Simple Application-Specific Constraints for RDF Models

Shawn Simister
simister@google.com

Dan Brickley
danbri@google.com

When building applications that rely on RDF data, the validity of that data for a specific application or use case can be simply represented by a set of required property paths and value constraints. In our experience designing and building a validation tool [1] for Schema.org [2] markup in emails (Microdata [3], RDFa [4] and JSON-LD [5]) for use in Google Search, Google Now and Gmail [6] we've found that what may seem like a potentially infinite number of possible constraints can be represented quite succinctly using extisting standards like the SPARQL query language [7] and serialized as RDF.

The benefit of ingesting RDF data in our applications is that it allows us to accept arbitrarily detailed descriptions of the entities that appear within emails. The challenge of interpreting that data in our application is that the application which is consuming the data it will require certain constraints to be satisfied in order to be able to do something useful with the data. These constraints are defined in the source code of each application but are not visible to anyone outside the organization and are difficult to maintain and reconcile across different products. By defining all of these constraints in a simple, declarative syntax we can share these constraints internally and externally in a way that is not tied to any one tool or implementation.

Our tool is not the first to validate RDF models using constraints represented as SPARQL queries and property paths. One of the first approaches that informed our design was Schematron [10]. Schematron uses XPath expressions [11] to look for required paths in XML documents. What is interesting about Schematron is that it uses two path expressions: one to find the context node which is being validated and another to check the constraint on that node. This simple concept of a context query and a constraint query is surprisingly powerful and is central to how our tool validates constraints. Based on the approach set forth by Schematron, Dan Brickley and Libby Miller built Schemarama [12] which demonstrated that it was possible to apply the same path-based validation to RDF models using the Squish[c] query language. Finally, the SPIN Modeling Vocabulary [13] has features which demonstrate how to model constraints using SPARQL ASK queries and store those constraints as part of the RDF model.

We'll start by examining how our validator works and then go into more detail about what alternatives we considered and the reasons why we made the design decisions

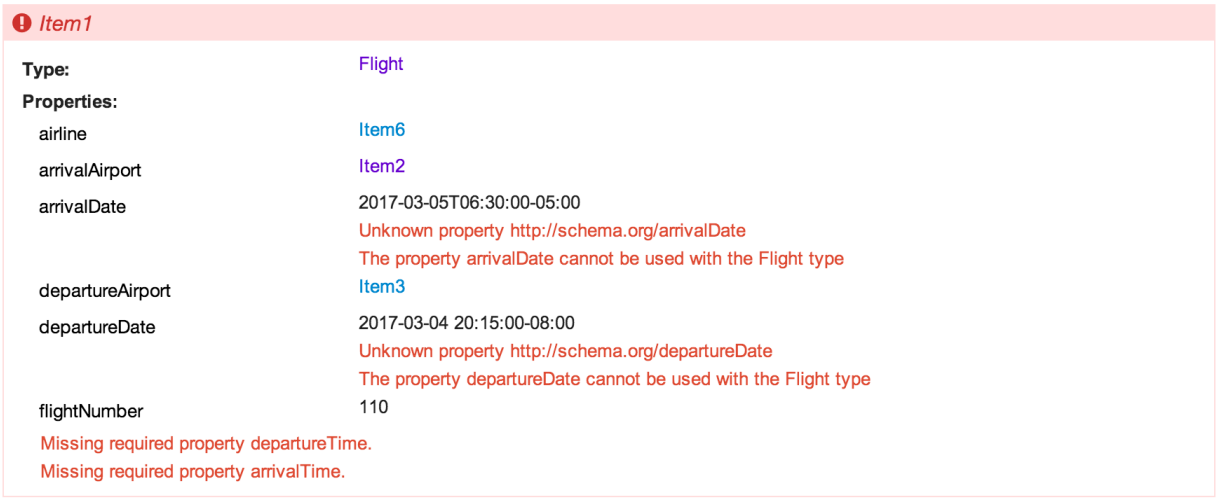

Figure 1: Sample results from our validator tool showing error messages for draft schema.org types.

To help illustrate the simplicity of our approach it is important to understand how constraints are represented and interpresent by the validation tool. A schema constraint in our system consists of the following elements:

- A `context` query which may be either a property path or a SPARQL SELECT query with a ?context variable.
- A `constraint` query which may be either a property path or a SPARQL ASK query that uses the ?context variable.
- An `expectedValue` for the constraint.
- An error `message` to be shown if the constraint fails.
- A `severity` level for this constraint (one of [critical, error, warning, info])

All of these properties are optional except for the constraint query. This is possible because each property has default behavior defined for it. If a context query is missing it can be denoted in the constraint query using the ?context variable. The expected value is “true” for all ASK queries (including property paths which can be expanded to path queries). If no error message is specified for a property path constraint, our tool will automatically generate one. The severity is “error” by default. This set of default values is what allows us to represent most of the required properties of our application as simply as:

```
{
  "@context": {...},
  "@id": "schema:FlightReservation",
  "constraints": [{
    "constraint": "schema:reservationFor"
  },{
    "constraint": "schema:reservationFor/schema:flightNumber"
  }]
```

```
}
```

This constraint (written in JSON-LD) simply states that any item with the FlightReservation type must have a value for the *reservationFor* property and that that item must have a *flightNumber*.

However, the default values can easily be overridden to allow a great deal of expressivity. For example, the following constraint provides a warning that boarding passes will only be shown in Google Now for flights which occur at a future date.

```
{
  "@context": {...},
  "@id": "schema:FlightReservation",
  "constraints": [{
    "context": "schema:reservationFor",
    "constraint": "ASK WHERE {?s schema:boardingTime ?t.
                   FILTER(?t > NOW())}",
    "severity": "warning",
    "message": "A future date is required to show a boarding pass.",
  }]
}
```

By providing a set of default behaviors and a mechanism for overriding those defaults with more expressive queries and error messages we can have the benefits of simple, easily readable paths for most constraints while still maintaining the flexibility to describe the few complex constraints of our application.

Since our property paths are essentially simplified to SPARQL queries (especially with the addition of property paths in SPARQL 1.1), why not just represent all of the constraints as SPARQL queries? First, there's the matter of attaching messages, severity levels and other metadata to the constraints. This can be done by binding the metadata to the context node and returning the metadata as part of a SPARQL SELECT query but the queries become overly verbose and prone to errors. By contrast the abbreviated path syntax used by our system is much more concise and doesn't require extensive knowledge of SPARQL. Secondly, its unclear what is the best way to distribute a collection of SPARQL queries. Since our validator reads in constraints as JSON-LD it makes it very easy to parse those same constraints in any programming language (languages like PHP, Python and Ruby ship with JSON parsers built in).

Perhaps the closest comparison to our approach is the SPIN Modeling Vocabulary. SPIN has a well-defined schema for modeling constraints as RDF and uses SPARQL to allow you to model very complex constraints. However, SPIN is not simply a constraint language; it has many other features including the ability to infer property values using rules defined as SPARQL queries. Adopting the SPIN vocabulary while only supporting a subset of its features and adding our own vocabulary for path queries and severity is confusing to developers. If our tool understands SPIN rules you would expect it to be able to do inference. Likewise, you might expect to be able to use

our `severity` or `expectedValue` properties in other SPIN tools but those are not part of SPIN.

A final alternative that we considered was to use OWL[14] as a way to validate schema constraints. The reason why chose not to use it was that the need for an OWL reasoner made our tool more difficult to implement than one which simply relied on SPARQL queries as well as the difficulty in expressing simple property path constraints which make up the majority of the constraints in our system. Others have explored OWL-based closed world reasoning, e.g. Clark & Parsia's Pellet approach [15].

W3C Workshops allow the Web standards community to reflect upon possible new standards work. What, if anything, might be needed in this area? On the one hand, there is evidence going back to the origins of RDF and XML that there are several very different notions of validation in play. Historically RDF schemas (and the more ambitious OWL specifications) have essentially encoded generalizations about the world: that a Person might have a biologicalParent, that a TouristAttraction can be considered a special kind of Place, or that a CreativeWork might have an 'author' which could be a Person or Organization. By contrast, a classic XML schema defines document types. Our exploration of application-specific constraints for RDF echoes several previous investigations, and reflect a common experience from practical deployments of RDF.

It is easy to assume that vocabulary design for RDF begins and ends with schema/ontology definitions. In practice, RDFS/OWL are not sufficiently constraining. Just as a human-oriented dictionary doesn't tell you what to say, no RDFS/OWL definition can say anything about which terms ought to be used in any specific RDF description. This problem was also recognised in the Dublin Core community several years ago. Dublin Core users recognised that many kinds of description would naturally combine multiple independently defined RDF vocabularies. However no W3C framework existed for subsequently documenting these descriptive patterns. DC's work on Application Profiles, and on Description Set Profiles (http://dublincore.org/documents/profile-guidelines/ http://dublincore.org/documents/dc-dsp/) explores ways in which the structure and constraints used in such descriptions could be documented in a machine-readable way.

Looking to W3C for possible standards in this area, a natural question is whether constraints ought to be expressed in terms of concrete RDF graph structures (for which SPARQL 1.1 seems well suited), or in terms of the entities in the world described by those graphs. For example, consider a graph with a node 'uri-1' of rdf:type 'Person', and two biologicalParent arcs, pointing to two other nodes 'uri-2', 'uri-3' each also with rdf:type edges pointing to 'Person'. If our application constraint is that we want our graph to contain certain information about each of a person's two 'biological parents', it is worth noting that this graph is entirely consistent with a situation in which both uri-2 and uri-3 represent the same real-world person. W3C's Semantic Web, RDF and Linked Data efforts present us with a rich, complex and sometimes daunting range of relevant technologies. OWL (which itself comes in several flavours and versions) is certainly relevant: it could help us understand which properties of uri-2 and uri-3 (eg. date of

birth) imply that they can't (or can) stand for the same entity.

Unfortunately, many real world data checking problems are not expressed in a suitably clean and tidy form for OWL reasoning. Even this biologicalParent scenario is in tension with recent scientific developments (see http://www.bbc.co.uk/news/health-23079276). Solutions expressed solely in terms of concrete RDF graphs can seem more practical and worldly since constraints are expressed on a case by case basis rather than as general observations, but we suggest more experience is needed before launching a Recommendation track standards initiative. The W3C community has already invested many hours of teleconference time in the creation and maintenance of RIF, OWL, RDF and SPARQL. It is natural for advocates of each of these technologies to expect to see them re-used in validation-oriented constraint languages. It might be that W3C's most important role for now could be to bring together implementation experience from each of these approaches, and to capture requirements and deployment scenarios rather than directly launching a standards-track initiative. This Workshop is a good first step, but if the next step is a standardization group there is a significant risk of creating a large and unwieldy technology. We would welcome a group (whether WG or community group) that investigates this area more thoroughly, and had as its output very specific recommendations towards chartering REC-track work within W3C.

References

1. Schema Validator Tool — https://developers.google.com/gmail/schemas/testing-your-schema
2. Schema.org — http://schema.org
3. Microdata — http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html
4. RDFa — http://www.w3.org/TR/rdfa-syntax/
5. JSON-LD — http://json-ld.org/
6. Structured data in Google Search, Google Now and GMail.
7. SPARQL — http://www.w3.org/TR/sparql11-query/
8. Property Paths — http://www.w3.org/TR/sparql11-query/#propertypaths
9. RDF — http://www.w3.org/RDF/
10. Schematron — http://www.schematron.com/
11. XML Path Language — http://www.w3.org/TR/xpath/
12. Schemarama — http://swordfish.rdfweb.org/discovery/2001/01/schemarama/
13. SPIN Modeling Vocabulary — http://www.w3.org/Submission/spin-modeling/
14. OWL — http://www.w3.org/TR/owl-ref/
15. Pellet Integrity Constraints: Validating RDF with OWL — http://clarkparsia.com/pellet/icv/