Definition of and Requirements for RDF Validation

(Position paper for the W3C RDF Validation Workshop - 10 & 11 September 2013)

Harold Solbrig
Mayo Clinic

The primary advantage of RDF is that it provides a single, consistent (note) structure, meaning that basic tools and applications that manage, query and update RDF graphs need only be written once. The model behind RDF is simple, easy to comprehend and universal.

This simplicity, however, comes at a price – there is very little that one can assume about a specific RDF graph. The only iron clad guarantee is that the graph consists of a set of triples[1] – a subject, a predicate and an object and that the subject will either be a URI or a blank node, the predicate a URI and the object a URI, blank node or literal.

Even with these drawbacks, RDF is thriving as a *secondary* medium, where the data in an RDF graph is derived from external resources – resources that were created using existing tools such as SQL Tables, XML Documents, UML Models and the accompanying software that enforced naming, content and relationship constraints across the collection of resources. What is missing, however, is a standardized, formal mechanism that allows the same type or rules available in the source resources to be expressed directly about a given RDF "store".

As a result, there is no easy way to prevent information from being added to "store" using different names, inconsistent structures or unexpected and potentially invalid relationships. A simple example of the sort of issue that this problem produces is described below:

> A developer runs a set of queries across an RDF triple store and, assuming that the queries were constructed correctly (!), concludes that all subjects of type ":Person" also were the subject of a :firstName and a :lastName predicate and the objects of both the :firstName and :lastName predicates were always literals. Given this information, the developer constructs the following query:

[1] Even this assumption is somewhat weak, in that a reified "triple" could consist of a subject, predicate, but fail to declare an object.

```
SELECT ?first ?last
WHERE
{   ?person rdf:type :Person.
    ?person :firstName ?first.
    ?person :lastName ?last.
}
```

The developer has no guarantees beyond what may have been published by the store itself that:

1) A new subject of type :Person may be entered into the store without an accompanying :firstName, :lastName or both.
2) That the target of :firstName will always be a literal and not another URI or a blank node
3) That new person information won't be entered using the :given and :surname predicates to represent the same information.

Not surprisingly, a not insignificant portion of any SPARQL document involves techniques for anticipating and addressing the sort of situations described above and may others. Today, any application that queries RDF Graphs has to either:
(a) Assume that *any* possible combination of triples can occur and write defensive code that consists of collections of optional parameters
(b) Assume that certain invariants apply to the store itself.

The problem with (a) is that it consumes time and resources and, no matter how thoroughly though out, there is no guarantee that two developers will arrive at the same solutions, meaning that two "identical" queries may produce radically different and potentially incompatible output.

The problem with (b) is that, unless the assumptions are published as a part of the store itself, there is no guarantee that any two developers make the same assumptions or the assumptions are complete or correct.

We believe that the topic currently described under the rubric, "RDF Validation", describes the ability to:

a) publish
b) discover
c) interpret
d) apply

a set of *invariants* that apply to what we will call an "RDF Store"

The term, "RDF Graph" references a set of RDF triples[2], and two RDF graphs are considered to be equivalent if and only if they contain the same set of triples with the exception of the names assigned to blank nodes[3]. "RDF Graph" can be used to reference a textual document formatted in RDF XML, Turtle or other syntax, the state of a "triple store" at a given point in time, the result of a SPARQL Query or any other resource or function whose output is a set of zero or more triples that is compliant with the abstract RDF semantics.

Basic set theory can be used to describe various relationships (e.g. "subgraph", "entailment", etc.) that could exist between two or more RDF Graphs, but additional constructs are needed to describe the set of possible RDF Graphs that are considered to be "valid" in a given context. A set of rules that describe the valid states of are often referred to as "invariants" – "a property of a class of mathematical objects that remains unchanged when transformations of a certain type are applied to the objects. The particular class of objects and type of transformations are usually indicated by the context in which the term is used.[4]"

In the case of RDF, the class of objects is known (RDF Triples), the type of transformation is fairly straightforward – a two part operation that adds (unions) one graph with a second and then removes (set minus) another graph from the result of the first operation. The part that is not so clear, however, is the *context*. When we talk about "RDF Validation", we are talking about declaring a set of invariants that are (declared to be) true about *all possible graphs* -- a concept that, while useful, is a tad weak when we are talking about validating an RDF document, triple store or other resource.

For the purposes of this discussion, we propose a new term, "*RDF Store*," a quad consisting of:

- An **identity** – an immutable constant that identifies an instance of store
- An **RDF Graph** that represents the state of the Store at a given point in time
- An **update function** that takes 3 graphs:
    - The RDF Graph that represents the current state of the RDF Store
    - An RDF Graph to be added (unioned) with the first argument
    - An RDF Graph to be removed (set minus) from the first argument

  and, if the result of applying the operation to the current state of the RDF graph yields a graph the is true for the set of invariants, replaces the current state of the RDF Store with the resultant Graph
- A set of **invariants** – a set of predicates that define the (potentially infinite) set of "valid" graphs for the store

[2] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#dfn-rdf-graph

[3] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-graph-equality

[4] http://en.wikipedia.org/wiki/Invariant_%28mathematics%29

We would argue that the focus of this workshop and subsequent follow on should be to:

1) Identify or create a standard syntax and semantics for expressing invariants against an RDF graph
2) Describe how an RDF Store can be recognized and how the identity and invariants can be discovered
3) Describe how invariants should be enforced – describe the behavior of the update function and how it can be used to validate or apply an addition/removal tuple to the current state of the store
4) (Extra – could be delayed) – Address how the invariants can be updated (meta-invariants) and describe what transformations can occur without changing the RDF Store identity, potentially tying the rules into the notion of "Semantic Versioning[5]

Diving into the next level of detail, we believe that the invariant "language" should meet the following requirements:

1) The description of the graph invariants should be expressed in a standardized formal language.
    a. The formal language should be expressible in the form of RDF graph
    b. The formal language should be able to be validated for consistency
    c. The formal language should be self-defining – it should be able to describe a strong enough set of invariants about itself that only valid expressions in the formal language should not be admitted.
        i. Note that this does not cover the notion of logical consistency
2) The invariant language should be expressive enough to include:
    a. The set of assertions that can be expressed in UML 2.x class and attributes assertions. (Exceptions? Derived / Default / Primitive Types)
    b. The set of assertions that can be expressed in XML Schema
3) The invariant language should, to the extent possible, take advantage of existing, widely available tooling – it should be possible to express and implement the invariant language with little or no new tooling development.

[5] http://semver.org/