) then there is also a triple (S,`foaf:knows`,O) for any subject S and object O.

2 Installation and Usage

The RDD checker is implemented as a Java based command line tool that covers parsing, first-order logics translation, and SPARQL translation of the constraints specified in the RDD dataset. SPARQL queries can also be directly evaluated over a serialized RDF data set.

Thus, in all cases the RDD checker takes an RDD as input. Additionally, the following input/output combinations are possible:

- If no RDF data set is provided as input, the output either consists of the set of FOL constraints imposed by the RDD or a set of SPARQL queries. The SPARQL queries are designed in a way that they try to identify constraint violations in the database. Depending on the command line configuration, these may either be SPARQL `ASK` queries (thus returning *true* if a constraint violation could be detected) or SPARQL `SELECT` queries extracting constraint violating situations in the database.
- If, in addition, an RDF data set is provided, the checker is run in evaluation mode. It then loads the data into a Sesame[1] in-memory store and evaluates (either the `ASK` or `SELECT`) queries over the RDF data, and outputs an evaluation report, indicating the evaluation time and a verification report.

Within this section, we will sketch how to get to run the RDDChecker command line tool, In addition, the output format is discussed. The input syntax for RDDs will be described in Section 3, the semantics will be described in Section 4, and the output format for the first-order fomulas, the SPARQL queries, and the verification report will be described in Section 5.

The RDDChecker is available in two versions.

- First, a **binary package** containing the required .jar files is the easiest way to get started in case you just want to use the checker.
- Second, the **source package** contains a ready-to-use Eclipse project including the Java sources - it is the distribution of choice for developers who want to modify or extend the source code.

[1]`http://www.openrdf.org/`

2.1 Installing the Binary

The binary package provides a .jar distribution to run the RDDChecker from the command line.

2.1.1 Prerequisities

In order to use the binary package you need JRE 1.7 installed and the *java* command on the Windows classpath.

2.1.2 Installation

The binary distribution comes as a zip file, `RDDChecker2.11-bin.zip`. To use it, you need to proceed as follows:

1. Download the binary package `RDDChecker2.11-bin.zip`.
2. Unzip the .zip file to some `<INSTALL_FOLDER>`. Unpacking will create a subdirectory `<INSTALL_FOLDER>/RDDChecker2.11-bin`.
3. Open a Windows command line and change to the installation folder, i.e. `<INSTALL_FOLDER>/RDDChecker2.11-bin`.
4. You may now execute the RDDChecker by just running the script `RDDChecker.bat` contained in this folder (i.e., type this name into your command line window). The command line options you need to pass to the script are described in Section 2.3.

Note that the binary distribution also contains example RDD files to test your installation. They are located in folder `RDDChecker2.11-bin/examples`.

2.2 Setting up the Eclipse Project

The source distribution contains all resources you need to perform development on top of the RDDChecker. This includes source code plus a collection of test cases and associated example RDD files. The source distribution is packaged as a ready-to-use Eclipse project (but may also be imported and set up in other Java development environments, of course).

2.2.1 Prerequisities

You need Eclipse *Kepler* (other versions may work as well, but have not been tested). Eclipse must be set up to use JDK version 1.7 or higher. In addition, you need to install the JavaCC Eclipse plugin from the Eclipse marketplace.

2.2.2 Setup

The source distribution comes as a zip file, `RDDChecker2.11-src.zip`. To set it up, you need to proceed as follows:

1. Download the source package `RDDChecker2.11-src.zip`.
2. Open Eclipse *Kepler*, you may want to create a fresh workspace.
3. Select `File - Import - General - Existing Projects into Workspace`. Then choose the option `Select archive file` and choose the source package downloaded in the first step. Confirm with `Finish`. The RDDChecker will be imported as a new project called *rdd-checker*.

To test whether your installation has been successful, you may locate class `edu.freiburg.dbis.rdd.RDDChecker` in the *src* folder, right click on the class, choose `Run as - Java Application`. The output in the console should display usage information, i.e. parameters you need to pass.

In order to pass parameters, choose `Run as - Run configurations` and enter your arguments in tab `Arguments`, section `Program Arguments`. The command line options you need to pass are described in Section 2.3.

Note that the source distribution also contains example RDD files to test your installation. The examples in Section 2.3 make use of them. They are located in folder *examples*. A collection of JUnit tests is available in the folder *test*. An ANT build script, which allows you to build the .jar file from the sources, can be found in folder *build*.

In order to set up *log4j* (to get rid of the warnings that show up in the console when executing the tool), in the `Run configurations` choose `Classpath`, click on `User Entries`, choose `Advanced - Add Folders`, and select folder *rdd-checker/etc*.

2.3 Command Line Options

The command line options allow to configure the RDD checker. The command line string supports the following mandatory and optional options

```
RDDChecker.bat -r <arg> [-d <arg>] [-t <arg>] [-m <arg>]
                        [-o <arg>] [-e <arg>] [-l <arg>]
```

defined as follows:

`-h,--help` display help

`-r,--rdd <arg>` RDD input file

`-d,--data <arg>` RDF data file

`-t,--type <arg>` output format or evaluation type: ASK, SELECT, FOL (if not specified: ASK)

`-m,--mimetype <arg>` Mime type of RDF data input file (if not specified, guessed from file ending, default: application/rdf+xml)

`-o,--outputfile <arg>` output file (if not specified: STDOUT)

`-e,--errorfile <arg>` error file, summarizing all violated constraints (if not specified: not written)

`-l, --limit <arg>` number of witnesses to show for constraint violations, where 0 means all (if not specified: 3)

From these options, only the `-r` parameter specifying the RDD input file is obligatory. Note that the default output type (option `-t`) is ASK. When specifying an RDF data file using parameter `-d`, the output type specifies the evaluation type, i.e. whether `ASK` or `SELECT` queries are used for evaluation. In combination with an RDF file, evaluation mode FOL is not supported.

2.3.1 Examples Without Evaluation

Example 1. Run the RDD checker from command line using one of the example RDDs. Write the output, namely the SPARQL ASK queries derived from the RDD, to STDOUT.

```
RDDChecker.bat -r examples/test/semantics/MixedFeatures.rdd
```

Example 2. Run the RDD checker from command line using one of the example RDDs. Write the FOL formulas derived from the RDD to STDOUT.

```
RDDChecker.bat -r examples/test/semantics/MixedFeatures.rdd
  -t FOL
```

Example 3. Run the RDD checker from command line using one of the example RDDs. Write the SELECT queries derived from the RDD to file *out.txt* (in the top-level directory).

```
RDDChecker.bat -r examples/test/semantics/MixedFeatures.rdd
  -t SELECT -o out.txt
```

2.3.2 Examples With Evaluation

Example 1. Run the RDD checker from command line using one of the example RDDs and evaluate the ASK queries derived from the RDD on the given RDF file. Write the output report to STDOUT.

```
RDDChecker.bat -r examples/test/sp2bench/sp2bench_showcase.rdd
  -t ASK -d examples/test/sp2bench/sp2b-10000.n3
```

Example 2. Run the RDD checker from command line using one of the example RDDs and evaluate the SELECT queries derived from the RDD on the given RDF file, interpreting it as NTriples file (note that the mime type of NTriples is text/plain). Store the output in file *out.txt* (in the top-level directory) and the report indicating failed constraints only in file *errorReport.txt* (in the top-level directory).

```
RDDChecker.bat -r examples/test/sp2bench/sp2bench_showcase.rdd
  -t SELECT -d examples/test/sp2bench/sp2b-10000.n3
  -o report.txt -e errorReport.txt
```

3 The RDD Language

3.1 Overview

This section provides a reference to the syntax of the RDD language. For an introducing example we refer back to Section 1.1. A structural overview of the RDD language in UML-style notation is shown in Figure 2.

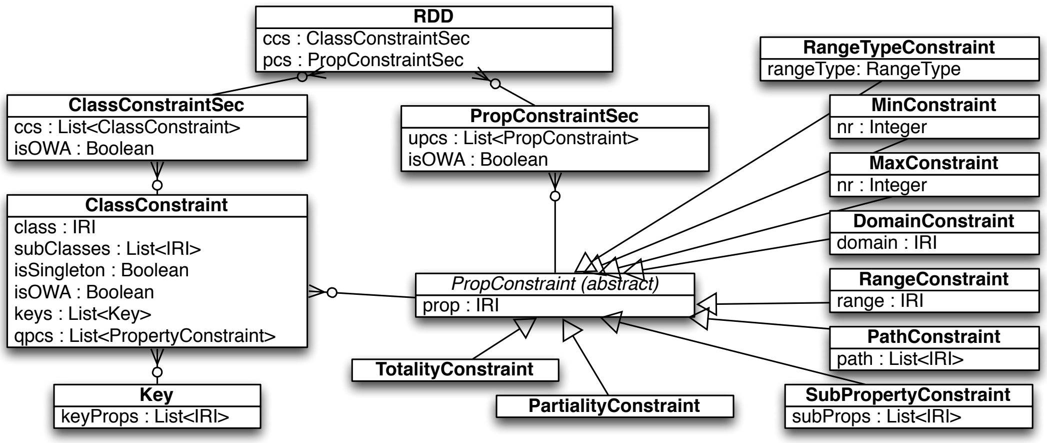

Figure 2: Structural Overview of the RDD Language

Boxes in the figure denote concepts, arrowed lines sub-concept relationships, and the second line type indicates that concept A uses B. At top-level, RDDs consist of a ClassConstraintSec and a PropConstraintSec, which contain lists of ClassConstraints and PropConstraints, respectively, plus a boolean flag indicating whether the sections should be interpreted under Open or Closed World Assumption (i.e., whether the classes and properties in the sections describe the RDF data completely or nor) – the `OWA` keyword enables RDDs to be written in a pay-as-you-go fashion, where known constraints are specified, while unknown parts are left unspecified.

Central to the RDD concept is the notion of PropConstraints, an abstract concept that is further subclassed into specific subclasses such as RangeTypeConstraint MinConstraint, MaxConstraint, etc.

PropConstraints are used in two different contexts: (1) The PropConstraintSec (cf. keyword `PROPERTIES`) contains a list *upcs* of PropConstraints, implementing *unqualified*, global characteristics of properties. For instance, the TotalityConstraint in Fig. 1 (keyword `TOTAL`) for *prop* :=*rdfs:label* asserts that *every resource* has exactly one label. (2) Variable *qpcs* inside ClassConstraints represents *qualified*, class-specific

PropConstraints, e.g. the MinConstraint (keyword `MIN`) for the property *ex:course* and $nr := 1$ in Fig. 1 in the class section of *ex:Student* ensures that *every instance of ex:Student* visits at least one course.

In addition to *qpcs*, a ClassConstraint contains (i) a list of subclasses (keyword `SUBCLASS`), enforcing that instances of the subclasses inherit inner constraints of the superclass, (ii) a boolean flag *isSingleton*, enforcing that exactly one instance of the class exists, (iii) the *isOWA* flag, and (iv) a list of keys. We sketch their semantics in the next section.

3.2 Full Grammar

In the following, we provide the abstract grammar for RDDs; the individual constraint types will be explained in detail in Section 4. The start symbol of the grammar is ⟨*RDD*⟩, terminals are marked in **bold** font. For simplicity, we do not include comments in the grammar, which can be made using `//` (single line comments) and opening `/*` with closing `*/` (multi-line comments), like in common object-oriented programming languages such as C++ or Java.

⟨*RDD*⟩ ::= ⟨*PrefixDecl*⟩* ⟨*ClassConstraintSec*⟩ ⟨*PropConstraintSec*⟩

⟨*ClassConstraintSec*⟩ ::= ⟨*WA*⟩ **CLASSES {** ⟨*ClassConstraint*⟩* **}**
⟨*PropConstraintSec*⟩ ::= ⟨*WA*⟩ **PROPERTIES {** ⟨*PropConstraint*⟩* **}**

⟨*ClassConstraint*⟩ ::= ⟨*WA*⟩ (**SINGLETON**)? **CLASS** ⟨*IRI*⟩
  (**SUBCLASS** ⟨*IRIList*⟩)?
  { (⟨*Key*⟩ | ⟨*PropConstraint*⟩)* **}**
⟨*Key*⟩ ::= **KEY** ⟨*IRIWithRangeTypeList*⟩ **;**
⟨*PropConstraint*⟩ ::= ⟨*ConstraintList*⟩? ⟨*IRIWithRangeType*⟩ **;**

⟨*ConstraintList*⟩ ::= ⟨*Constraint*⟩ (**,** ⟨*Constraint*⟩)*
⟨*Constraint*⟩ ::= ⟨*MinConstraint*⟩ | ⟨*MaxConstraint*⟩ |
  ⟨*DomainConstraint*⟩ | ⟨*RangeConstraint*⟩ |
  ⟨*PartialityConstraint*⟩ | ⟨*TotalityConstraint*⟩ |
  ⟨*PathConstraint*⟩ | ⟨*SubPropertyConstraint*⟩

⟨*MinConstraint*⟩ ::= **MIN(**⟨*INTEGER*⟩**)**
⟨*MaxConstraint*⟩ ::= **MAX(**⟨*INTEGER*⟩**)**
⟨*DomainConstraint*⟩ ::= **DOMAIN(**⟨*IRI*⟩**)**
⟨*RangeConstraint*⟩ ::= **RANGE(**⟨*IRI*⟩**)**
⟨*PathConstraint*⟩ ::= **PATH(**⟨*IRISeq*⟩**)**
⟨*SubPropertyConstraint*⟩ ::= **SUBPROPERTY(**⟨*IRIList*⟩**)**

$\langle$*PartialityConstraint*$\rangle$ ::= **PARTIAL**

$\langle$*TotalityConstraint*$\rangle$ ::= **TOTAL**

$\langle$*WA*$\rangle$ ::= **OWA** | **CWA**

$\langle$*IRIList*$\rangle$ ::= $\langle$*IRI*$\rangle$ (**,** $\langle$*IRI*$\rangle$)*

$\langle$*IRISeq*$\rangle$ ::= $\langle$*IRI*$\rangle$ (**/** $\langle$*IRI*$\rangle$)*

$\langle$*IRIWithRangeTypeList*$\rangle$ ::= $\langle$*IRIWithRangeType*$\rangle$
(**,** $\langle$*IRIWithRangeType*$\rangle$)*

$\langle$*IRIWithRangeType*$\rangle$ ::= $\langle$*IRI*$\rangle$ (**:** $\langle$*RangeType*$\rangle$)?

$\langle$*RangeType*$\rangle$ ::= **IRI** | **BNODE** | **RESOURCE** | **LITERAL**((**$\langle$*IRI*$\rangle$**))?

$\langle$*PrefixDecl*$\rangle$ ::= as defined in rule *[6] PrefixDecl* in [1]

$\langle$*IRI*$\rangle$ ::= as defined in rule *[136] iri* in [1]

$\langle$*PrefixedName*$\rangle$ ::= as defined in rule *[137] PrefixedName* in [1]

$\langle$*IRIREF*$\rangle$ ::= as defined in rule *[139] IRIREF* in [1]

$\langle$*INTEGER*$\rangle$ ::= as defined in rule *[146] INTEGER* in [1]

4 Semantics

The RDD checker implements the semantics defined [2]. Thus, we refer the interested reader to the latter publication for a precise definition of the semantics.[2] In this handbook, we instead sketch and quickly describe the constraint types informally (Section 4.1).

4.1 Semantics of Constraint Types

4.1.1 Semantics of PropConstraints

In this subsection we sketch the semantics and examples for the individual property constraint types. Note that these constraints can be used either inside a `CLASS` section (thus restricting the property use for instances of the class) and inside the global `PROPERTIES` section, thus specifying unqualified/global property restrictions.

RangeTypeConstraint

RangeTypeConstraints allow for the specification of a range type, which indicates that the property points to either a IRI, Blank Node, Resource, or a (possibly typed) Literal. The associated keywords are are `IRI`, `BNODE`, `RESOURCE` (either IRI or blank node), and `LITERAL`. For `LITERAL`, an optional datatype restriction may be specified in parantheses. The following example asserts that property `foaf:age` points to a literal of type `xsd:integer`.

```
foaf:age : LITERAL(xsd:integer)
```

Min/MaxConstraint

A Min/MaxConstraint indicates that the associated property occurs at least or at most a certain number of times, respectively. The following examples assert that property `ex:course` (`foaf:knows`) appear at least (at most) one (three) times per instance.

```
MIN(1) ex:course
MAX(3) foaf:knows
```

Domain/RangeConstraint

[2]In the course of the evaluation and testing process, we observed some practicability issues and slightly modified the semantics in one case: If the property constraint section is marked as CWA, we by default include rdf:type into the list of properties (even though it may not be explicitly listed in the properties section), as this property is crucial for RDFs and needs not to be listed explicitly.

A DOMAIN/RANGECONSTRAINT indicates a guaranteed type for the subject and object pointed to by a certain property, respectively. The following two constraints assert that property `foaf:knows` always points from instances of `foaf:Person` (`DOMAIN`) to instances of `foaf:Person` (`RANGE`).

```
DOMAIN(foaf:Person) foaf:knows
RANGE(foaf:Person) foaf:knows
```

PathConstraint

A PATHCONSTRAINT indicates that the value to which a property points can as well be reached by following a given sequence of properties. The following example asserts that all objects reached by property `ex:taughtBy` can as well be reached by following the (more precisely, one possible) path along the two properties `ex:course` and `ex:givenBy`.

```
PATH(ex:course/ex:givenBy) ex:taughtBy
```

SubPropertyConstraint

A SUBPROPERTYCONSTRAINT indicates that for every triple using the indicated subproperty, there is also a corresponding triple (i.e., with identical subject and object), for the property under considertaion. In this line, the following example asserts that a triple `(s,ex:taughtBy,o)` implies the existence of a triple `(s,foaf:knows,o)`.

```
SUBPROPERTY(ex:taughtBy) foaf:knows
```

Totality/PartialityConstraint

TOTALITY/PARTIALITYCONSTRAINTs express that the associated property occurs exactly or at most one time, respectively. In this line, the following example specifies that predicate `rdfs:label` occurs exactly one time per instance, and `foaf:age` occurs at most once.

```
TOTAL rdfs:label
PARTIAL foaf:age
```

4.1.2 CWA/OWA Constraints

CWA/OWA constraints in CLASSES section

A CWA constraint used as part of the `CLASSES` section asserts that the classes defined in the `CLASSES` section are complete, i.e. there are no instances typed with classes other than those defined in the `CLASSES` section. In this line, the following example imposes a constraint saying that all instances in the data set are typed as `foaf:Person` or as `ex:Student`.

```
CWA CLASSES {
  OWA CLASS foaf:Person {
    ...
  }
  OWA CLASS ex:Student {
    ...
  }
}
```

Using keyword OWA instead of CWA implies no such completeness constraint.

CWA/OWA constraints in CLASS section

A CWA constraint used as part of a specific CLASS section asserts that the properties listed in the CLASS section are complete w.r.t. instances of the class, i.e. instances of the class have no other properties than those listed in the CLASS section. In this line, the following example imposes a constraint saying that only the properties rdfs:label, foaf:firstName, foaf:lastName, foaf:age, and foaf:mbox are used to describe instances of class foaf:Person.

```
OWA CLASSES {
  CWA CLASS foaf:Person {
    TOTAL rdfs:label;
    foaf:firstName;
    foaf:lastName;
    foaf:age;
    foaf:mbox;
  }
}
```

Using keyword OWA instead of CWA implies no such completeness constraint.

CWA/OWA Constraints in PROPERTIES section

A CWA constraint used as part of the PROPERTIES section asserts that the properties listed in the PROPERTIES section are complete, i.e. there are no other properties occurring in the RDF data set than those listed in the PROPERTIES section. In this line, the following example imposes a constraint saying that only properties rdfs:label, foaf:firstName, foaf:lastName, foaf:age, and foaf:mbox are used in the RDF data set.

```
CWA PROPERTIES {
    TOTAL rdfs:label;
    foaf:firstName;
    foaf:lastName;
    foaf:age;
    foaf:mbox;
  }
```

Using keyword OWA instead of CWA implies no such completeness constraint.

4.1.3 KeyConstraint

A KeyConstraint can be used inside a CLASS section only. It expresses that the associated list of properies forms a key for instances of the class, namely (i) every property is present exactly once for every instance of the class, and (ii) no other instance of the class points to the same values via these properties. The example below defines a composed key formed by the properties rdfs:label and foaf:mbox. This means that two persons may coincide in the values of either rdfs:label or foaf:age, but the combination of both properties is unique.

```
OWA CLASS foaf:Person {
   KEY rdfs:label, foaf:age;
}
```

4.1.4 SINGLETON Constraint

A SINGLETON keyword associated with a CLASS section specifies that the class has exactly one instance. As an example, the following constraint expresses that there is exactly one foaf:Person typed instance in the data set.

```
OWA SINGLETON CLASS foaf:Person {
   ...
}
```

4.1.5 SUBCLASS Constraint Inheritance

RDDs offers a mechanism to inherit constraints to subclasses. To this end, the SUBCLASS keyword can be used in concunction with a class, followed by a (non-empty) list of subclasses. The following example specifies that class foaf:Person has two subclasses, namely ex:Student and ex:Professor. As a consequence, all constraints specified in the body of the class do not only hold for instances of class foaf:Person, but also apply to instances of ex:Student and ex:Professor. More precisely, in the example below instances of all three classes have exactly one property `rdfs:label`, which is always pointing to a `LITERAL`.

```
OWA CLASS foaf:Person SUBCLASS ex:Student, ex:Professor {
   TOTAL rdfs:label : LITERAL;
}
```

Note that this mechanism works recursively, e.g. if in the example above there is a `CLASS` section for `ex:Professor` defining `ex:AssociateProfessor` as a subclass of `ex:Professor`, then the constraints are recursively inherited to subclass `ex:AssociatProfessor` as well (and its subclasses, if any).

5 Output Format

5.1 Evaluation Mode

Whenever an RDF file is specified in addition to the RDD input file (parameter `-d`), the tool will parse the RDD, set up SPARQL queries to check the constraints (either `ASK` or `SELECT` queries, depending on parameter `-t`), load the data into a Sesame memory store [3], evaluate the queries over the memory store, and output an evaluation report. A fragment of the output report for RDD *examples/test/sp2bench/sp2bench_showcase.rdd* evaluated over RDF document *examples/test/sp2bench/sp2b-10000.n3* is shown below.

```
############### The RDF document is INCONSISTENT with the RDD
############### Evaluation summary
############# RDD constraints success/failure: 89/6
############# FOL constraints success/failure: 109/6
############# Evaluation time: 1199ms


############### Complete evaluation report
############# (0) Class specific constraints
########### (0.0) Constraints for class <http://localhost/vocabulary/bench/Journal>
######### (0.0.0) [CLASS CWA] constraint
####### Evaluation time: 3ms
####### Result: success
####### Implied FOL constraints: 1 total (1 succeeded, 0 failed)
##### (0.0.0.0) Implied FOL constraint
## FO Formula:
Forall ?s1 ?p1 ?o1 (
  Or(?p1=<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
     ?p1=<http://swrc.ontoware.org/ontology#editor> ?p1=<http://purl.org/dc/elements/1.1/title>
     ?p1=<http://purl.org/dc/terms/issued> ?p1=<http://swrc.ontoware.org/ontology#volume>
     ?p1=<http://swrc.ontoware.org/ontology#number>)
  :- And(?s1[<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ->
          <http://localhost/vocabulary/bench/Journal>] ?s1[?p1 -> ?o1]))
## Query:
SELECT * {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://localhost/vocabulary/bench/Journal> .
?s1 ?p1 ?o1 .
 FILTER (!(?p1=<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ||
          ?p1=<http://swrc.ontoware.org/ontology#editor> ||
          ?p1=<http://purl.org/dc/elements/1.1/title> ||
          ?p1=<http://purl.org/dc/terms/issued> ||
          ?p1=<http://swrc.ontoware.org/ontology#volume> ||
          ?p1=<http://swrc.ontoware.org/ontology#number> ))} LIMIT 3
## Evaluation time: 3ms
## Result: success

######### (0.0.1) [Qualified RangeTypeConstraint] constraint
####### Textual representation:
<http://purl.org/dc/elements/1.1/title> : LITERAL(<http://www.w3.org/2001/XMLSchema#string>)
####### Evaluation time: 4ms
####### Result: success
####### Implied FOL constraints: 1 total (1 succeeded, 0 failed)
##### (0.0.1.0) Implied FOL constraint
## FO Formula:
Forall ?s1 ?o1 (And(LITERAL(?o1) DATATYPE(?o1,<http://www.w3.org/2001/XMLSchema#string>)) :-
                And(?s1[<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ->
                     <http://localhost/vocabulary/bench/Journal>]
                    ?s1[<http://purl.org/dc/elements/1.1/title> -> ?o1]))
## Query:
SELECT * {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://localhost/vocabulary/bench/Journal> .
?s1 <http://purl.org/dc/elements/1.1/title> ?o1 .
 FILTER (!(isLiteral(?o1) &&
          (datatype(?o1)=<http://www.w3.org/2001/XMLSchema#string>) ))} LIMIT 3
## Evaluation time: 4ms
## Result: success

...

############ (1) Unqualified property constraints
######## (1.0) [Unqualified TotalityConstraint] constraint
###### Textual representation:
TOTAL <http://www.w3.org/2000/01/rdf-schema#label>;
###### Evaluation time: 5ms
###### Result: failure
###### Implied FOL constraints: 2 total (1 succeeded, 1 failed)
#### (1.0.0) Implied FOL constraint
## FO Formula:
Forall ?s1 ?o1 ?o2 (?o1=?o2 :-
  And(?s1[<http://www.w3.org/2000/01/rdf-schema#label> -> ?o1]
      ?s1[<http://www.w3.org/2000/01/rdf-schema#label> -> ?o2]))
## Query:
SELECT * {
?s1 <http://www.w3.org/2000/01/rdf-schema#label> ?o1 .
?s1 <http://www.w3.org/2000/01/rdf-schema#label> ?o2 .
 FILTER (!(?o1=?o2 ))} LIMIT 3
## Evaluation time: 2ms
## Result: success
#### (1.0.1) Implied FOL constraint
## FO Formula:
Forall ?s1 (Exists ?o1 (?s1[<http://www.w3.org/2000/01/rdf-schema#label> -> ?o1]) :- RESOURCE(?s1))
## Query:
SELECT * {
   { ?s1 ?v1 ?v2} UNION { ?v3 ?s1 ?v4 } UNION { ?v5 ?v6 ?s1 }
   FILTER (isIri(?s1) || isBlank(?s1))
   FILTER NOT EXISTS { ?s1 <http://www.w3.org/2000/01/rdf-schema#label> ?o1 }} LIMIT 3
## Evaluation time: 3ms
## Result: failure
## Counterexamples (showing 3 counterexamples -> limit reached, there may be more):
v1 -> http://www.w3.org/2000/01/rdf-schema#subClassOf,
  s1 -> http://localhost/vocabulary/bench/Journal, v2 -> http://xmlns.com/foaf/0.1/Document
v1 -> http://www.w3.org/2000/01/rdf-schema#subClassOf,
  s1 -> http://localhost/vocabulary/bench/Proceedings, v2 -> http://xmlns.com/foaf/0.1/Document
v1 -> http://www.w3.org/2000/01/rdf-schema#subClassOf,
  s1 -> http://localhost/vocabulary/bench/Inproceedings, v2 -> http://xmlns.com/foaf/0.1/Document



############ (2) CLASSES CWA constraint
######## (2.0) [CLASSES CWA] constraint
###### Evaluation time: 1ms
###### Result: failure
###### Implied FOL constraints: 1 total (0 succeeded, 1 failed)
#### (2.0.0) Implied FOL constraint
## FO Formula:
Forall ?s1 ?o1 (Or(?o1=<http://localhost/vocabulary/bench/Inproceedings>
                ?o1=<http://localhost/vocabulary/bench/Journal>
```

```
               ?o1=<http://localhost/vocabulary/bench/Proceedings>
               ?o1=<http://xmlns.com/foaf/0.1/Person>
               ?o1=<http://xmlns.com/foaf/0.1/Document>)
               :- ?s1[<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> -> ?o1])
## Query:
SELECT * {
?s1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?o1
FILTER (!(?o1=<http://localhost/vocabulary/bench/Inproceedings> ||
          ?o1=<http://localhost/vocabulary/bench/Journal> ||
          ?o1=<http://localhost/vocabulary/bench/Proceedings> ||
          ?o1=<http://xmlns.com/foaf/0.1/Person> ||
          ?o1=<http://xmlns.com/foaf/0.1/Document> ))} LIMIT 3
## Evaluation time: 1ms
## Result: failure
## Counterexamples (showing 3 counterexamples -> limit reached, there may be more):
s1 -> http://localhost/publications/articles/Journal1/1940/Article1,
  o1 -> http://localhost/vocabulary/bench/Article
s1 -> _:9a2e25124fd645eca2a167e1fb98f231references1,
  o1 -> http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
s1 -> http://localhost/publications/articles/Journal1/1940/Article2,
  o1 -> http://localhost/vocabulary/bench/Article


############ (3) PROPERTIES CWA constraint
(no PROPERTIES CWA constraint specified)
```

The first section (separated by a blank line) indicates some global stats, namely

- whether the RDF document is `CONSISTENT` or `INCONSISTENT` with the RDD in overall (the example shows the output for an inconsistent document),
- the number of RDD constraints, i.e. constraints as specified by the user inside the RDD, that were evaluated successfully (or failed),
- the number of FO constraints, which are obtained from the RDD constraints by applying the semantics of RDDs, that were evaluated successfully or not; as the example shows, this number in general is higher than the number of RDD constraints, because a single RDD constraint may be checked via one or more FO constraints, and
- the total evaluation time, measured as the time needed for evaluating all the queries derived from the RDD.

What is following is a detailed report for all the RDD constraints, split into four sections:

- Section "**(0) Class specific constraints**" shows class specific constraints, ordered by classes. More precisely, the second number of identifiers in section 0 specifies the class: 0.0 stands for the first class, 0.1

for the second, etc.; the third number specifies the constraint: 0.0.0 stands for the first constraint of the class 0, 0.0.1 stands for the second constraint of class 0, etc.; the fourth number indicates the FOL constraint in which the constraint was decomposed.

To illustrate the output by example, the first class considered is (0.0) `http://localhost/vocabulary/bench/Journal`. The first constraint imposed to this class is the (0.0.0) `CLASS CWA` constraint, which was evaluated in 3ms and holds on the document. It was checked by executing the SPARQL query associated with FOL constraint (0.0.0.0) which indicated successful verification of the constraint in 3ms. The second constraint over the class is the qualified range type constraint (0.0.1), specifiying that property title, for every instance of the class, is pointing to an xsd:literal typed object; it also was verified successfully.

- Section "**(1) Unqualified property constraints**" summarizes the constraints specified in the `PROPERTIES` section. Its structure is analogous to the structure of section (0). In the example above, this section contains the single RDD constraint (1.0) `TOTAL rdfs:label`, which did not hold on the input document. More precisely, the latter constraint was mapped into two FOL constraints (and two associated queries), where the first FOL constraint (1.0.0) asserts that the property is not multi-valued (which holds), and the second FOL constraint (1.0.1) asserts that the property must occur for all resources (which fails, as witnessed by the counterexamples).
- Section "**(2) CLASSES CWA constraint**" summarizes the result of the `CWA` constraint defined over the `CLASSES` section, if present. In the example above, this constraint failed, and the counterexamples section indicates some instances and their respective types (i.e., classes) that are not specified in the RDD.
- Section "**(3) PROPERTIES CWA constraint**" summarizes the result of the `CWA` constraint defined over the `PROPERTIES` section, if present. In the example above, no such constraint was specified.

5.2 Non-Evaluation Mode – SPARQL ASK and SELECT

In case the output type `ASK` and `SELECT` is used and no RDF input file is given, the ASK and SELECT queries are output. This mode makes sense if you want to extract the verification queries for evaluation in a proprietary SPARQL engine. The queries are output one by one in UTF-8 format.

5.3 Non-Evaluation Mode – First-order Constraints

The translation of the constraints in first-order constraints, which is output when type `FOL` is used, follows the formal semantics defined in [2]. The output of the RDDChecker is thus a set of first-order formulas. Regaring the first-order syntax, the RDDChecker output is a (a slightly extended) version of the RIF RDF and OWL derivate [4]. We refer the interested reader to the standard for a detailed explanation of the standard and illustrate the output by example. Let us consider the following small RDD.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

OWA PROPERTIES {
  MIN(1) rdfs:label;
}
```

As discussed before, this RDD implies a single constraint, saying that every resource has at least one outgoing property `rdfs:label`. When processing the RDD with the RDDChecker, the following output will be generated:

```
Forall ?s1 (
  Exists ?o1 (
    ?s1[<http://www.w3.org/2000/01/rdf-schema#label> -> ?o1])
  :- RESOURCE(?s1)
)
```

Note that the symbol "`:-`" separates the head of an implication (left side) from its body (right side). Hence, this output can be read as follows: *For every subject ?s1 that is contained in relation* `RESOURCE` *(which is a special relation containing all the resources in the RDF data set), there exists some ?o1 such that ?s1 points has an outgoing property rdfs:label pointing to ?o1.*

In case the RDD implies multiple constraints, the output contains multiple lines, each of them containing one constraint.

6 Acknowledgments

RDDChecker and its theoretical foundations were supported by the German Federal Ministry of Economics and Technology as part of the project Durchblick, grant KF2587503BZ2, and by Deutsche Forschungsgesellschaft as part of the project CORSOS, grant LA 598/7-1.

Kudos to the RDD team, namely Prof. Georg Lausen, Peter Fischer, Alexander Schätzle, and Philipp Schledermann (all from Freiburg University) for helpful discussions and hints during the development of the language and the implementation of the RDD checker.

References

[1] "SPARQL 1.1 Query Language," www.w3.org/TR/sparql11-query/.

[2] M. Schmidt and G. Lausen, "Pleasantly Consuming Linked Data with RDF Data Descriptions," 2013, TR, arXiv (submit/0758082).

[3] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *The Semantic WebISWC 2002*. Springer, 2002, pp. 54–68.

[4] "RIF RDF and OWL Compatibility (Second Edition)," http://www.w3.org/TR/rif-rdf-owl/.