ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Software Engineering and Compiler Construction

From XML Schema to JSON Schema - Comparison and Translation with Constraint Handling Rules

Bachelor Thesis at the University of Ulm

Submitted by:
Falco Nogatz
falco.nogatz@uni-ulm.de

Reviewer:
Prof. Dr. Thom Frühwirth

2013

Version December 10, 2013

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

This thesis identifies similar semantics in the two schema definition languages XML Schema and JSON Schema to build a dictionary which covers typical use cases to automatically transform first to the latter. As the range of functions of both XML Schema and JSON Schema are not identical, concrete transformation rules to reproduce similar behavior of data constraints are discussed and implemented by use of the logic programming language Constraint Handling Rules. As a result, a Prolog library `xsd2json` is created which provides tools to translate complex XML Schema documents into their equivalent JSON Schema documents.

Contents

1. Introduction **1**

1.1. Motivation 1

1.2. Scope of this Thesis 3

1.3. Methodology 3

1.4. Road Map 3

2. Technologies **5**

2.1. XML 5

2.2. XML Schema 8

2.3. JSON 10

2.4. JSON Schema 12

2.5. Prolog 13

2.6. Constraint Handling Rules 14

3. General Translation Process **17**

3.1. Read in XML Schema into Prolog 19

3.2. XML Flattening 23

3.3. Setting Defaults 25

3.4. Fragment Translation 26

3.5. Wrap JSON Schema 31

3.6. Clean up and JSON Output 32

4. Translation Rules **35**

4.1. XSD Primitive Types 36

4.2. Constraining Facets 38
4.3. Translation of nested XSD Elements 42

5. Evaluation **53**
5.1. Test Framework 53
5.2. Current Limitations 54

6. Conclusion **55**
6.1. Summary 55
6.2. Outlook 56

A. User Manuals **59**
A.1. xsd2json 59
A.1.1. Installation 59
A.1.2. Usage 60
A.2. Test Framework 60
A.2.1. Installation 61
A.2.2. Provided Tests 61
A.2.3. Pretty TAP output 62
A.2.4. Transform a single XSD File 62

B. Source Codes of Prolog Predicates **63**
B.1. `xsd_flatten_attributes/2` 63
B.2. `xsd_flatten_nodes/4` 64
B.3. `xsd_namespace/1` 65
B.4. `xsd_namespaces/1` 65
B.5. `lookup/4` 66
B.6. `merge_json/4` 66
B.7. `merge_json/3` 69
B.8. `remove_at_from_property_names/2` 69
B.9. `is_required_property/2` 75

C. Source Codes of CHR Rules **77**

C.1. Translation of `xs:attribute` 77

List of Tables **85**

Bibliography **87**

1 Introduction

1.1. Motivation

XML, the Extensible Markup Language, is one of the most used format to save and exchange structured data, especially in web services. There exist applications for nearly every use case scenario [Cov05], covering the data formats to exchange calendar information as well as mathematical expressions. Because of its wide distribution, a very large ecosystem has been evolved: Beside a big number of tools to display and manipulate XML files, access partial data via query languages like XPath, data formats to specify the schema of XML documents has been established. One of them is the *XML Schema Definition* (XSD), which is more expressive than the traditional Document Type Definition (DTD). XML Schema allows to define the general format of an XML document, that means which elements are allowed, the number and order of their occurrences, of which data type they should be, etc.

But since its proposal in 2006 there is a new player on the field of data formats: Data serialized in the *JavaScript Object Notation* (JSON) is far smaller than its XML counterpart and its instances are valid JavaScript objects, which made this interchange format especially interesting for web developers as they no longer need a separate conversion step when loading information in asynchronous web applications via AJAX (Asynchronous JavaScript and XML). By now most popular web services use JSON for their application programming interfaces instead of XML. While the pure JSON format has been established quite fast, the ecosystem is not yet comparable to XML: The first draft for a schema language to "define validation, documentation, hyperlink navigation, and interaction control of JSON data" [KZ09] is from 2009 and in 2013 still a draft [KZ13], even though its author Kris Zyp mentioned that the current version is ready to use and a potential fifth draft would be only a clean up.

Besides its draft status one of the reasons that JSON Schema is not established like XSD is that there is currently no easy way to transform already defined XML Schemas into equivalent JSON Schemas. There already exist some XML to JSON converter, that are able to translate an XML document into its JSON counterpart. By using such like XML to JSON converter it might be possible to translate a single XML document into its JSON equivalent, but this will not be an option for the Schema instances: One can not simply convert a given XML Schema document with the hope, that the resulting JSON document will be a valid JSON Schema document as well, because the semantics of both metalanguages are different. This is why there is a need for a dedicated XML Schema to JSON Schema converter.

By the use of Prolog and Constraint Handling Rules as its primary programming language, it is natural to solve this problem in a declarative manner: We want to describe several use cases like "every `xs:element` within a `xs:sequence` node should be translated by this JSON Schema fragment" by simply referencing the addressed XML Schema nodes without having to worry about the actual tree traversal process. Using the rule-based approach we can establish concrete translations of common structures within an XML Schema document.

1.2. Scope of this Thesis

Both description languages, XML Schema and JSON Schema, do not provide the same range of functions to characterize the constraints of its instances. In XML documents, information can be stored either as the text entity of an XML node or as one of its attributes, while JSON objects are simply key-value pairs. Therefore it is easier to find a JSON Schema equivalent to an XML Schema than vice versa.

We will investigate common XML Schema use cases and provide equivalent JSON Schema instances. Beginning with simple data types and their restrictions we provide translation rules for complex XML types with respect to their contexts. We will use the logic programming languages Prolog and Constraint Handling Rules (CHR) to directly implement the deduced translation rules. With the use of declarative programming languages we can concentrate on the translation process instead of the implementation of the tree traversal.

1.3. Methodology

This thesis provides a broad set of translation rules for common XML Schema snippets to their equivalent JSON Schema instances. While discussing them by a theoretical point of view, a prototype for a converter of XSD instances into valid JSON Schema instances is one of its main results. At the end, we will evaluate the elaborated translation rules and the prototype and discuss what further technologies would help to establish a complete translation.

1.4. Road Map

As both examined description languages, XML Schema and JSON Schema, are for them self valid XML respectively JSON instances, we start with a brief introduction into those data formats followed by their Schema definition languages. At the end of chapter

2 we introduce the used declarative programming language CHR and its host language Prolog.

Next in chapter 3, we elaborate the general transformation process by dividing it into several steps: Beginning with the read in of an XML document into Prolog, we create CHR constraints for each XML Schema node and attribute. Those are used to translate XML Schema fragments into their equivalent JSON Schema. The whole process ends ups with the collection of all JSON Schema fragments and building of a single JSON Schema document.

The concrete translation rules of the so called XML Schema fragments are introduced in chapter 4. Beginning with XML Schema's primitive data types, followed by derived types, up to combined elements we elaborate JSON Schema equivalents for typical XML Schema use cases.

At the end the test framework is presented which is an important part of the test-driven development approach of `xsd2json`. We discuss current limitations of the implementation and give an outlook on future improvements.

2

Technologies

This chapter introduces the examined description languages, the XML Schema Definition language and its JSON counterpart JSON Schema. At the end we introduce Prolog and Constraint Handling Rules which are used to describe the translation rules and for the implementation of the converter prototype.

2.1. XML

The Extensible Markup Language, for short XML, is a markup language. It is both human-readable and machine-readable and therefore used as a interoperable data format. It is a subset of the Standard Generalized Markup Language (SGML) and standardized by the World Wide Web Consortium (W3C). The latest version is the XML 1.1 (Second Edition) Specification [bra06].

Listing 2.1: Example XML document for a product catalog

```
<?xml version="1.0"?>
<catalog>
  <item id="4711">
    <name>Unit 1</name>
    <price currency="Euro">19.99</price>
    <price currency="Dollar">27</price>
  </item>
  <item id="4712">
    <name>Unit 2</name>
    <price currency="Euro">12.50</price>
  </item>
</catalog>
```

An XML document consists of a number of elements and their attributes. A simple example XML document is shown in Listing 2.1. To describe its components we use the following terminology:

- *Tag*
  A tag is a name bordered by a smaller-than and greater-than character, i.e. the tag with the name `price` is written as `<price>`. We distinguish three types of tags:
  - *Opening tag*, e.g. `<price>`
    The opening tag marks the beginning of the content of an XML element.
  - *Closing tag*, e.g. `</price>`
    The closing tag marks the end of the content of an XML element.
  - *Empty tag*, e.g. `<price/>`
    The empty tag is the short variant for a starting tag followed immediately by its closing tag, e.g. `<price></price>`. As its name implies, it neither has a child element nor a textual content.
- *Element*
  An element is either an empty tag or the combination of an opening tag and its

closing tag. Its *content* is all between the opening and the closing tag and might be again some XML elements or only text or a mix of both.

- *Parent element, child element, siblings*
  We refer to the element in whose content another element appears with an opening and/or closing tag, as its parent element. On the other hand the *child elements* of an element are all elements whose opening and/or closing tag in the very first level of the element's content.
  Siblings are all element with the same parent element.

- *Attribute*
  Each element can have a number of attributes which can be specified in its opening or empty tag as key-value pairs, e.g. `<price currency="Euro" />`. The attribute's value can be enclosed in either single or double quotes.

An XML document is *well formed*, if the following constraints are satisfied:

1. A single root element
   Every XML document must have a single top-level element which is called the *root element*. It contains all other elements or textual context or could be empty. Because of the fact, that comments (`<!-- ... -->`) and preprocessing instructions (`<? ... ?>`) are in fact no XML elements, one could use them additionally on the document's top level. The root element has neither a parent element nor siblings.

2. Closing tag for each opening tag
   Because the so called *Omittag* feature of SGML is set to `No` in the XML declaration [Cla97], neither opening tag nor closing tags can be optional, in other words for each opening tag there must be a closing tag.

3. No overlapping elements
   Every element must have a single parent element, in other words its opening and closing tag must be in the content of the same element.

4. No smaller-than character in texts
   The smaller-than character < is reserved as the starting symbol of tags and must therefore not occur in an element's textual content.

In the following we will suppose to handle only well formed XML documents.

2.2. XML Schema

While the characteristic of being well formed only means that a given XML document is syntactically correct, one often aims for an additional property: it should be also valid against some specification and keep some conditions. This is where XML Schema comes into play.

XML Schema (XSD for XML Schema Definition, or formerly WXS for W3C XML Schema) provides a mechanism to describe the general structure of an XML instance, that means the occurring elements, their attributes and how they can be nested. It is standardized by the W3C, the latest version is the XML Schema 1.1 Specification ([GSMT+08] and [PGM+08]). Although the XSD 1.1 Specification is the official W3C Recommendation since April 2012 [Arc12], this thesis uses the XML Schema 1.0 (Fifth Edition) Specification [Bir04] as its basis for the translated XML Schema instances. The newer specification introduces conditional types based on an XPath expression and the new XSD elements `<xs:assert>` and `<xs:assertion>` which are used to specify XPath constraints. As there is as of yet no standardized XPath equivalent for JSON instances those constraints could not be translated by a converter into equivalent JSON Schema.

In an XML Schema the general structure of an XML document is specified as well as constraints for the contained entities. The following components of an XML document are described [BFRW01] by the schema:

- *Element*
  Every element used in the XML document is defined by an element declaration which includes the element's name, namespace and type. The element's name-

space do not need to be specified explicitly but can be derived by its parent element. The element is either of a simple or complex type.

- *Attribute*
  Every attribute used within an XML element is defined by an attribute declaration. The attribute has a (derived) target namespace and is always of a simple type. Additionally the fixed or default value can be declared.
- *Simple Type*
  There are two general types in XML Schema: simple and complex types. Simple type (also called data types) instances are single values, i.e. in general strings or numbers. With the use of restrictions it is possible to specify their format or possible values.
- *Complex Type*
  A complex type describes the content of an element, that means which child elements are allowed, in which order and amount. It also specifies the element's attributes. Complex types can be restricted and extended.

By using the XML Schema metalanguage we can characterize these elements, attributes and types of an XML document by an XSD document, which is an XML document for itself. The concrete XML Schema syntax to specify the document's structure and its type are introduced in chapters 3 and 4, where an equivalent JSON Schema translation for each is elaborated. Just to show the general layout of an XML Schema document we present in Listing 2.2 a possible schema for the XML document mentioned above:

Listing 2.2: Example XML Schema for 2.1

```
1 <?xml version="1.0" ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="catalog">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="item" type="item"
7           maxOccurs="unbounded" />
8       </xs:sequence>
```

```
9       </xs:complexType>
10    </xs:element>
11
12    <xs:complexType name="item">
13      <xs:sequence>
14        <xs:element name="name" type="xs:string" />
15        <xs:element name="price" type="price"
16          maxOccurs="unbounded" />
17      </xs:sequence>
18
19      <xs:attribute name="id" type="xs:nonNegativeInteger" />
20    </xs:complexType>
21
22    <xs:simpleType name="price">
23      <xs:restriction base="xs:decimal">
24        <xs:minInclusive value="0" />
25      </xs:restriction>
26    </xs:simpleType>
27  </xs:schema>
```

2.3. JSON

JSON is a data format designed to exchange human- and machine-readable information in a key-value format. It is formal specified in RFC 4627 [Cro06] and was introduced by Douglas Crockford in 2006. Since then it became popular especially in web services and is going to take over from XML as the first choice to communicate with application programmable interfaces [Duv11]. Being directly supported in JavaScript, it is perfectly suited for web-based architectures. Compared to XML, parsing JSON in modern browsers is faster and the network transmission time is lower [NPRI09].

Listing 2.3: Example JSON document for a product catalog

```
{
  "items": {
    "4711": {
      "name": "Unit 1",
      "prices": {
        "Euro": 19.99,
        "Dollar": 27
      }
    },
    "4712": {
      "name": "Unit 2",
      "prices": {
        "Euro": 12.50
      }
    }
  }
}
```

A JSON document consists of a number of key-value pairs (also called attribute-value pair), ordered lists and and simple values of a basis type [JSO13a]. An example JSON document is shown in Listing 2.3. We use the following terminology to describe its components:

- *Value*
  A value is either an array, an object, a string encapsulated in double quotes, for example `"foo"`, or any number or one of `true`, `false`, `null`.

- *Array*
  An array is an ordered collection of values, encapsulated in the brackets `[` and `]`. The values are separated by commas and do not need to be of the same type.

- *Object*
  An unordered collection of name-value pairs is called object and must be encapsulated in curly braces `{` and `}`. The *key* is a string encapsulated in double quotes and must be followed by a colon `:` and its value. The keys must be unique for each object.

While the JavaScript syntax (specified as ECMAScript in [ECM99]) defines more basis data types than boolean, null, string and number, the additional formats like Date, Function and Regular Expression are not part of the JSON specification and must therefore modelled as strings or numbers. We will see in the next section 2.4 that there are formats defined for common properties like dates, phone numbers etc.

2.4. JSON Schema

While the pure JSON format has been established over the recent years, the ecosystem is not similar to XML: The first draft for a schema language to “define validation, documentation, hyperlink navigation, and interaction control of JSON data” was created in 2009 [KZ09] and is in 2013 still a draft [KZ13], even though its author Kris Zyp mentioned that the current version is ready to use and a potential fifth draft would only be a clean up [JSO11]. The latest version of the specification, Draft 04, is supported by a number of validators in multiple programming languages. A list of current implementations can be found in [JSO13c].

A JSON Schema is defined in a JSON document for itself. A possible schema for the JSON product catalog mentioned in Listing 2.3 is shown in Listing 2.4. The semantic of the JSON Schema elements is specified in [Gal13].

Listing 2.4: Example JSON Schema for 2.3

```
1 {
2   "type": "object",
3   "properties": {
4     "items": {
```

```
5       "type": "object",
6       "additionalProperties": {
7         "type": "object",
8         "properties": {
9           "name": {
10            "type": "string"
11          },
12          "prices": {
13            "type": "object",
14            "additionalProperties": {
15              "type": "number",
16              "minimum": 0
17            }
18          }
19        }
20      }
21    }
22  }
23 }
```

2.5. Prolog

To specify the concrete translation rules of XML Schema snippets into an equivalent JSON Schema we use a combination of the logic programming languages Prolog and CHR (see 2.6). By using a declarative programming approach, we can concentrate on the translation rules itself instead of implementation details of the tree traversal.

Prolog was one of the first logic programming languages. There exist multiple implementations, the most popular are SICStus Prolog and SWI-Prolog. Following a declarative approach, a Prolog program consists of facts and rules. A rule consists of a head and body as shown in Listing 2.5.

Listing 2.5: Structure of a Prolog rule

```
1 Head :- Body.
```

The rule's head must be a single predicate whereas the body can be an conjunction (separated by a comma) or disjunction (separated by a semicolon) of any number of predicates. The predicates in the rule's body are called *goals*. The semantic of a rule with the structure mentioned in Listing 2.5 is "head is true if body is true" and is therefore a horn clause. If the body of a rule is empty or `true` like in Listing 2.6, we call it *fact*.

Listing 2.6: Example of a Prolog fact

```
1 foo(bar).
2 foo(bar2) :- true.
```

Once the user posts a goal (or it is called for example in the guard of a CHR rule), the Prolog engine tries to find a binding for the given free (also called unbounded) variables that satisfies the predicate with respect to the defined facts and rules. If all variables are bound, the Prolog engine searches for a satisfiable resolution. The given facts and rules are examined in their order in the source code. If the application of multiple rules is possible, backtracking is used to prove the given goal. [CM84] provides further information about the semantics of Prolog.

2.6. Constraint Handling Rules

Constraint Handling Rules (CHR) is a declarative programming language following the constraint-based programming paradigm. It was invented in 1991 by Thom Frühwirth and is usually used as an extension for a host language. Today there are multiple implementations, also in non-declarative programming languages like Java. The most popular host language is currently Prolog, therefore the CHR library is included in many Prolog implementations like SWI-Prolog and SICStus. The implemented prototype of an XSD to JSON Schema converter is tested with the CHR library for SWI-Prolog.

CHR adds a new term besides the known predicates of Prolog: `constraints`. While every goal in Prolog has to be proven immediately by the use of variable binding, unification and backtracking, we can add constraints to a *constraint store*. By defining CHR rules it is possible to manipulate the entries of this constraint store, that means depending on its currently saved constraints it is possible to remove some of them or add new ones.

Effectively the CHR extension adds three types of rules to our logic programming syntax to manipulate the contents of the constraint store. These rules of a CHR program are well-known from the mathematical logic for computer programming: Besides the simplification and propagation rules there is a mixed type called simpagation. Simplification rules replace constraints with usually simpler ones while propagation rules add new constraints depending on the known. Simpagation rules can be used to replace constraints based on known information which is for example useful for removing redundant data. Listing 2.7 shows the concrete syntax of the three rule types.

Listing 2.7: Syntax of CHR rules

```
1 -- Propagation
2 Head ==> Guard | Body.
3
4 -- Simplification
5 Head <=> Guard | Body.
6
7 -- Simpagation
8 Head1 \ Head2 <=> Guard | Body.
```

Unlike Prolog the head of a CHR rule can consist of more than one constraint. The semantic of a propagation rule is “if all constraints mentioned in the rule’s head and the guard can be satisfied, add the constraints of the rule’s body to the constraint store”. The same applies for a simplification rule with the difference, that all constraints of the head are removed from the constraint store. The simpagation rule is a combination of both: The constraints specified in `Head1` are kept, whereas constraints of `Head2` are removed from the constraint store. The guard is optional for all rules and may only

contain built-in predicates, that means only (possibly user-defined) Prolog predicates and no CHR constraints.

If a constraint is added to the constraint store, for each defined CHR rule is checked whether the expected constraints are now all available in the constraint store and the guard can be satisfied. If that is the case the rule gets applied with the semantic introduced before. [Frü95] offers further information about the concrete mechanisms how CHR programs are executed.

In the following we generally refer to constraints as “CHR constraints” to prevent confusions with the definition of the XML Schema. We will call some of its restriction “constraints” too, especially the constraining facets introduced in section 4.2.

3

General Translation Process

The technologies introduced before are the basis for the prototype implementation of a converter that translates XML Schema documents into equivalent JSON Schema documents. The aim was to create a Prolog/CHR module that offers a predicate `xsd2json(XSD,JSON)` which holds the translated JSON Schema `JSON` for the given XML Schema `XSD`. With this module a simple command line tool that directly translates XSD files has been created.

In this chapter we present the general translation process, beginning with the loading of an XML Schema instance into Prolog right up to the creation of the related JSON Schema. The overall translation process is splitted into six steps as shown in Figure 3.1:

1. Read in XML Schema into Prolog
2. XML Flattening

3. Setting Defaults
4. Fragment Translation
5. Wrap JSON Schema
6. Clean up and JSON Output

The different steps are distinguished by their function and used programming language: The read in XML Schema is represented as a nested term in Prolog and gets flattened into multiple CHR constraints which are the basis for the actual translation. The translated fragments are first modelled as CHR constraints and again subsequently wrapped into a nested Prolog term which represents the resulting JSON Schema. In the last step, now by the use of Prolog, unnecessary information gets removed.

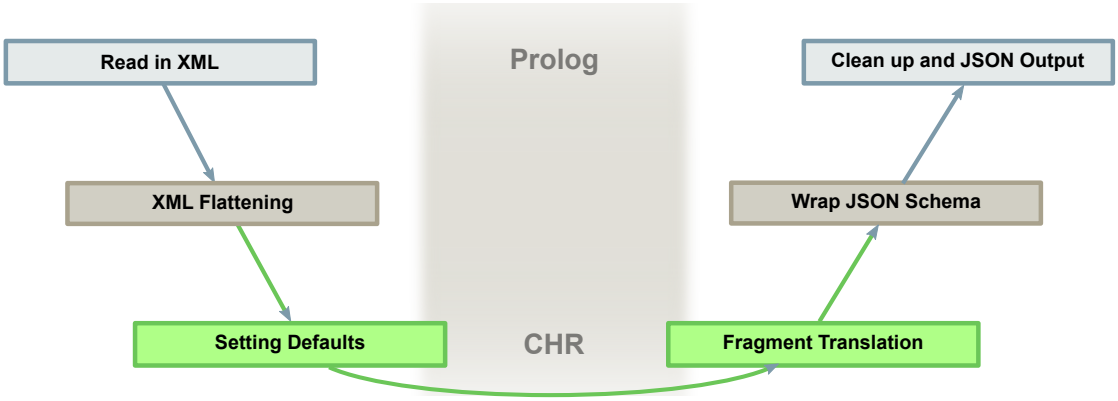


Figure 3.1.: Steps of the overall translation process

The translation steps are also directly mapped to the implementation of the `xsd2json/2` Prolog predicate which generates the translated JSON Schema as seen in Listing 3.1.

Listing 3.1: Definition of the `xsd2json/2` Prolog predicate

```
1 xsd2json(Input,Result) :-
2   % Read in XML
3   %   and generate an ID for its root element
4   load_xsd(Input,XSD),
5   root_id(Root_ID),
```

```
6
7   % XML Flattening
8   %   that also triggers the Setting of the Defaults
9   xsd_flatten_nodes(Root_ID,0,XSD,Children_IDs),
10
11  % Trigger the Translation of single Fragments
12  transform,
13
14  % Trigger the Wrap-up of the JSON Schema and get
15  %   the resulting object from the CHR constraint store
16  Children_IDs = [First_Element|_],
17  build_schema,
18  get_json(First_Element,JSON),
19
20  % Clean up the JSON Schema
21  remove_at_from_property_names(JSON,Result).
```

In the following, we discuss in detail the steps as well as the predicates and CHR constraints used in the definition of `xsd2json/2`. The actual translation rules of concrete XML Schema fragmets, which are triggered when the `transform/0` CHR constraint is added in the third step, have its own chapter in 4. There the translation of XML's basis types are introduced as well as the definition of elements, attributes and complex types.

3.1. Read in XML Schema into Prolog

SWI-Prolog already provides a wide support for the work with XML documents in general and XML Schemas in particular. By the use of its SGML/XML parser [Wie05], XML documents (and so XML Schemas as they are XML documents for themselves) can be read in as a nested term. The library can be loaded via `:- use_module(library(sgml)).` but will be autoloaded in SWI-Prolog in most cases.

The SGML/XML parser provides a Prolog predicate `load_structure(+Source, -ListOfContent, +Options)` [Wie13a] which parses a given XML `Source` to a `ListOfContent`. There is a wide range of `Options` to control for example the handling of namespaces and whitespaces. As we want to parse XML documents, we call `load_structure/3` in the definition of `load_xsd/2` with the `Options` specified in Listing 3.2:

- `space(remove)`
  Whitespace-only nodes are removed, that means line-breaks and spaces which are added in the XML just for layout purposes are ignored.
- `dialect(xmlns)`
  SWI-Prolog's SGML/XML parser has different parsing modes. The given dialect supports the namespaces of the XML elements.
- `xml_no_ns(quiet)`
  Normally, if an XML element has a namespace (like `xs` in `xs:complexType`) which has not been specified via an `xmlns:xs="..."` attribute, the parser will throw an error message. By use of the `quiet` handling, these error messages are suppressed.
- `call(xmlns,register_namespace)`
  If an XML element has a namespace specified, we will call the user-defined predicate `register_namespace/3`.

Listing 3.2: Used options for the `load_structure/3` predicate

```
1 load_xsd(Input,XSD) :-
2   Options = [
3     space(remove),
4     dialect(xmlns),
5     xml_no_ns(quiet),
6     call(xmlns, register_namespace)
7   ],
8   load_structure(Input,XSD,Parse_Options).
```

The `xml_no_ns/1` and `call(xmlns,register_namespace)` directives are used to implement a special behaviour once the XML parser comes across an XML namespace: By the use of the dynamic Prolog predicate `register_namespace/3` the found XML namespaces are saved into new `namespace_uri(Namespace,URI)` facts as shown in Listing 3.3. If an XML element has a namespace which has not been declared before (for example `<xs:complexType>` without a `xmlns:xs="..."` attribute for it or its parent element), the first Prolog rule is used and it will assumed to be the XML Schema namespace `http://www.w3.org/2001/XMLSchema`. By this assumption it is possible to run partial test cases without having to define the used namespace.

Listing 3.3: Creation of new `namespace_uri/2` Prolog facts

```
1  register_namespace(Namespace,Namespace,_Parser) :-
2    asserta(
3      namespace_uri(Namespace,'http://www.w3.org/2001/XMLSchema')
4    ).
5
6  % Namespace and URL given
7  register_namespace(Namespace,URL,_Parser) :-
8    Namespace \== URL,
9    asserta(
10     namespace_uri(Namespace,URL)
11   ).
```

The generated, dynamic `namespace_uri/2` facts will be used in the guards of CHR rules to ensure that its translation rules are applied only for XML Schema elements.

The so defined `load_xsd/2` predicate called with the example XML Schema shown in Listing 2.2 would result in a large nested Prolog list. An extract is presented in Listing 3.4. Additionally a `namespace_uri(xs,'http://www.w3.org/2001/XMLSchema')` fact is created.

Listing 3.4: Calling `load_xsd/2` with the XML Schema example of Listing 2.2

```
1  ?- xsd2json:load_xsd('example.xsd',XSD).
```

```
2
3  XSD = [
4    element(
5      'http://www.w3.org/2001/XMLSchema':schema,
6      [xmlns:xs='http://www.w3.org/2001/XMLSchema'],
7      [
8        element(
9          'http://www.w3.org/2001/XMLSchema':element,
10         [name=catalog],
11         [
12           element(
13             'http://www.w3.org/2001/XMLSchema':complexType,
14             [],
15             [...] % shortened
16           )
17         ]
18       ),
19       element(
20         'http://www.w3.org/2001/XMLSchema':complexType,
21         [name=item],
22         [...] % shortened
23       ),
24       element(
25         'http://www.w3.org/2001/XMLSchema':simpleType,
26         [name=price],
27         [...] % shortened
28       )
29     ]
30   )
31 ].
```

3.2. XML Flattening

As shown in Listing 3.4, the term provided by SWI-Prolog's SGML/XML parser is a list of entities of the form `element(Name,Attributes,Children)`. The element's `Name` might be an atom like `element` or the element's namespace and its tag name separated by a colon. The `Attributes` are specified as a list of key-value pairs and the `Children` is again a list of `element/3` structures or strings and might be empty.

With this complex term returned by the SGML/XML parser translations are very difficult: One has to walk through the whole tree multiple times to get the context of each element, that means its parent element and children and sibling nodes. It might not be possible to translate the entire XML Schema by a single tree traversal. Therefore we want to split the complex, nested term into smaller fragments, so that each node and attribute is represented by a single CHR constraint. The relations between the nodes and their attributes is realized via unique identifiers.

We introduce the following CHR constraints to hold the information of the XML term:

- `node/5`
  `node(Namespace,Name,ID,Children_IDs,Parent_ID)`
  For each node (represented as `element/3` structure in the term) a `node/5` constraint is created that holds its namespace, name, a unique identifier, a list of identifiers of its direct children and the identifier of its parent element.

- `node_attribute/4`
  `node_attribute(ID,Key,Value,Source)`
  For each attribute a new `node_attribute/4` constraint is generated, which holds the identifiers of the related node and the attribute's key and value. The last component is an atom which is set to `source` for all CHR constraints generated in the flattening process. The other possible value, `default`, is reserved for default values and introduced in section 3.3.

- `text_node/3`
  `text_node(ID,Text,Parent_ID)`
  If an element's child is only a string (and no `element/3` structure) 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.

The concrete implementation of the `xsd_flatten_nodes/4` predicate called in the definition of `xsd2json/2` is pretty straight forward and can be found in B.2. The XML flattening for the XML document given in Listing 2.2 would result in multiple CHR constraints, an extract is presented in Listing 3.5.

Listing 3.5: Flattening the XML Schema example of Listing 2.2

```
?- xsd2json:load_xsd('example.xsd',XSD),
     xsd2json:xsd_flatten_nodes([],0,XSD,_).

node(http://www.w3.org/2001/XMLSchema,schema,
  [0],[[0,0],[1,0],[2,0]],[])
node(http://www.w3.org/2001/XMLSchema,complexType,
  [1,0],[[0,1,0],[1,1,0]],[0])
node(http://www.w3.org/2001/XMLSchema,sequence,
  [0,1,0],[[0,0,1,0],[1,0,1,0]],[1,0])
node(http://www.w3.org/2001/XMLSchema,element,
  [0,0,1,0],[],[0,1,0])
...

node_attribute([0],xmlns:xs,
  http://www.w3.org/2001/XMLSchema,source)
node_attribute([1,0],name,item,source)
node_attribute([0,0,1,0],name,name,source)
node_attribute([0,0,1,0],type,xs:string,source)
...
```

As shown in Listing 3.5 the format of the unique identifiers is a combination of the element's position in its current level – that means first child is labelled by `[0]`, second child by `[1]` etc. – and the identifier of its parent element. The creation of the identifiers

can be manipulated by changing the definition of `new_id/3` and its format is therefore irrelevant for the following steps. Only its uniqueness has to be ensured.

3.3. Setting Defaults

The extract provided in Listing 3.5 contains only the information about the nodes that are used in the definition of the XML Schema complex type `<xs:complexType name="item">`. This complex type represents a sequence of XML nodes beginning with `<xs:element name="name" type="xs:string" />`, as shown in Listing 2.2. For this element, its name and and type is defined, but not its quantity. As defined in the XML Schema Specification [W3C04] this quantity of an element can be declared via the `minOccurs` and `maxOccurs` attributes. As they are not set in 2.2, their default value of `1` is used.

For the next step, where the translation of XML Schema fragments like this complex type is done, we want to consult the `minOccurs` and `maxOccurs` attributes no matter if they were defined explicitly or by default. Therefore we simply propagate the `node_attribute/4` CHR constraint for each node and attribute for which the XML Schema specification has defined a default value. To distinguish the default from the explicitly defined attributes, we set the last component of the `node_attribute/4` constraint to `default`. The implementation for the example mentioned above, the propagation of the `minOccurs` and `maxOccurs` attributes for every `xs:element` node, is presented in Listing 3.6.

Listing 3.6: Propagation of default `minOccurs` and `maxOccurs` attributes

```
1 node(Namespace,element,Element_ID,_Element_Children,_Parent_ID)
2   ==>
3     xsd_namespace(Namespace)
4   |
5     node_attribute(Element_ID,minOccurs,'1',default).
6
7 node(Namespace,element,Element_ID,_Element_Children,_Parent_ID)
```

```
8   ==>
9     xsd_namespace(Namespace)
10  |
11    node_attribute(Element_ID,maxOccurs,'1',default).
```

If an attribute has both a default and an explicitly set value, the one set in the XML Schema document should be used. Therefore duplicates of `node_attribute/4` constraints with the same identifier and key should be eliminated. This is realized by the single simpagation rule shown in Listing 3.7.

Listing 3.7: Eliminate `node_attribute/4` duplicates

```
1  node_attribute(ID,Key,_Value_Kept,source)
2    \
3      % remove the default one
4      node_attribute(ID,Key,_Value_Removed,default)
5    <=>
6      % no new constraint added
7      true.
```

To reference an attribute's value in a CHR rule it is enough to leave the last component of `node_attribute/4` unbounded, as it would match no matter if it is set to `source` or `default`.

3.4. Fragment Translation

Before examining the next step we want to have a look at the intended result of the overall translation process: some JSON representation in Prolog. With SWI-Prolog it is possible to (de)serialize JSON by use of its `http/json` library [Wie13b]. Unlike the SGML/XML parser, this library is not loaded by default and therefore has to be loaded via `:- use_module(library(http/json)).` first.

The JSON document given in Listing 2.3 is represented in Prolog by the structure in Listing 3.8.

Listing 3.8: Prolog representation of the example JSON of 2.3

```
1  json([
2    items=json([
3      '4711'=json([
4        name='Unit 1',
5        prices=json([
6          'Euro'=19.99,
7          'Dollar'=27
8        ])
9      ]),
10     '4712'=json([
11       name='Unit 2',
12       prices=json([
13         'Euro'=12.5
14       ])
15     ])
16   ])
17 ])
```

The JSON components described in 2.3 have the following Prolog equivalents:

- Value
  The values can be specified as normal strings, numbers or atoms.
- Array
  An array is represented in Prolog by a list.
- Object
  An object is represented by a `json(List)` structure, where the `List` is of format `[Key1=Value1, Key2=Value, ...]`. The keys must be atoms or strings. So the empty JSON object `{}` would be represented by the Prolog term `json([])`.

By the use of the CHR constraints generated in the step before, small XML Schema fragments can now be translated into their equivalent JSON Schemas, which are represented as JSON objects. Those JSON objects are hold in a `json/2` CHR constraint, whose first component is the identifier of the `node/5` constraint by what the rule has been called, the second component is a JSON representation as defined above.

The concrete translation rules are introduced in the chapter 4, nevertheless we want to present their general structure. Listing 3.9 shows two translation rules related to XML Schema's `xs:element`: The first rule creates a `json/2` constraint for all `xs:element` nodes, whose `type` attribute was set to a basis type (tested in the rule's guard via `valid_xsd_type/2`). The second rule covers all `xs:element` nodes, whose `fixed` attribute has been set.

Listing 3.9: Propagation of `json/2` constraints for `xs:element`

```
1  transform,
2      node(Namespace,element,ID,_Children,_Element_Parent_ID),
3      node_attribute(ID,type,Type_With_NS,_)
4    ==>
5      xsd_namespace(Namespace),
6      valid_xsd_type(Type_With_NS,Type)
7    |
8      convert_xsd_type(Type,JSON),
9      json(ID,JSON).
10
11 transform,
12     node(Namespace,element,ID,_Children,_Element_Parent_ID),
13     node_attribute(ID,fixed,Fixed,_)
14   ==>
15     xsd_namespace(Namespace)
16   |
17     json(ID,json([enum=[Fixed]])).
```

Both rules presented in Listing 3.9, as well as all other translation rules, follow a typical schema:

- Rule's head:
  - One `transform/0` constraint
    The `transform/0` constraint is added in the definition of `xsd2json/2` and triggers the translation of the flattened XML nodes.
  - At least one `node/5` constraint
    Each translated fragment must be assigned to an original XML node. Therefore every translation rule contains at least one `node/5` constraint, whose identifier is referenced in the generated `json/2` constraint. If the translation depends on multiple nodes, for example to get all `xs:sequence` within a `xs:complexType`, their relation is ensured by their unique identifiers: the element's parent identifier is always the last component of the `node/5` constraint. That way siblings, childs and parents can easily be found.
  - Some `text_node/3` and `node_attribute/4` constraints
    Depending on the actual XML Schema fragment, it might be necessary to get the element's text content or some attribute.
  - Some `json/2` constraints
    The entire JSON Schema is built step by step: Beginning with the leaves of the XML Schema tree, the fragments are translated and generate various `json/2` constraints. Those might be picked up for the translation of a parent element. For example, the translation rule of a `xs:sequence` adds sequentially the already generated JSON Schema fragments of its child nodes by using their `json/2` constraints.
- Guard:
  By the use of the Prolog predicates `xsd_namespace/1` (for a single namespace) and `xsd_namespaces/1` (for a list of namespaces) it is ensured that only XML

Schema elements are consulted in the translation process. The concrete definition of these Prolog predicates can be found in B.3.

- Rule's Body:
  In general only a single CHR constraint is added: the generated JSON Schema which is held by a `json/2`.

As already shown in the example of Listing 3.9 there might be multiple CHR rules that will be called for a single node. If an `xs:element` has both a simple type (first rule) and the `fixed` attribute set (second rule), at least two `json/2` constraints with the same identifier are added. To get at the end a single JSON Schema, there must be a rule to combine those `json/2` constraints of the same node. The necessary simpagation rule is shown in Listing 3.10. It uses the predicate `merge_json/3`, which merges two JSON objects into a single one. Its definition can be found in B.6.

Listing 3.10: Merge multiple `json/2` constraints of the same identifier

```
1 json(ID,JSON1),
2     json(ID,JSON2)
3   <=>
4     merge_json(JSON1,JSON2,JSON)
5   |
6     json(ID,JSON).
```

In addition to the `json/2` constraint we introduce another constraint which holds the JSON representation of an XML Schema fragment: For each type defined in the XML Schema, in other words all `xs:complexType` and and `xs:simpleType` nodes with a `name` attribute set, a `schema_definition/3` constraint is propagates which holds its JSON Schema representation. In this way it is possible to reference the JSON Schema translation of globally defined XSD types by a single CHR constraint which is used in the next step. The `schema_definition/3` constraint has three components: The name of the defined XSD type (that means the actual value of its `name` attribute), the ID of the `xs:complexType` respectively `xs:simpleType` node and its JSON Schema

translation. The concrete rules that propagates suchlike `schema_definition/3` constraints can be found in section 4.3.

3.5. Wrap JSON Schema

Because the XML Schema fragments are translated one by one, the previous step will terminate as soon as the root element of the given XML Schema has been translated, in other words all CHR rules applicable for the `node/5` with the identifier of the root element have been called. The root element of the produced JSON Schema has a special role as it can hold global type definitions (similar to the first-level, named `xs:complexType` and `xs:simpleType` in the XML Schema example of Listing 2.2) in a `definitions` object. This way it is possible to define a type globally and reference it in other type definitions. An example can be found in Listing 3.11. The `definitions` key could be part of any (nested) JSON Schema, but we use it only in the top level. The so defined, named sub schemas can be referenced via a special `$ref` property whose value must be an XPath-like expression to the location of the definition. The root object is referenced by a sharp `#`.

Listing 3.11: Global type definition in JSON Schema

```
1  {
2    %% -- global definition
3    "definitions": {
4      "foo": {
5        "type": "number",
6        "minimum": 42
7      }
8    },
9
10   %% -- referencing it in another schema
11   %% --   via the $ref key
12   "type": "object",
```

```
13    "properties": {
14      "bar": {
15        "$ref": "#/definitions/foo"
16      }
17    }
18  }
```

In this step, the wrap-up of the produced JSON Schema, we want to combine all the `schema_definition/3` constraints with the `json(ID,JSON)` constraint of the root element. For that reason the CHR rule in Listing 3.12 propagates a new `json/2` constraint with a `definitions` property set that includes the sub schema. Those new `json/2` constraints get merged into a `json/2` constraint by the simpagation rule already mentioned in Listing 3.10.

Listing 3.12: Wrap-up of `schema_definition/3` constraints

```
1  build_schema,
2      schema_definition(Name,_ID,Inline_Schema),
3      node(Namespace,schema,Schema_ID,_Schema_Children,_)
4    ==>
5      xsd_namespace(Namespace),
6      first_id(Schema_ID)
7    |
8      json(Schema_ID,json([
9        definitions=json([Name=Inline_Schema])
10     ])).
```

3.6. Clean up and JSON Output

To return the generated JSON Schema by the use of `xsd2json/2` it is necessary to get the `json/2` constraint for the root element. Therefore we introduce a new `get_json/2` constraint which will be added with the identifier of the root element as its first component

an unbounded variable as the second. By the simpagation rule of Listing 3.13 the free variable gets bounded to the finished JSON Schema.

Listing 3.13: Bind root element's JSON

```
1 json(ID,JSON)
2   \
3     get_json(ID,Result)
4   <=>
5     JSON = Result.
```

Finally, the created JSON Schema object gets cleaned up: In the creation process, the names of XML attributes (specified as `xs:attribute` in the XML Schema) that are translated into equivalent JSON Schema objects are prefixed with an `@`. If there is no `xs:element` in this `xs:complexType` with the same name, the `@` is removed from the attribute's name by use of a `remove_at_from_property_names/2` Prolog predicate in the `xsd2json/2` definition. This predicate removes the `@` only in JSON objects whose key is `properties`. Its implementation and an example can be found in B.8.

4

Translation Rules

In the previous chapter we introduced the general translation process. In a first step an XML document, given as a nested Prolog term, is flattened into a number of `node/5`, `node_attribute/4` and `text_node/3` CHR constraints. Next, these constraints should be taken to translate fragments of the XML Schema into its equivalent JSON Schema.

After having read in and flattened the XML Schema, the translation process for XML Schema fragments is triggered by adding a `transform/0` constraint to the constraint store. The general structure of such translation rules has already been introduced in section 3.4 as well as common Prolog predicates that are used in the rules' guards.

The translation of fragments happens one by one, beginning in general with the leaves of the tree that represents an XML Schema. It is obvious that in most cases this will be elements with either a self-defined data type (`xs:simpleType`; see also 3.5) or a

standard XSD data type like `xs:string` or `xs:nonNegativeInteger`. In this chapter we want first focus on the translation of such XSD data types and thereafter introduce the concrete translation rules of typical XML Schema fragments.

4.1. XSD Primitive Types

XML Schema provides a big number of predefined data types. The complete list can be found in [PGM+08]. By contrast, the number of data types defined for JSON Schema [KZ13] is quiet limited. There are only seven primitive types:

- `array`
- `boolean`
- `integer`
- `number`
- `null`
- `object`
- `string`

To restrict those primitive types, the JSON Schema Validation specification [Gal13] allows also a semantic validation by the use of the `format` keyword. In that way it is for example possible to set the `type` of a value to `string` and restrict its format to `email`. Nevertheless the consideration of these formats is not mandatory for validation implementations. We will use them if possible.

Although the number of primitive data types might seem higher for XML Schema, at the very most each can be translated into an equivalent JSON Schema, due to the fact, that for example `xs:nonNegativeInteger` is only an `xs:integer` with some restrictions, which can be modelled in a JSON Schema as well.

The JSON Schema translation for each XML Schema data type is presented in table 4.1.

Table 4.1.: Translation of simple XSD data types

XML Schema type	JSON Schema type definition
`xs:string`	`{ "type": "string" }`
`xs:boolean`	`{ "type": "boolean" }`

`xs:float`
`xs:double`
`xs:decimal`
	`{ "type": "number" }`

`xs:integer`	`{ "type": "integer" }`
`xs:positiveInteger`	`{ "type": "integer", "minimum": 0, "exclusiveMinimum": true }`
`xs:negativeInteger`	`{ "type": "integer", "maximum": 0, "exclusiveMaximum": true }`

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

This list of translations is directly implemented as a `convert_xsd_type/2` Prolog predicate, an example of its usage was already shown in Listing 3.9.

4.2. Constraining Facets

While XML Schema provides predefined data types like `xs:positiveInteger`, which imply a minimum, those restrictions of the value range could be declared also via restrictions of the basis type `xs:integer`. Those restrictions like XML Schema's `xs:mimumum` are called *constraining facets*. They can be specified as a restriction of a basis type via an `xs:restriction` node. Listing 4.1 shows an equivalent, self-defined XSD data type which is equivalent to the predefined `xs:positiveInteger`.

Listing 4.1: `xs:positiveInteger` as restriction of `xs:integer`

```
1 <xs:simpleType name="myPositiveInteger">
2   <xs:restriction base="xs:integer">
3     <xs:minExclusive value="0" />
4   </xs:restriction>
5 </xs:simpleType>
```

By the use of `xs:restriction` within an `xs:simpleType` a new so called *derived data type* is defined *by restriction*. It is not only possible to derive from primitive data types but also from derived data types as well. In that way it would be possible to create a new data type derived from `myPositiveInteger` which adds for example an upper bound as constraining facet.

For the translation process of the `xs:restriction` XSD fragment with its child nodes it does not matter if a primitive or derived data type is the basis type specified in the `base` attribute – the translated constraining facets are simply added. Therefore we can again establish a set of translation rules for all constraining facets as shown in table 4.2, with `X` standing for a number or string in case of `xs:pattern`.

Table 4.2.: Translation of constraining facets (X is placeholder)

XML Schema constraining facet	JSON Schema equivalent
`<xs:minExclusive value="X" />`	`{ "minimum": X, "exclusiveMinimum": true }`
`<xs:maxExclusive value="X" />`	`{ "maximum": X, "exclusiveMaxmimum": true }`
`<xs:minInclusive value="X" />`	`{ "minimum": X, "exclusiveMinimum": false }`

`<xs:maxInclusive value="X" />`	`{ "maximum": X, "exclusiveMaximum": false }`
`<xs:minLength value="X" />`	`{ "minLength": X }`
`<xs:maxLength value="X" />`	`{ "maxLength": X }`
`<xs:length value="X" />`	`{ "minLength": X, "maxLength": X }`
`<xs:pattern value="X" />`	`{ "pattern": "X" }`

Not all restrictions are allowed for every basis data type [PGM+08]. But as we assume that the given XML Schema document is valid, we do not carry out tests.

The translation rules for the different constraining facets given by table 4.2 are defined as `convert_xsd_restriction/3` predicates in Prolog like it is shown in Listing 4.2 for `xs:minExclusive`. The `json_true/1` predicate binds the given variable to the representation of boolean true for SWI-Prolog's JSON library. `to_number/2` converts a given numeric string into the equivalent number.

Listing 4.2: Definition of `convert_xsd_restriction/3` for `xs:minExclusive`

```
convert_xsd_restriction(minExclusive,Value,json(JSON_List)) :-
  to_number(Value,Number),
  json_true(True),
  JSON_List = [minimum=Number,exclusiveMinimum=True].
```

The so defined `convert_xsd_restriction/3` facts are used by the simpagation rule shown in 4.3 to translate the `xs:restriction` node that contains a constraining facet.

Listing 4.3: Translation of `xs:restriction` with a constraining facet

```
transform,
    node(NS1,restriction,Restriction_ID,
      _Restriction_Children,_Restriction_Parent_ID),
    node(NS2,Constraint_Name_XSD,Constraint_ID,
      _Constraint_Children,Restriction_ID),
    node_attribute(Constraint_ID,value,Value,_)
  ==>
    xsd_namespaces([NS1,NS2]),
    convert_xsd_restriction(
      Constraint_Name_XSD,Value,json(JSON_List))
  |
    json(Restriction_ID,json(JSON_List)).
```

As for each constraining facet within an `xs:restriction` node a new `json/2` CHR constraint gets propagated. If there are multiple child nodes, they will simply get combined via `merge_json/3` as introduced in B.6. If the same constraining facet occurs multiple times, its more specific restriction is used. The only exception is the occurrence of multiple `xs:pattern` facets: As specified in [PGM+08] their regular expressions are ORed.

The constraining facets are not the only restriction of the derived data type. The basis type, specified via the `base` attribute of the `xs:restriction` node, also has to be

considered. We translate the basis type via the table 4.1 of section 4.1 and propagate its JSON equivalent too. This rule is shown in Listing 4.4.

Listing 4.4: Translation of `xs:restriction` with its `base` attribute

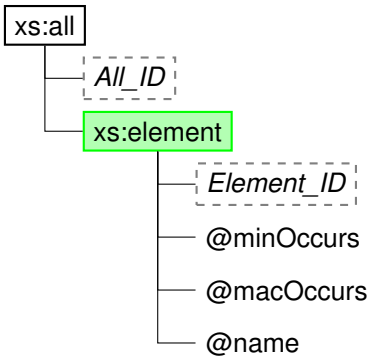
```
1  transform,
2      node(Namespace,restriction,ID,_Children,_Parent_ID),
3      node_attribute(ID,base,Base,_)
4    ==>
5      xsd_namespace(Namespace),
6      namespace(Base,Base_Namespace,Base_Type),
7      xsd_namespace(Base_Namespace),
8      convert_xsd_type(Base_Type,JSON)
9    |
10     json(ID,JSON).
```

4.3. Translation of nested XSD Elements

With the translation rules mentioned in the two sections before we only know how to translate `xs:element` and `xs:attribute` nodes with either a basis type specified via its `type` attribute or a derived type by the use of `xs:simpleType` and `xs:restriction`. However an XML Schema is more than only the definition of simple types: Via `xs:complexType` nodes attributes of elements can be specified as well as their child nodes. In that way, the occurrence of an `xs:element` within an `xs:sequence` node is for example a very common fragment of an XML Schema instance. In this section, we want to show how to translate typical nested XSD nodes, depending on its sibling elements and given attributes.

The general approach has already been introduced in section 3.4: 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. Two examples, which do not depend on already translated `json/2` constraints, were shown in Listing 3.9.

Figure 4.1.: `xs:element` within an `xs:all` node



To illustrate the XML Schema fragments and their JSON Schema translations we present an extract of the XML Schema tree. An example for the translation of an `xs:all` node which has an `xs:element` child node is shown in figure 4.1.

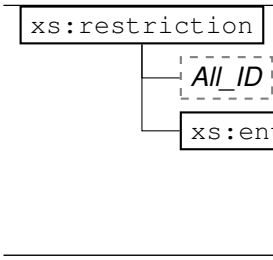
As shown in figure 4.1, XSD elements are represented by framed nodes and their ID in a dashed box. For all green highlighted nodes a `json/2` constraint with the same identifier already exists, that means in the example of figure 4.1 there is already a `json(Element_ID,JSON)` constraint in the CHR constraint store. For all nodes that are not highlighted green there might be already a related `json/2` constraint, but it will not be examined for this concrete translation rule. Boxes with the title `text` that are highlighted gray stand for `text_node/3` constraints.

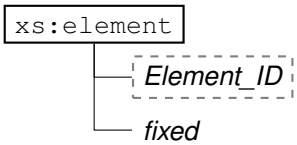
The used attributes of an XSD element are prefixed with an `@`, that means that for the example of figure 4.1 the `minOccurs`, `maxOccurs` and `name` attributes are used in the head of the CHR rule. The generated JSON Schema equivalents are modelled as in tables 4.1 and 4.2. The values of the attributes are referenced by variables of the same name in upper CamelCase. In that way we refer to the value of the `maxOccurs` attribute of the XML Schema fragment as `MaxOccurs` in the related JSON Schema fragment.

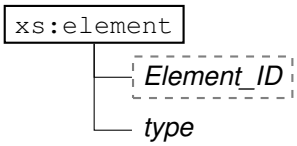
If necessary we provide conditions under which the translation rule could apply. Those can be directly implemented as guards in the CHR rules. The check, if the given elements are of the XML Schema namespace as already introduced in section 3.4, are conditions for all rules and therefore not listed explicitly.

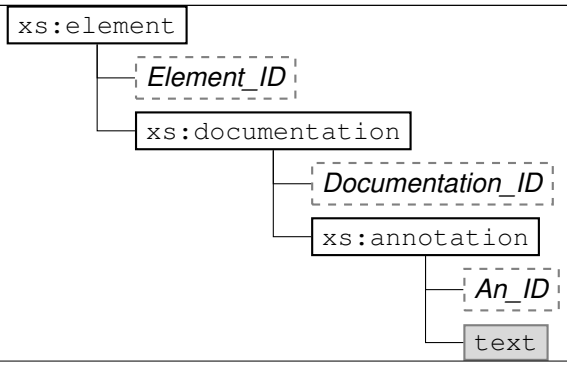
Table 4.3.: Translation of nested XSD elements

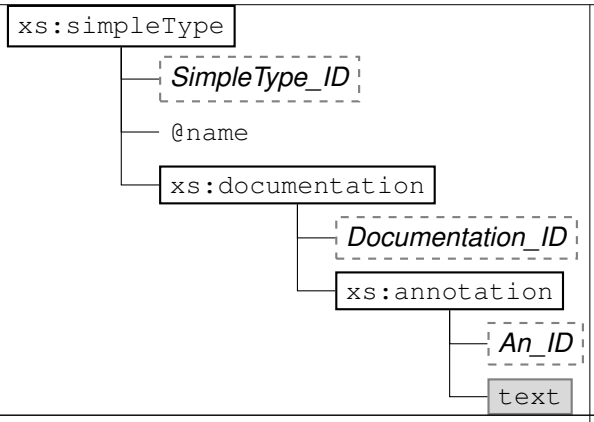
XML Schema Fragment	JSON Schema equivalent

	`{`
	`"enum": [Value]`
	`}`

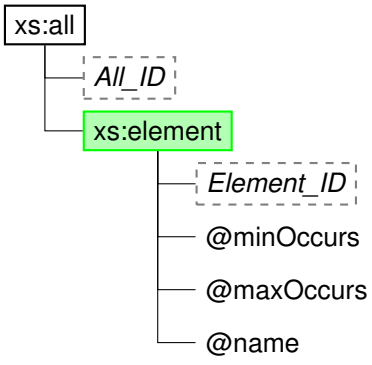
	`{`
	`"enum": [Fixed]`
	`}`

	`{`
	`"type": Type`
	`}`
	(with `Type` not being a basis or derived type)

	`{`
	`"description": Text`
	`}`

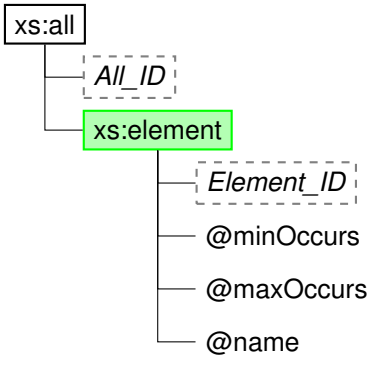



```
{
  "description": Text
}
```



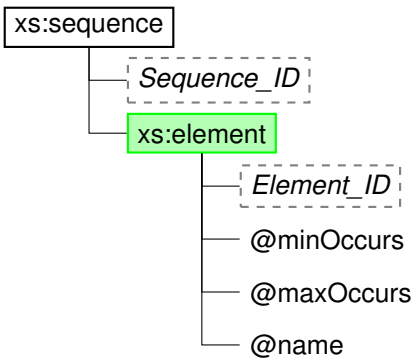

```
{
  "type": "object",
  "properties": {
    Name: JSON
  },
  required: [Name]
}
```

(if `is_required_property/2`, see B.9, is true)



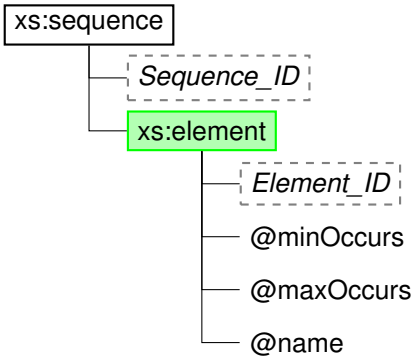

```
{
  "type": "object",
  "properties": {
    Name: JSON
  }
}
```

(if `is_required_property/2`, see B.9, is false)



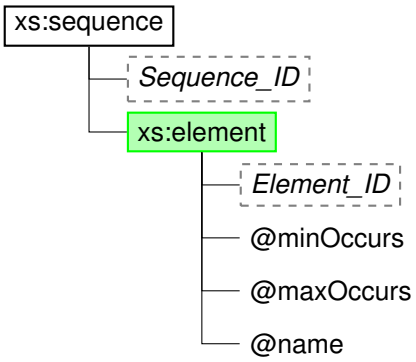

```
{
  "type": "object",
  "properties": {
    Name: JSON
  },
  required: [Name]
}
```

(if `MinOccurs=1` and `MaxOccurs=1`)



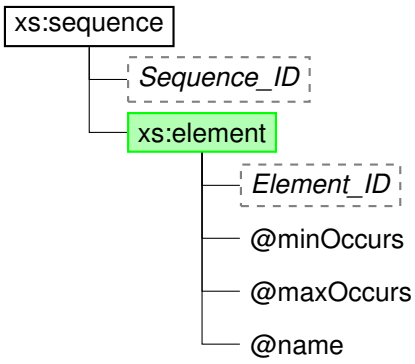

```
{
  "type": "object",
  "properties": {
    Name: JSON
  }
}
```

(if `MinOccurs=0` and `MaxOccurs=1`)



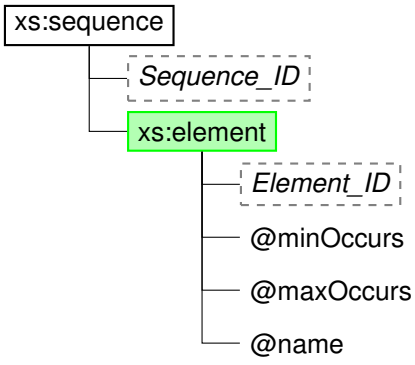

```
{
  "type": "object",
  "properties": {
    Name: {
      "type": "array",
      "items": JSON,
      "minItems": MinOccurs
    }
  },
  required: [Name]
}
```

(if `MinOccurs>0` and `MaxOccurs=unbounded`)



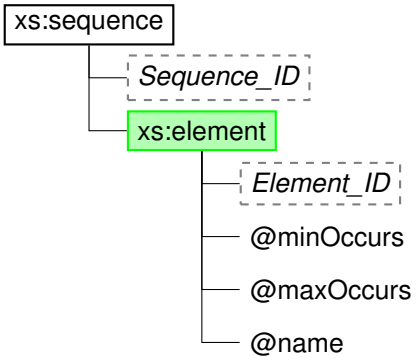

```
{
  "type": "object",
  "properties": {
    Name: {
      "type": "array",
      "items": JSON,
      "minItems": MinOccurs
    }
  }
}
```

(if `MinOccurs=0` and `MaxOccurs=unbounded`)



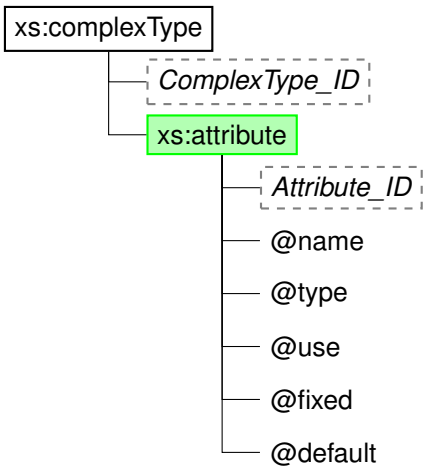

```
{
  "type": "object",
  "properties": {
    Name: {
      "type": "array",
      "items": JSON,
      "minItems": MinOccurs,
      "maxItems": MaxOccurs
    }
  },
  required: [Name]
}
```

(if `MinOccurs>0` and `MaxOccurs` is not `unbounded`)



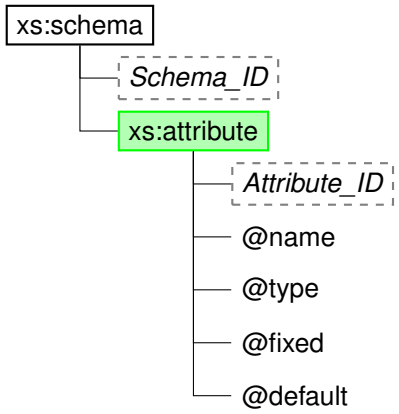

```
{
  "type": "object",
  "properties": {
    Name: {
      "type": "array",
      "items": JSON,
      "minItems": MinOccurs,
      "maxItems": MaxOccurs
    }
  }
}
```

(if `MinOccurs=0` and `MaxOccurs` is not `unbounded`)




```
{
  "type": "object",
  "properties": {
    @Name: {
      "type": JSON,
      "enum": [Fixed]
    }
  },
  required: [@Name]
}
```

Note: The name of a property given as attribute is prefixed with an `@` which might be removed later.
The values of the `required`, `enum` and `enum` keys depend on the attributes of the `xs:attribute` node. The complete implementation of the handling of `xs:attribute` nodes can be found in the appendix C.1.




```
{
  "definitions": {
    @Name: {
      "type": JSON,
      "enum": [Fixed]
    }
  }
}
```

Note: Because @Name is only an identifier in the `definitions` part the prefix-`@` will not be removed later.
The values of the `required`, `enum` and `enum` keys depend on the attributes of the `xs:attribute` node. The implementation is similar to the one shown in the appendix C.1.

Not all translation rules can be illustrated as in figure 4.1 and table 4.3: For the `xs:sequence` and `xs:all` nodes the translation rules specified in 4.3 are used, whereas their parent `xs:complexType` node simply uses this generated JSON Schema fragment without adding any further constraints. This behaviour is implemented in the CHR rules shown in Listing 4.5.

Listing 4.5: Translation of `xs:complexType` with nested `xs:sequence` or `xs:all`

```
1 % xs:all
2 transform,
3     node(NS1,complexType,ComplexType_ID,
4       _ComplexType_Children,_ComplexType_Parent_ID),
5     node(NS2,all,All_ID,_All_Children,ComplexType_ID),
6     json(All_ID,All_JSON)
7   ==>
```

```
8       xsd_namespaces([NS1,NS2])
9     |
10      json(ComplexType_ID,All_JSON).
11
12  % xs:sequence
13  transform,
14      node(NS1,complexType,ComplexType_ID,
15        _ComplexType_Children,_ComplexType_Parent_ID),
16      node(NS2,sequence,Sequence_ID,
17        _Sequence_Children,ComplexType_ID),
18      json(Sequence_ID,Sequence_JSON)
19    ==>
20      xsd_namespaces([NS1,NS2])
21    |
22      json(ComplexType_ID,Sequence_JSON).
```

By the use of the rules given in table 4.1 it is possible to translate primitive XML Schema data types into their equivalent JSON Schema types and restrictions. The rules presented in section 4.2 are used to create new types as a derivation by restriction. With help of the translation rules given in section 4.3 it is possible to combine elements and attributes of these (derived) basis types into more complex ones and therefore to create an expressive XML Schema.

5

Evaluation

5.1. Test Framework

The `xsd2json` module has been developed by a bottom-up approach: Following the idea of test-driven development, a separate test framework was an important part of the development cycle. This test framework is not implemented in Prolog like the `xsd2json` module but in JavaScript with node.js. Node.js is a software platform known for its scalable, non-blocking architecture and especially used to program lightweight web server.

Due to its growing popularity there are a great many open source modules for node.js. One of them is `interpreted` [Mad13], a wrapper for node.js to perform input/output tests. We use it to compare the results generated by the `xsd2json` command line tool

with its expected JSON Schema equivalent. Eventual differences are exported as TAP messages, following the *Test Anything Protocol* [Les13].

The test framework is part of the `xsd2json` development suite and therefore part of the published [Nog13] source code. Its manual is listed in the appendix A.2. There are currently more than 60 test cases specified as XML Schema and JSON Schema documents. Unfortunately a full iteration over all test files takes some time because the command line interface is way slower than the programmatical usage of `xsd2json`. See appendix A for further information.

5.2. Current Limitations

XML Schema has some features for which there are not yet equivalents in JSON Schema: The lack of an XPath-like way to address a specific property inside a nested JSON prevents the translation of keys. Therefore the XML Schema elements `xs:key`, its referencing element `xs:keyref` can not be translated because they specify the key-elements in their `xs:selector` and `xs:field` nodes as XPath expressions. The same applies for XML Schema's `xs:unique` element: Although JSON Schema provides a way to ensure unique identifiers in a sub schema, the XPath expression can not be translated yet.

Another missing feature is the handling of referenced XML Schema documents. With the aid of using the `xs:import` and `xs:include` XML Schema nodes it is possible to reference XML Schemas specified in other files. While the JSON Schema specification provides a similar feature, this has not been implemented in `xsd2json`, which translates only a single file as of yet. Therefore it would be possible to translate every single file, but the references would have set manually. To support sub schemas specified via `xs:import` and `xs:include` the handling of XML namespaces has to be improved because in every XML Schema file the same namespace could be assigned to different URIs (confer section 2.2).

6

Conclusion

In this chapter the results of the thesis are summarized and future improvements of the `xsd2json` module discussed.

6.1. Summary

In this work the metalanguages XML Schema and JSON Schema were compared with the aim to provide a set of translation rules of typical use cases. Beginning with the basis types of XML Schema, we presented sub schemas in JSON Schema that guarantee nearly the same semantics.

In a next step XML Schema's concept of derivation by restriction has been introduced. Because there is no native derivation in JSON Schema we implemented rules that directly consider the constraining facets specified in an `xs:restriction` XML Schema node.

With the help of these rules it was possible to translate XML Schema fragments that specify data types, regardless whether primitive or derived and whether being called directly within an `type` attribute or as part of an `xs:simpleType` definition.

With the help of translation rules for `xs:complexType` nodes and its children it was possible to embed those primitive and derived typed elements and attributes within a sequence of properties. With respect to their allowed occurrences, specified as `minOccurs` and `maxOccurs` attributes, the JSON Schema type was dynamically set to a primitive type, object or array.

The elaborated translation rules, written in CHR, were embedded in an Prolog predicate `xsd2json/2`. The overall process was divided into six steps, beginning from the read in of an XML Schema document, followed by the flattening of the nested Prolog term, right up to the assembling of the translated JSON Schema fragments.

6.2. Outlook

Not all use cases could be implemented as of yet: As mentioned earlier in section 5.2 there is no way to translate `xs:key` and `xs:keyref` elements. It is also not possible to ensure uniqueness via `xs:unique`. All these elements use XPath expressions to address elements within an XML document, but JSON does not have an official equivalent yet. A possible way to translate those elements even so would be to evaluate the XPath expressions in the translation process to get the addressed nodes. SWI-Prolog already provides an XPath handling, but only for XML documents. It would be more difficult to apply a given XPath expression not to an XML instance but its Schema document. Nevertheless the translation of XPath expressions is a key issue as it was also the reason to restrict our work to the older XML Schema specification 1.0 instead of 1.1. As explained in section 2.2 the newer XML Schema 1.1 specification makes extensive use of XPath expressions.

While we introduced the concept of derivation by restriction, XML Schema provides another form to create new data types, namely by extension. It is possible to extend the complex content of an already defined XSD type by adding new attributes and

elements to its `xs:sequence` and `xs:all` nodes. Although the addition of attributes would be a single CHR rule, the extension of `xs:sequence` and `xs:all` nodes are more complicated.

Following the test-driven development approach it was possible to translate expressive XML Schema instances. But because of missing a precise survey, which XML Schema node can be a child of which other, it might not be clear that all use cases are covered yet. Therefore one might be strive for a grammar for XML Schema in terms of a hyper schema.

A

User Manuals

In this appendix there are the user manuals and installation guides for both the `xsd2json` Prolog/CHR library and its test framework.

A.1. xsd2json

`xsd2json` is a library for Prolog and CHR to translate a XML Schema file into equivalent JSON Schema. It can be used both programmatically and as a command line tool.

A.1.1. Installation

All you need is SWI-Prolog. See there for installation instructions.

A.1.2. Usage

`xsd2json` provides a command line interface. An example usage is shown in listing A.1.

Listing A.1: Call `xsd2json` from the command line

```
swipl --quiet --nodebug --g 'main,halt' \
  -s cli.pl -- < /path/to/your.xsd
```

Unfortunately the command line version is way slower than using `xsd2json` programmatically, that means by directly calling it in Prolog. The `xsd2json.pl` module provides a predicate `xsd2json/2` which can be used to convert a given XML Schema file into the equivalent JSON Schema. It can be called via `swipl -s xsd2json.pl` followed by `xsd2json('/path/to/your.xsd',JSON)`, which binds the `JSON` variable to the created JSON Schema. A prettier output can be generated by using the `json_write/2` predicate of SWI-Prolog's `http/json` library. An example usage is presented in listing A.2. This will print the produced JSON Schema in front of the content of the CHR constraint store, so it might be necessary to scroll up the output.

Listing A.2: Call `xsd2json/2` within SWI-Prolog

```
?- use_module(library(http/json)).

?- xsd2json('/path/to/your.xsd',JSON),
  json_write(user_output,JSON).
```

A.2. Test Framework

This command line tool is written with node.js and provides some functionalities to support the test-driven development of the `xsd2json` library.

A.2.1. Installation

The tests are run by node.js, version `0.10` or later is required. Before using the testing suite its dependencies must be installed via `npm install`.

A.2.2. Provided Tests

The command line tool provides three commands: `interpreted`, `transform` and `validate-json`.

Run interpreted Tests

Each file in the `xsd` directory will be converted to a JSON Schema instance. Its output gets compared with the related JSON file in the `json` directory. The test output, containing confirmation or differences, is produced following the Test Anything Protocol (TAP; [Les13]).

It is possible to run the interpreted tests for all files via `node test.js interpreted`. By specifying files via the `-ignore` parameter it is possible to exclude some, for example as in listing A.3 the files `xsd/schema2.xsd` and `xsd/schema3.xsd`.

Listing A.3: Exclude file from interpreted test

```
1 node test.js interpreted --ignore schema2 --ignore schema3
```

It is possible to restrict the interpreted tests to several files by using the `-file` parameter. It is similar to the `-ignore` flag but allows regular expressions, encapsulated in slashes, too.

By calling `node test.js interpreted -files` a list of possible test files is displayed. `node test.js interpreted -help` lists all options, their defaults and possible values.

Validate tested JSON Output

As the expected JSON output files in the `json` subfolder are created manually it might be useful to check if they really satisfy the JSON Schema Core Meta-Schema [JSO13b], that means if they are valid JSON Schema instances. This test can be run via `node test.js validate-json`. It produces a TAP output too.

A.2.3. Pretty TAP output

To get a better visual experience about passed and failed tests it is possible to pipe the normal TAP output to the `tap-prettify` module like shown in listing A.4.

Listing A.4: Pretty TAP output with `tap-prettify`

```
1 node test.js validate-json | \
2   node node_modules/tap-prettify/bin/tap-prettify.js -
```

A.2.4. Transform a single XSD File

For testing purposes it might be useful to a single XSD file via `node test.js transform`. The XML Schema file can be either referenced by the `-input` flag or directly piped to standard input. The `transform` command simply pipes input and output of the given file to the `xsd2json` command line interface.

B

Source Codes of Prolog Predicates

In this appendix there are several important source codes of some Prolog predicates.

B.1. `xsd_flatten_attributes/2`

The `xsd_flatten_attributes(ID,List)` Prolog predicate is used to flatten a given `List` of attributes of the form `[Key1=Value, Key2=Value, ...]` for the element with the identifier `ID`.

Listing B.1: Implementation of `xsd_flatten_attributes/2`

```
1 xsd_flatten_attributes(_ID,[]).
2
3 xsd_flatten_attributes(ID,
```

```
4   [Attribute=Value|List_Of_Attributes]) :-
5   node_attribute(ID,Attribute,Value,source),
6   xsd_flatten_attributes(ID,List_Of_Attributes).
```

B.2. xsd_flatten_nodes/4

The `xsd_flatten_nodes(Base_ID,Position,Nodes,Children_IDs` Prolog predicate is used to flatten a given, nested XML term `Nodes`, for example the result of `load_xsd(Input,Nodes)`, into a number of `node/5`, `node_attribute/4` and `text_node/3` CHR constraints.

Listing B.2: Implementation of `xsd_flatten_nodes/4`

```
1  xsd_flatten_nodes(_Base_ID,_Pos,[],[]).
2
3  xsd_flatten_nodes(Base_ID,Pos,[Node|Nodes],[ID|Sibling_IDs]) :-
4    % is an XML node, no text
5    Node = element(Node_Type,Node_Attributes,Child_Nodes),
6    new_id(Base_ID,Pos,ID),
7    namespace(Node_Type,Namespace,Node_Type_Without_NS),
8    % flatten the node's attributes
9    xsd_flatten_attributes(ID,Node_Attributes),
10   node(Namespace,Node_Type_Without_NS,ID,Children_IDs,Base_ID),
11   % flatten sibling nodes
12   Next_Pos is Pos+1,
13   xsd_flatten_nodes(Base_ID,Next_Pos,Nodes,Sibling_IDs),
14   % flatten all children
15   xsd_flatten_nodes(ID,0,Child_Nodes,Children_IDs).
16
17 xsd_flatten_nodes(Base_ID,Pos,[Node|Nodes],[ID|Sibling_IDs]) :-
18   atom(Node),  %% is simply a text node
19   new_id(Base_ID,Pos,ID),
```

```
20   text_node(ID,Node,Base_ID),
21   % flatten sibling nodes
22   Next_Pos is Pos+1,
23   xsd_flatten_nodes(Base_ID,Next_Pos,Nodes,Sibling_IDs).
```

B.3. xsd_namespace/1

The `xsd_namespace(Value)` predicate is true if the given `Value`, which could be an atom or string, is the XML Schema namespace `http://www.w3.org/2001/XMLSchema`. This might only be the case, if a `namespace_uri/2` fact exists for this namespace, which might be generated during the XML parsing.

Listing B.3: Implementation of `xsd_namespace/1`

```
1  xsd_namespace('http://www.w3.org/2001/XMLSchema').
2  xsd_namespace(Namespace) :-
3    namespace_uri(Namespace,'http://www.w3.org/2001/XMLSchema').
```

B.4. xsd_namespaces/1

The `xsd_namespaces(List)` predicate is true if all the atoms or string given in the `List` are known XML Schema namespaces. This is checked via `xsd_namespace/1` B.3.

Listing B.4: Implementation of `xsd_namespaces/1`

```
1  xsd_namespaces([]).
2  xsd_namespaces([Namespace|Namespaces]) :-
3    xsd_namespace(Namespace),
4    xsd_namespaces(Namespaces).
```

B.5. `lookup/4`

By the use of `lookup(Key,List,Value,List_Without_Value)` it is possible to search for a given `Key` in a `List` of key-value pairs in the form `[Key1=Value1,...]`. The fourth component contains the `List` without the search key-value pair.

Listing B.5: Implementation of `lookup/4`

```
1 lookup(Key,[Key=Value|Without_Key],Value,Without_Key).
2 lookup(Key,[Not_Key=Some_Value|Rest],Value,
3   [Not_Key=Some_Value|Without_Key]) :-
4     Key \= Not_Key,
5     lookup(Key,Rest,Value,Without_Key).
```

B.6. `merge_json/4`

`merge_json(JSON1,JSON2,Result,On_Conflict` merges two given JSON objects `JSON1` and `JSON2` into a single one `Result`. If `JSON1` and `JSON2` contain an object with an identical key but different value, it fails if `On_Conflict` is `0` or `hard`, and succeeds with a renaming if `On_Conflict` is `9` or `soft`. The most common usage is with `On_Conflict=hard`. Only for attributes a renaming might be advisable.

Listing B.6: Implementation of `merge_json/4`

```
1 merge_json(JSON1,JSON2,Merged,soft) :-
2   merge_json(JSON1,JSON2,Merged,9).
3 merge_json(JSON1,JSON2,Merged,hard) :-
4   merge_json(JSON1,JSON2,Merged,0).
5
6 merge_json(JSON1,JSON2,_Merged,_On_Conflict) :-
7   (var(JSON1); var(JSON2)), !, false.
8
9 merge_json(json([]),json(JSON_List2),
```

```
10    json(JSON_List2),_On_Conflict).
11
12  merge_json(json([Key=Value|Rest_JSON_List1]),json(JSON_List2),
13    json(Merged),On_Conflict) :-
14    % Key also exists in JSON_List2 and value is equal
15    lookup(Key,JSON_List2,Value,JSON2_Without_Key),
16    merge_json(json(Rest_JSON_List1),json(JSON2_Without_Key),
17      json(Rest_Merged),On_Conflict),
18    Merged = [Key=Value|Rest_Merged].
19
20  merge_json(json([Key=Value|Rest_JSON_List1]),json(JSON_List2),
21    json(Merged),On_Conflict) :-
22    % Key also exists in JSON_List2 and value is no atom
23    lookup(Key,JSON_List2,Value_in_JSON_List2,JSON2_Without_Key),
24    \+atom(Value),
25    % If `Key` is `required` or `enum` use union instead of
26    %   the merge_json/3 predicate which would result
27    %   in an append of both lists.
28    % This might be necessary due to different orders to
29    %  apply the CHR rules.
30    (
31        (Key == required; Key == enum),
32        union(Value,Value_in_JSON_List2,Merged_Value)
33      ;
34        Key \== required,
35        Key \== enum,
36        merge_json(Value,Value_in_JSON_List2,Merged_Value)
37    ),
38    % merge the rest of the lists independently of the
39    %   current key
40    merge_json(json(Rest_JSON_List1),json(JSON2_Without_Key),
```

```
41      json(Rest_Merged),On_Conflict),
42    Merged = [Key=Merged_Value|Rest_Merged].
43
44  merge_json(json([Key=Value|Rest_JSON_List1]),json(JSON_List2),
45    json(Merged),On_Conflict) :-
46    % Key also exists in JSON_List2 and value is no atom
47    lookup(Key,JSON_List2,_Value_in_JSON_List2),
48    \+atom(Value),
49    % couldn't be merged --> rename
50    On_Conflict == 9,
51    New_Key = '@'Key,
52    merge_json(json([New_Key=Value|Rest_JSON_List1]),
53      json(JSON_List2),json(Merged),On_Conflict).
54
55  merge_json(json([Key=Value|Rest_JSON_List1]),json(JSON_List2),
56    json(Merged),On_Conflict) :-
57    % Key does not exist in JSON_List2
58    \+lookup(Key,JSON_List2,_),
59    merge_json(json(Rest_JSON_List1),json(JSON_List2),
60      json(Rest_Merged),On_Conflict),
61    Merged = [Key=Value|Rest_Merged].
62
63  merge_json(List1,List2,Merged_List,_On_Conflict) :-
64    is_list(List1), is_list(List2),
65    append(List1,List2,Merged_List).
```

B.7. merge_json/3

`merge_json(JSON1,JSON2,Result)` is a shortcut for the Prolog predicate `merge_json(JSON1,JSON2,Result,hard)`, that means the merging fails if both `JSON1` and `JSON1` have an object with an identical key but different value.

Listing B.7: Implementation of `merge_json/4`

```
1 merge_json(JSON1,JSON2,JSON) :-
2   merge_json(JSON1,JSON2,JSON,hard).
```

B.8. remove_at_from_property_names/2

The Prolog predicate `remove_at_from_property_names(JSON,Result)` removes the `@`-prefix of all keys within a JSON object whose own key was set to `properties`. Some example calls are listed in B.8.

Listing B.8: Example usage of `remove_at_from_property_names/2`

```
1  ?- JSON = json([
2    type=object,
3    properties=json([
4      % @-prefix is removed
5      '@foo'=json([
6        type=number
7      ])
8    ])
9  ]), xsd2json:remove_at_from_property_names(JSON,Result).
10 Result = json([
11   type=object,
12   properties=json([
13     foo=json([type=number])
14   ])
```

```
15 ]) .
16
17
18 ?- JSON = json([
19   type=object,
20   properties=json([
21     % can not remove @-prefix as there is
22     %   another 'foo' property
23     '@foo'=json([
24       type=number
25     ]),
26     foo=json([
27       type=string
28     ])
29   ])
30 ]), xsd2json:remove_at_from_property_names(JSON,Result).
31 Result = JSON . % only the order of the keys in
32                 %   in json([...]) might differ
33
34 ?- JSON = json([
35   definitions=json([
36     '@foo'=json([
37       type=number
38     ])
39   ])
40 ]), xsd2json:remove_at_from_property_names(JSON,Result).
41 Result = JSON . % only object keys within
42                 %   an object called 'properties'
43                 %   are inspected
```

Because of the fact, that only keys within an `properties` object should be renamed, the `remove_at_from_property_names/2` rules have to consider at least the first two levels of the given JSON Schema object and then traverse recursively through the whole tree. This results in a long definition as shown in Listing B.9.

Listing B.9: Implementation of `remove_at_from_property_names/2`

```
1  remove_at_from_property_names(json([]),json([])).
2
3  remove_at_from_property_names(json(List),JSON) :-
4    lookup(properties,List,
5      json(Properties),List_Without_Properties),
6    lookup(AtKey,Properties,AtKey_Value,Properties_Without_AtKey),
7    mark_attribute(Mark),
8    string_concat(Mark,Key_Str,AtKey),
9    (
10       lookup(Key_Str,Properties_Without_AtKey,
11         Key_Value,Properties_Without_AtKey_And_Key),
12       Key = Key_Str
13     ;
14       term_to_atom(Key,Key_Str),
15       lookup(Key,Properties_Without_AtKey,
16         Key_Value,Properties_Without_AtKey_And_Key)
17   ),
18   remove_at_from_property_names(
19     json(Properties_Without_AtKey_And_Key),
20     json(New_Properties_Without_AtKey_And_Key)
21   ),
22   New_Properties = [
23     AtKey=AtKey_Value,
24     Key=Key_Value
25     | New_Properties_Without_AtKey_And_Key
26   ],
```

```
27    remove_at_from_property_names(
28      json(List_Without_Properties),
29      json(New_List_Without_Properties)
30    ),
31    JSON = json([
32      properties=json(New_Properties)
33      | New_List_Without_Properties
34    ]).
35
36  remove_at_from_property_names(json(List),JSON) :-
37    lookup(properties,List,
38      json(Properties),List_Without_Properties),
39    lookup(AtKey,Properties,
40      AtKey_Value,Properties_Without_AtKey),
41    mark_attribute(Mark),
42    string_concat(Mark,Key,AtKey),
43    \+lookup(Key,Properties_Without_AtKey,
44      _Key_Value,_Properties_Without_AtKey_And_Key),
45    remove_at_from_property_names(
46      json(Properties_Without_AtKey),
47      json(New_Properties_Without_AtKey)
48    ),
49    New_Properties = [
50      Key=AtKey_Value
51      | New_Properties_Without_AtKey
52    ],
53    remove_at_from_property_names(
54      json(List_Without_Properties),
55      json(New_List_Without_Properties)
56    ),
57    JSON = json([
```

```
58      properties=json(New_Properties)
59      | New_List_Without_Properties
60    ]).
61
62  remove_at_from_property_names(json(List),JSON) :-
63    List = [Key=Value|Rest],
64    is_list(Value),
65    remove_at_from_property_names(json(Rest),json(New_Rest)),
66    JSON = json([Key=Value|New_Rest]).
67
68  remove_at_from_property_names(json(List),JSON) :-
69    List = [Key=Value|Rest],
70    Value \= json(_),
71    remove_at_from_property_names(json(Rest),json(New_Rest)),
72    JSON = json([Key=Value|New_Rest]).
73
74  remove_at_from_property_names(json(List),JSON) :-
75    lookup(properties,List,
76      json(Properties),List_Without_Properties),
77    \+((lookup(AtKey,Properties,
78      _AtKey_Value,_Properties_Without_AtKey),
79      mark_attribute(Mark),
80      string_concat(Mark,_Key_Str,AtKey))),
81    remove_at_from_property_names(
82      json(List_Without_Properties),
83      json(New_List_Without_Properties)
84    ),
85    (
86        Properties == [],
87        New_Properties = []
88      ;
```

```
89        Properties = [First=First_Value|Rest_Properties],
90        remove_at_from_property_names(
91          json(Rest_Properties),
92          json(New_Rest_Properties)
93        ),
94        (
95            First_Value = json(_),
96            remove_at_from_property_names(
97              First_Value,
98              New_First_Value
99            )
100         ;
101           First_Value \= json(_),
102           New_First_Value = First_Value
103       ),
104       New_Properties = [
105         First=New_First_Value
106         | New_Rest_Properties
107       ]
108   ),
109   JSON = json([
110     properties=json(New_Properties)
111     | New_List_Without_Properties
112   ]).
113
114 remove_at_from_property_names(json(List),JSON) :-
115   \+lookup(properties,List,
116     json(_Properties),_List_Without_Properties),
117   List = [Key=json(Value)|Rest],
118   remove_at_from_property_names(json(Rest),json(New_Rest)),
119   remove_at_from_property_names(json(Value),json(New_Value)),
```

```
120   JSON = json([Key=json(New_Value)|New_Rest]).
```

B.9. `is_required_property/2`

`is_required_property(MinOccurs,MaxOccurs)` checks if an `xs:element` within an `xs:sequence` or `xs:all` is *required*. This is the case if `MinOccurs` is at least `1`.

Listing B.10: Implementation of `is_required_property/2`

```
1 is_required_property('1',_).
```

C

Source Codes of CHR Rules

In this appendix there are several important source codes of some CHR rules. While the complete source code of the `xsd2json` module and its testing framework are available online [Nog13], we present here some instructional but complex translation rules.

C.1. Translation of `xs:attribute`

The `xs:attribute` node can occur within an `xs:complexType` node and is translated with respect to its attributes `name`, `type`, `use`, `fixed` and `default`.

Listing C.1: Implementation of the translation of `xs:attribute`

```
1 /**
2  * 'xs:complexType' which has an attribute with '@type'
```

```
3   *   attribute being set, i.e. '\+var(Type)'.
4   *
5   * Not supported attributes of this object as having no
6   *   result to the created JSON:
7   *   - form
8   *   - id
9   */
10 transform,
11     node(NS1,complexType,ComplexType_ID,
12       _ComplexType_Children,_ComplexType_Parent_ID),
13     node(NS2,attribute,Attribute_ID,
14       _Attribute_Children,ComplexType_ID),
15     node_attribute(Attribute_ID,name,Attribute_Name,_),
16     node_attribute(Attribute_ID,type,Type_With_NS,_),
17     node_attribute(Attribute_ID,use,Use,_),
18     node_attribute(Attribute_ID,fixed,Fixed,_),
19     node_attribute(Attribute_ID,default,Default,_)
20   ==>
21     \+var(Type_With_NS),
22     xsd_namespaces([NS1,NS2]),
23     reference_type(Type_With_NS,json(Attribute_JSON))
24   |
25     % check 'fixed' entity
26     (
27         var(Fixed),
28         Attribute_JSON2 = Attribute_JSON
29       ;
30         \+var(Fixed),
31         cast_by_json(Attribute_JSON,Fixed,Fixed_Casted),
32         Attribute_JSON2 = [enum=[Fixed_Casted]|Attribute_JSON]
33     ),
```

```
34      % check `default` entity
35      (
36          var(Default),
37          Attribute_JSON3 = Attribute_JSON2
38        ;
39          /**
40           * As mentioned in the XSD specification not both
41           *   `fixed` and `enum` can be set.
42           */
43          \+var(Default),
44          var(Fixed),      % see explanation above
45          cast_by_json(Attribute_JSON2,Default,Default_Casted),
46          Attribute_JSON3 = [
47            default=Default_Casted
48            |Attribute_JSON2
49          ]
50      ),
51      % generate JSON
52      mark_attribute(Mark),
53      string_concat(Mark,Attribute_Name,Attribute_Name2),
54      JSON1 = [
55        type=object,
56        properties=json([
57          Attribute_Name2=json(Attribute_JSON3)
58        ])
59      ],
60      % check `required` entity
61      (
62          Use == required,
63          JSON2 = [required=[Attribute_Name]|JSON1]
64        ;
```

```
65          Use \= required,
66          JSON2 = JSON1
67      ),
68      json(ComplexType_ID,json(JSON2)).
69
70
71  /**
72   * 'xs:complexType' which has an attribute with an
73   *   inline type definition, i.e. 'var(Type)' and an
74   *   'xs:simpleType' child node.
75   *
76   * This rule will be called once the JSON for the inner
77   *   'xs:simpleType' has been generated.
78   */
79  transform,
80      node(NS3,simpleType,SimpleType_ID,
81        _SimpleType_Children,Attribute_ID),
82      json(SimpleType_ID,json(SimpleType_JSON)),
83      node(NS1,complexType,ComplexType_ID,
84        _ComplexType_Children,_ComplexType_Parent_ID),
85      node(NS2,attribute,Attribute_ID,
86        _Attribute_Children,ComplexType_ID),
87      node_attribute(Attribute_ID,name,Attribute_Name,_),
88      node_attribute(Attribute_ID,type,Unbound_Type,_),
89      node_attribute(Attribute_ID,use,Use,_),
90      node_attribute(Attribute_ID,fixed,Fixed,_),
91      node_attribute(Attribute_ID,default,Default,_)
92    ==>
93      var(Unbound_Type),
94      xsd_namespaces([NS1,NS2,NS3])
95    |
```

```
96      Attribute_JSON = SimpleType_JSON,
97      % check `fixed` entity
98      (
99          var(Fixed),
100         Attribute_JSON2 = Attribute_JSON
101       ;
102         \+var(Fixed),
103         cast_by_json(Attribute_JSON,Fixed,Fixed_Casted),
104         Attribute_JSON2 = [enum=[Fixed_Casted]|Attribute_JSON]
105     ),
106     % check `default` entity
107     (
108         var(Default),
109         Attribute_JSON3 = Attribute_JSON2
110       ;
111         /**
112          * As mentioned in the XSD specification not both
113          *   `fixed` and `enum` can be set.
114          */
115         \+var(Default),
116         var(Fixed),      % see explanation above
117         cast_by_json(Attribute_JSON2,Default,Default_Casted),
118         Attribute_JSON3 = [
119           default=Default_Casted
120           |Attribute_JSON2
121         ]
122     ),
123     mark_attribute(Mark),
124     string_concat(Mark,Attribute_Name,Attribute_Name2),
125     JSON1 = [
126       type=object,
```

```
127       properties=json([
128         Attribute_Name2=json(Attribute_JSON3)
129       ])
130     ],
131     % check `required` entity
132     (
133         Use == required,
134         JSON2 = [required=[Attribute_Name]|JSON1]
135       ;
136         Use \= required,
137         JSON2 = JSON1
138     ),
139     json(ComplexType_ID,json(JSON2)).
140
141
142 /**
143  * `xs:complexType` which has an attribute with `@ref`
144  *   attribute being set.
145  *
146  * As specified, `@name` and `@type` can not be both
147  *   present.
148  */
149 transform,
150     node(NS1,complexType,ComplexType_ID,
151       _ComplexType_Children,_ComplexType_Parent_ID),
152     node(NS2,attribute,Attribute_ID,
153       _Attribute_Children,ComplexType_ID),
154     node_attribute(Attribute_ID,ref,Ref,_),
155     node_attribute(Attribute_ID,use,Use,_),
156     node_attribute(Attribute_ID,fixed,_Fixed,_),
157     node_attribute(Attribute_ID,default,_Default,_)
```

```
158   ==>
159     xsd_namespaces([NS1,NS2])
160   |
161     string_concat('#/definitions/@',Ref,Definition_Ref),
162     Attribute_JSON = [
163       '$ref'=Definition_Ref
164     ],
165     JSON1 = [
166       type=object,
167       properties=json([
168         Ref=json(Attribute_JSON)
169       ])
170     ],
171     % check `required` entity
172     (
173         Use == required,
174         JSON2 = [required=[_Attribute_Name]|JSON1]
175       ;
176         Use \= required,
177         JSON2 = JSON1
178     ),
179     json(ComplexType_ID,json(JSON2)).
```

List of Tables

4.1. Translation of simple XSD data types 37

4.2. Translation of constraining facets 39

4.3. Translation of nested XSD elements 44

Bibliography

[Arc12] ARCHIVE, W3C N.: *Two XML Schema Specifications are Recommendations*. http://www.w3.org/News/2012#entry-9412. Version: April 2012

[BFRW01] BROWN, Allen ; FUCHS, Matthew ; ROBIE, Jonathan ; WADLER, Philip: *XML Schema: Formal Description*. World Wide Web Consortium, Working Draft WD-xmlschema-formal-20010925, September 2001

[Bir04] BIRON, A. Malhotra P.: XML Schema Part 2: Datatypes Second Edition / W3C Recommendation. 2004. – Forschungsbericht

[bra06] BRAY, Tim (Hrsg.) ; PAOLI, Jean (Hrsg.) ; SPERBERG-MCQUEEN, C.M. (Hrsg.) ; MALER, Eve (Hrsg.) ; YERGEAU, François (Hrsg.) ; COWAN, John (Hrsg.). W3C - WORLD WIDE WEB CONSORTIUM: Extensible Markup Language (XML) 1.1 (Second Edition) / W3C - World Wide Web Consortium. Version: Second, September 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/. W3C - World Wide Web Consortium, September 2006. – W3C Recommendation

[Cla97] CLARK, James: Comparison of SGML and XML. In: *World Wide Web Consortium Note* 15 (1997)

[CM84] CLOCKSIN, William F. ; MELLISH, Christopher S.: Programming in PROLOG. (1984)

[Cov05] COVER, Robin: *XML Applications and Initiatives*. http://xml.coverpages.org/xmlApplications.html. Version: Juni 2005

[Cro06] CROCKFORD, D.: *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). http://www.ietf.org/rfc/rfc4627.txt. Version: July 2006 (Request for Comments)

[Duv11] DUVANDER, Adam: *1 in 5 APIs Say "Bye XML"*. http://blog.programmableweb.com/2011/05/25/1-in-5-apis-say-bye-xml/. Version: Mai 2011

[ECM99] ECMA: *ECMAScript Language Specification*. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf. Version: December 1999. – ECMA Standard 262, 3rd Edition

[Frü95] FRÜHWIRTH, Thom: *Constraint handling rules*. Springer, 1995

[Gal13] GALIEGUE, Francis: *JSON Schema: interactive and non interactive validation*. http://tools.ietf.org/html/draft-fge-json-schema-validation-00. Version: Februar 2013 (Internet-Draft)

[GSMT+08] GAO, Shudi ; SPERBERG-MCQUEEN, C. M. ; THOMPSON, Henry S. ; MENDELSOHN, Noah ; BEECH, David ; MALONEY, Murray: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620, June 2008

[JSO11] *Roadmap for the JSON Schema spec - JSON Schema Google Group*. https://groups.google.com/forum/#!msg/json-schema/JSfq2xPnlsA/24MU4zbN2BMJ. Version: Februar 2011

[JSO13a] *Introducing JSON*. http://www.json.org/. Version: November 2013

[JSO13b] *JSON Schema Core/Validation Meta Schema*. http://json-schema.org/schema. Version: Dezember 2013

[JSO13c] *JSON Schema Software*. http://json-schema.org/implementations.html. Version: Oktober 2013

[KZ09] K. ZYP, Ed.: *A JSON Media Type for Describing the Structure and Meaning of JSON Documents (Draft 01)*. http://tools.ietf.org/html/draft-zyp-json-schema-01. Version: Dezember 2009 (Internet-Draft)

[KZ13] K. ZYP, Ed.: *A JSON Media Type for Describing the Structure and Meaning of JSON Documents (Draft 04)*. http://tools.ietf.org/html/draft-zyp-json-schema-04. Version: Januar 2013 (Internet-Draft)

[Les13] LESTER, Andy: *Documentation for the TAP format*. https://metacpan.org/pod/release/PETDANCE/Test-Harness-2.64/lib/Test/Harness/TAP.pod. Version: Dezember 2013

[Mad13] MADSEN, Andreas: *npmjs: Interpreted*. https://npmjs.org/package/interpreted. Version: Dezember 2013

[Nog13] NOGATZ, Falco: *Github: Source Code of xsd2json*. https://github.com/fnogatz/xsd2json. Version: Dezember 2013

[NPRI09] NURSEITOV, Nurzhan ; PAULSON, Michael ; REYNOLDS, Randall ; IZURIETA, Clemente: Comparison of JSON and XML Data Interchange Formats: A Case Study. In: *CAINE* 2009 (2009), S. 157–162

[PGM+08] PETERSON, David ; GAO, Shudi ; MALHOTRA, Ashok ; SPERBERG-MCQUEEN, C. M. ; THOMPSON, Henry S.: *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. World Wide Web Consortium, Working Draft WD-xmlschema11-2-20080620, June 2008

[W3C04] W3C: *XML Schema Part 1: Structures Second Edition ; W3C Recommendation 28 October 2004*. http://www.w3.org/TR/xmlschema-1/. Version: 2004

[Wie05] WIELEMAKER, Jan: SWI-Prolog SGML/XML Parser. In: *SWI, University of Amsterdam, Roetersstraat* 15 (2005), S. 1018

[Wie13a] WIELEMAKER, Jan: *load_structure/3 - SWI-Prolog Manual*. http://www.swi-prolog.org/pldoc/man?predicate=load_structure/3. Version: Juli 2013

[Wie13b] WIELEMAKER, Jan: *Supporting JSON - SWI-Prolog Manual*. `http://www.swi-prolog.org/pldoc/doc_for?object=section(2,'5',swi('/doc/packages/http.html'))`. Version: Dezember 2013

Name: Falco Nogatz Matrikelnummer: 718320

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Falco Nogatz