From XML Schema to JSON Schema: Translation with CHR

Falco Nogatz, Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
{falco.nogatz,thom.fruehwirth}@uni-ulm.de

Abstract. Despite its rising popularity as data format especially for web services, the software ecosystem around the JavaScript Object Notation (JSON) is not as widely distributed as that of XML. For both data formats there exist schema languages to specify the structure of instance documents, but there is currently no opportunity to translate already existing XML Schema documents into equivalent JSON Schemas.
In this paper we introduce an implementation of a language translator. It takes an XML Schema and creates its equivalent JSON Schema document. Our approach is based on Prolog and CHR. By unfolding the XML Schema document into CHR constraints, it is possible to specify the concrete translation rules in a declarative way.

Keywords: Constraint Handling Rules, Language Translator, XML Schema, XSD, JSON Schema

1 Introduction

XML, the Extensible Markup Language [1], is today one of the most used formats to save and exchange structured data. Being a recommendation of the World Wide Web Consortium (W3C) since 1998, a large software ecosystem has been evolved, including data formats to specify the schema of XML documents. One of them is the XML Schema Definition (XSD) [2].

Since its proposal in 2006, there is an alternative data format especially used in web services: JSON, the JavaScript Object Notation. Its formal language to specify the format of a JSON document, called JSON Schema, is still in draft status [3]. Although there are validation tools implementing the IETF draft, the number of JSON Schemas used in practice is still moderate. One of the reasons is that there is currently no mechanism to translate an already existing XML Schema into equivalent JSON Schema.

As an application of XML, XSD documents are valid XML instances. Although JSON Schema is JSON-based as well, the naive approach of using an already existing XML to JSON translator as published by [6] would not result in a valid JSON Schema document. To satisfy the Core Meta-Schema [5], the demanded translator has to provide some additional logic, extending the general problems of translating XML to JSON instances as presented in [4].

In this paper, we propose an approach for an XSD to JSON Schema language translator based on Prolog and Constraint Handling Rules (CHR) [8]. The translator unfolds a given XML Schema into CHR constraints. By creating a CHR constraint for every XSD node it is possible to specify the concrete translation rules of common XML Schema fragments in a declarative way in form of CHR rules.

The paper is organized as follows. In Section 2, we will give an example to illustrate the problem and we will determine the considered versions of the XSD and JSON Schema specifications. The paper continues by presenting the introduced CHR constraints. In Section 3 the overall translation process is presented. Finally, the paper ends with concluding remarks in Section 4.

2 Preliminaries

The aim of this work is to create a Prolog/CHR module that offers a predicate `xsd2json(XSD,JSON)` which holds the equivalent JSON Schema as `JSON` for a given `XSD` instance. Before getting into the concrete translation process we want to introduce the used techniques and specify the scope of this tool. In what follows we explain the problem instance by giving an example of a simple XSD and its expected translated JSON Schema equivalent.

2.1 Problem Definition

Following the formal description of the XML Schema language [7], an XML Schema consists of four components: elements (`xs:element` nodes), simple types (`xs:simpleType` nodes), complex types (`xs:complexType` nodes) and attributes (`xs:attribute` nodes). Because the also introduced attribute groups and model groups are only placeholders in complex type definitions, we will omit those components for our translator. In Section 3.4, we will introduce translation rules for each of the four given components, depending on their structure and values.

Although the XML Schema 1.1 Specification has been the official W3C recommendation since April 2012, we restrict ourselves to the XML Schema 1.0 Specification. The more up-to-date specification primarily introduces conditional types and assertions based on XPath expressions. Since there is currently no XPath equivalent for JSON, it would not be possible to translate those new XPath-based elements at all.

For the target language JSON Schema we refer to the latest version of the specification, Draft 04 [3], which is already supported by a number of JSON validators in multiple languages. A list of current implementations can be found in [10].

2.2 Problem Instance Example

As a motivating example, we will consider a small XML document, as shown in Figure 1, and its related XSD, as specified in Figure 3.

```
<?xml version="1.0" ?>
<percentages>
  <value>99</value>
  <value>42</value>
  <value>0</value>
</percentages>
```

Fig. 1. Example XML

```
{
  "value": [ 99, 42, 0 ]
}
```

Fig. 2. JSON document, valid against the JSON Schema of Figure 4

The aim of the language translator is to create an equivalent JSON Schema of the XSD given in Figure 3. It should respect the following the semantics:

- There is a list of values.
- The list contains at most five values.
- Every value must be a nonnegative integer.

Following the XSD specification in [7], there is additional information implicitly given: By omitting the `minOccurs` attribute in an `xs:element` within an `xs:sequence` its default value `1` is used, so the list has to contain at least one value.

The equivalent JSON Schema that ensures these constraints is shown in Figure 4 and its corresponding JSON document in Figure 2. The `percentages` node of the XML document has no equivalent in the JSON Schema instance. This is caused by the circumstance that the `percentages` element adds no constraints and therefore might only be used to create a valid XML document, which requires a single root element. The language translator uses such assumptions to create a simple, but appropriate JSON Schema.

2.3 CHR Constraints

To provide translation rules for concrete XSD fragments, we use a combination of the logic programming languages Prolog and CHR [8][14]. This enables us to specify the translation rules in a declarative way. Since for each XSD node a new CHR constraint will be generated, it is possible to create CHR rules referencing constraints by their characteristics without having to implement the tree traversal of the XSD document.

We use CHR with Prolog as its host language. The suggested implementation can be found online at https://github.com/fnogatz/xsd2json and has been tested with the CHR library for SWI-Prolog [12]. To hold the information of a given XSD term we introduce the following CHR constraints:

- `node(Namespace,Name,ID,Children_IDs,Parent_ID)`
  For each XML node in the XSD document a new `node/5` constraint is generated, holding its namespace and tag name. To obtain a reference, a unique identifier is added as well as the list of its parent's and children's identifiers.
- `node_attribute(ID,Key,Value,Source)`
  For each XSD attribute a new `node_attribute/4` constraint is propagated,

holding its name as `Key`, its `Value` and the identifier of the related `node/5` constraint. The `Source` is `source` for explicitly set and `default` for inherited attributes. For example `maxOccurs="5"` of the innermost `xs:element` of Figure 3 is mapped to a constraint `node_attribute(_ID,maxOccurs,5,source)`.

– `text_node(ID,Text,Parent_ID)`
  If an element's child is simply a text and no nested XML node, a `text_node/3` constraint is generated. It gets a unique identifier like a regular child node and holds the text as well as the identifier of its parent element.

All translated fragments are stored in `json(ID,JSON)` constraints, holding the `JSON` Schema of the XSD node with the identifier `ID`. Because the entire JSON Schema is built step by step, the innermost fragments of the XSD propagate the first `json/2` constraints. These will be picked up for the translation of their parent elements, resulting in a JSON Schema for the entire XSD.

```
<?xml version="1.0" ?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="percentages">
    <xs:complexType>
      <xs:sequence>
        <xs:element
          name="value"
          maxOccurs="5"
          type="xs:nonNegativeInteger" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Fig. 3. Possible XSD for XML of Figure 1

```
{
  "type": "object",
  "properties": {
    "value": {
      "type": "array",
      "items": {
        "type": "integer",
        "minimum": 0,
        "exclusiveMinimum": false
      },
      "minItems": 1,
      "maxItems": 5
    }
  },
  "required": [ "value" ]
}
```

Fig. 4. Tanslated JSON Schema, based on the XSD of Figure 3

3 Translation Process

The overall translation process can be split into six subtasks as illustrated in Figure 5. The different steps can be distinguished by their function as well as by the used programming language.

In the following we will present the various steps. The main part of the translator, the translation rules of XSD fragments, is introduced in Section 3.4.

3.1 Read in XML Schema into Prolog

SWI-Prolog provides a wide support for working with XML documents. By use of its SGML/XML parser [11], an XSD document can be read in as a nested Prolog term. Figure 6 shows the term generated by the built-in `load_structure/3` predicate [12] for the XSD of Figure 3.

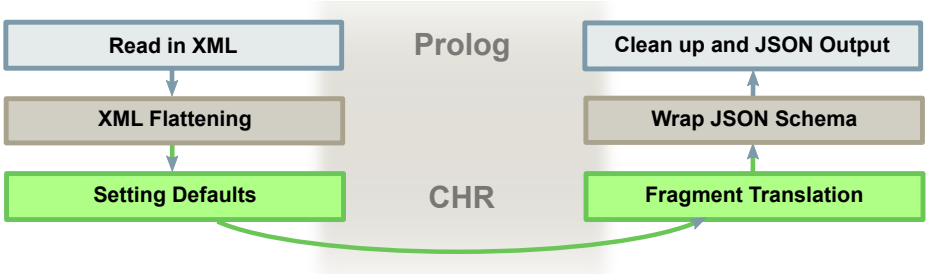

Fig. 5. Steps of the overall translation process

```
[ element(
    'http://www.w3.org/2001/XMLSchema':schema,        % namespace and name
    [ xmlns:xs='http://www.w3.org/2001/XMLSchema' ],  % attributes
    [ element(                                        % nested elements
        'http://www.w3.org/2001/XMLSchema':element,
        [ name=percentages ],                         % attributes
        [ ... ])                                % the other nested elements
    ])]
```

Fig. 6. Nested Prolog term of the XSD document of Figure 3

3.2 XML Flattening

This nested Prolog term can be traversed recursively to propagate the related `node/5`, `node_attribute/4` and `text_node/3` constraints. Their positions are retained by their unique identifiers and references to parent and child nodes.

3.3 Setting Defaults

Because Prolog's XML parser will only read in explicitly set attributes, we have to add the default attributes as shown in Section 2.2. The translation rules used in the next step refer to attributes like `minOccurs` and `maxOccurs`, which can be omitted. To ensure these optional attributes are always present, we propagate a `node_attribute/4` with the `Source` set to `default`, as mentioned in Section 2.3. If there is an identical `node_attribute/4` constraint with its last component set to `source`, the default one is removed by a CHR simpagation rule.

3.4 Fragment Translation

Before examining the most important step, we will have to a look at the intended result of the overall translation process: a Prolog representation of JSON Schema. Like for XML, SWI-Prolog comes with a library to serialize JSON. With the `http/json` library [12] a JSON object is represented by `json(L)`, in which `L` is a list of the form `[Key1=Value1,Key2=Value2,...]`. JSON arrays are represented by Prolog lists.

Referring to the constraints propagated in the steps before, we can translate XSD fragments to their equivalent JSON Schema. All translation rules follow the same form: They propagate a single `json/2` constraint that holds the JSON Schema of this XSD fragment; the guard ensures that all `node/5` and `node_attribute/4` constraints are of the XML Schema namespace. The rule's head contains the following parts:

- The `node/5` constraint for which the `json/2` representation is generated. In most cases its children's `node/5` constraints are referenced by using the `Parent_ID` component.
- Some `node_attribute/4` and `text_node/3` constraints, depending on the XSD fragment that has to be translated.
- Some `json/2` constraints to merge already translated fragments. This way it is possible to generate the JSON Schema translation of an XSD node by combining the translations of its child nodes.

Hereby the propagation rule to generate the JSON Schema for the innermost `xs:element` of Figure 3 is:

```
node(NS,element,ID,_C,_Parent), node_attribute(ID,type,With_NS,_)
  ==> xsd_namespace(NS), valid_xsd_type(With_NS,Type)
  |   convert_xsd_type(Type,JSON), json(ID,JSON).
```

This rule applies if the `xs:element` node has an XSD namespace and is of a primitive XSD data type. XML Schema provides various predefined data types. Although the number of data types defined for JSON is limited, it is possible to restrict them similarly to constraining facets [9] in XSD. Therefore we can define a `convert_xsd_type/2` predicate by providing JSON's equivalents of all predefined XML data types like in Table 1 in excerpts.

Table 1: Translation of simple XSD data types (extract of [13])

XSD primitive type	JSON Schema type definition
`xs:string`	`{ "type": "string" }`
`xs:float, xs:double, xs:decimal`	`{ "type": "number" }`

`xs:nonNegativeInteger`	`{ "type": "integer",`
	`"minimum": 0,`
	`"exclusiveMinimum": false }`

XSD's primitive data types can be restricted by constraining facets [9], for example to specify all possible values for a string. These facets can be translated by using a similar table of equivalents, which are collected in [13].

The primitive data types and constraining facets apply only to `xs:attribute` and certain `xs:element` nodes. However, an XSD is more than the definition of simple types: via `xs:complexType` nodes attributes of elements can be specified as well as their child nodes. The occurence of `xs:element` within an `xs:sequence` as shown in Figure 3 is a common structure in XSD documents. Therefore we

have to translate nested XSD nodes as the last part of this step. The general approach has already been introduced before: depending on specific `node/5`, `node_attribute/4` and `text_node/3` constraints and sometimes already translated fragments given as `json/2` constraints, we compose the translation of an XSD node.

As an example we introduce the actual translation rule for the nested `xs:sequence`/`xs:element` structure of the example in Figure 3:

```
node(NS1,sequence,Sequence_ID,_SC,_SP), node(NS2,element,El_ID,_EC,Sequence_ID),
    json(El_ID,Element_JSON), node_attribute(El_ID,name,Element_Name,_)
    node_attribute(El_ID,minOccurs,MinOccurs,_O),      % _O to match both origins
    node_attribute(El_ID,maxOccurs,MaxOccurs,_O),      %   'default' and 'source'
  ==>
    xsd_namespace(NS1), xsd_namespace(NS2)             % valid XSD namespaces?
  | JSON = [ type=object,                              % build JSON object
             properties=json([
               Element_Name=json([ type=array,
                                   items=Element_JSON,
                                   minItems=MinOccurs,
                                   maxItems=MaxOccurs
               ]) ]) ],
    (MinOccurs_Number >= 1, Full_JSON = [required=[Element_Name]|JSON];
      MinOccurs_Number < 1, Full_JSON = JSON),         % required property?
    json(Sequence_ID,json(Full_JSON)).                 % propagate JSON
```

Many applications of nested structures have been identified, documented in [13] and implemented by similar propagation rules. Because a `node/5` constraint might apply to multiple CHR rules, there can be various `json/2` constraints with the same identifier. These are merged by a simpagation rule with the help of a self-defined `merge_json` Prolog predicate.

3.5 Wrap JSON Schema

The previous step terminates as soon as the root element of the given XSD has been translated and its `json/2` constraints have been merged. In addition the globally defined type definitions are merged into the `definitions` object of the root's `json/2` constraint.

3.6 Clean-up and JSON Output

Finally, the created JSON Schema object is cleaned up: in the creation process, the names of XML attributes (specified as `xs:attribute` in the XSD) were prefixed with an `@` symbol. If there is no `xs:element` in this `xs:complexType` with the same name, the attribute's `@`-prefix is removed.

4 Conclusion

In this work, a language translator to convert XML Schema to an equivalent JSON Schema was implemented. The entire implementation is available online at

https://github.com/fnogatz/xsd2json, its detailed concept can be found in [13]. As the `xsd2json` Prolog/CHR module was developed in a bottom-up approach, it also provides a test framework and a large number of test cases.

The `xsd2json` module is still under development to be applicable for all XML Schema instances. Due to the lack of various features in JSON Schema it might not be possible to support all constraining semantics. For example, there is currently no XPath-like way to address a specific property inside a nested JSON document. Therefore, the XSD elements `xs:key`, `xs:keyref` and `xs:unique` cannot be supported as well as the new features like `xs:assertion` introduced by the most up-to-date XML Schema 1.1 Specification.

Another missing feature is the handling of referenced XSD documents. While the current implementation respects multiple namespaces, it translates only a single file. Therefore `xs:import` and `xs:include` are not supported.

References

1. Tim Bray and Jean Paoli and C Michael Sperberg-McQueen and Eve Maler and François Yergeau: Extensible markup language (XML). World Wide Web Journal volume 2, number 2, pages 27–66 (1997)
2. XML Schema, Structures Part. World Wide Web Consortium (W3C), Recommendation October 2004), http://www.w3.org/TR/xmlschema-1 (2004)
3. Kris Zyp: A JSON Media Type for Describing the Structure and Meaning of JSON Documents (Draft 04). IETF Internet-Draft, http://tools.ietf.org/html/draft-zyp-json-schema-04 (2013)
4. David Lee: JXON: an architecture for schema and annotation driven JSON/XML bidirectional transformations. Balisage: The Markup Conference, Balisage Series on Markup Technologies, volume 7 (2011)
5. JSON Schema and Hyperschema: Core/Validation Meta-Schema. http://json-schema.org/schema (2013)
6. Senthil Nathan and Edward J Pring and John Morar: Convert XML to JSON in PHP. http://www.ibm.com/developerworks/xml/library/x-xml2jsonphp/ (2007)
7. Allen Brown and Matthew Fuchs and Jonathan Robie and Philip Wadler: XML Schema: Formal Description. W3C Working Draft, volume 25, pages 1–25 (2001)
8. Thom Frühwirth: Constraint Handling Rules. Cambridge University Press (2009)
9. Biron, Paul, Ashok Malhotra, and World Wide Web Consortium: XML schema part 2: Datatypes. World Wide Web Consortium Recommendation REC-xmlschema-2-20041028 (2004)
10. JSON Schema and Hyperschema: JSON Schema Software. http://json-schema.org/implementations.html (2013)
11. Jan Wielemaker: SWI-Prolog SGML/XML Parser. SWI, University of Amsterdam, Roetersstraat, 15. Jg., p. 1018 (2005)
12. Thom Frühwirth and Jan Wielemaker and Leslie De Koninck and Markus Triska: SWI Prolog Reference Manual 6.2.2. Books on Demand (2012)
13. Falco Nogatz: From XML Schema to JSON Schema – Comparison and Translation with Constraint Handling Rules. (2013)
14. Thom Frühwirth and Frank Raiser: Constraint Handling Rules: Compilation, Execution, and Analysis. Books on Demand (2011)