Extending JSON-LD Framing Capabilities

Kosa Nenadić[1], Milan Gavrić[2], Imre Lendak[2]

[1]Schneider Electric DMS NS LLC Novi Sad, Narodnog Fronta 25 a,b,c,d, 21000 Novi Sad, Serbia; kosa.nenadic@schneider-electric-dms.com

[2]Faculty of Technical Sciences, University of Novi Sad, Trg D. Obradovića 6, 21000 Novi Sad, Serbia; gavricm@uns.ac.rs, lendak@uns.ac.rs

Abstract: Today, with the increasing popularity of JSON-LD on the Web, there is a need for transformation and extraction of such structured data. In this paper, the authors propose extensions of the JSON-LD Framing specification which are able to create a tree layout based on recursive application of prioritized inverse relationships defined in a frame. The extensions include recursive application of reverse framing, a new @priority keyword which prioritizes reverse properties, a new embedding rule defined with the @first keyword, and the new @reverseRoots keyword used for filtering the result hierarchies of full-length. The proposed Extended Framing Algorithm, together with an extended frame, can be applied on arbitrary JSON-LD input files regardless of the length of its reverse hierarchy chains present in the frame. The proposed solution was tested on JSON-LD documents containing the ENTSO-E CIM Profiles. The two test scenarios were selected because of their complexity and size, each of them containing the ENTSO-E CIM Profiles expressed in CIM RDF Schema and OWL 2 Schema, respectively.

Keywords: Common Information Model; ENTSO-E; Framing; JSON-LD; RDF; Semantic Web

1 Introduction

The Semantic Web represents the Web of Linked Data. With the growth of the Semantic Web, the World Wide Web Consortium (W3C) promoted common data formats and exchange protocols, including the Resource Description Framework (RDF) family of specifications based on the RDF data model. JavaScript Object Notation (JSON) is considered the de-facto standard for data exchange over the Internet, mainly due to its simplicity for developers and its consumption in mobile and web applications [1]. Although JSON syntax is simple and clear there is no associated semantics. In contrast, JSON-LD (i.e. a JSON based serialization for Linked Data) adds meaning to JSON documents. A JSON-LD document is an instance of an RDF data model. The data model of a JSON-LD document represents a labeled, directed graph. A single directed graph can be serialized in

multiple ways and expressing the same information [2]. In practice, many complex models defined as graphs need to be presented in a hierarchical way to allow convenient access to particular data [3]. A hierarchy is commonly defined with a tree data model. For example, various ontology editors such as OntoStudio, TopBraid Composer, Protégé, Web Protégé, ontology browsers such as VectorBase, and ontology libraries such as OntoLink use indented tree visualization to present hierarchical structures associated with ontology entities [4], [5]. In paper [6] various ontology visualization methods were analyzed, including graph and tree structures, and a new approach for an ontology visualization and evaluation based on descriptive vectors is presented. As a JSON native data model is a tree [7], there is a need to transform a JSON-LD graph into a JSON tree that is capable of modeling cyclic structure. For that purpose, a mapping mechanism is defined in the form of JSON-LD Framing [8] – a draft specification published by the Linking Data in JSON Community Group with the aim to query and create specific tree layouts of JSON-LD documents. In this specification, a frame is applied on an input graph defined in a JSON-LD document using the JSON-LD Framing Algorithm, shaping the input document into a tree layout result that satisfies conditions specified in the frame. The JSON-LD Framing Algorithm complements the JSON-LD 1.0 Processing Algorithms and API [9], which defines a set of algorithms (namely, expansion, compaction, flattening and RDF serialization/deserialization) for programmatic transformations of JSON-LD documents.

There is often a need to express data relationships defined in a graph in a reverse direction. For example, a common practice when expressing a parent-child relationship between two entities in a class hierarchy definition is that the relationship is declared from a child to a parent. The aim of the @reverse keyword in JSON-LD is to express an inverse relationship using a reverse property [10].

Currently, the JSON-LD Framing specification supports a basic reverse framing, meaning that each inverse relationship has to be explicitly declared in a frame and its sub-frames in order to be present in the result. In this paper, the authors propose the introduction of three new framing keywords, namely @priority, @first and @reverseRoots, and present the usage of the following extensions of the JSON-LD Framing Algorithm:

- recursive application of reverse framing,
- prioritized inverse relationships using the @priority keyword in a reverse property,
- node object embedding using the @first keyword as a value of the object embed flag (i.e. @embed),
- node object filtering based on the value of the @reverseRoots keyword.

The ultimate goal of the proposed extensions is to define simple frames which result in desired tree layouts.

The paper is organized into sections. Section 2 presents related work, while Section 3 defines the problem. The proposed extensions are described in Section 4 together with the pseudo-code of the algorithm. Section 5 contains the description of the testing methodology and the overview of two test scenarios. The analysis of the input and output data, the results of performance testing and the discussion are presented in Section 6. Additionally, Section 6 demonstrates the application of resulting framed outputs in a custom developed JSON tree viewer for Web-based content visualization.

2 Related Works

The recent advances in the business intelligence applications field and knowledge representation is paired with Semantic Web technology improvements. For instance, numerous new versions of W3C Web Recommendations for RDF were created or updated in the last three years [11]. In the same direction were efforts to convert data from a JSON format to a semantically-enriched, RDF format containing Linked Data URIs [12]. Similarly, a framework for integration of various governmental services was developed using semantically enhanced service descriptions [13]. Nearly at the same time, the W3C created JSON-LD [10] as a lightweight syntax to serialize Linked Data in JSON. JSON-LD gained significant popularity and worldwide adoption when the major search engines (Google, Microsoft, Yahoo! and Yandex) created schema.org – a structured data markup schema – with the aim to enable search engines to understand the information embedded in web pages in order to serve richer search results [14], [15]. In reference [1] JSON-LD is recognized as a format that increases the utility and applications of the Smart City's data.

The JSON-LD Implementation Report [16] gives an overview of JSON-LD implementation statuses in various programming languages. Most of the available libraries that implement JSON-LD, and the JSON-LD Processing Algorithms and API, also support framing in its current state. As the JSON-LD Framing specification is a work in progress, version 1.0 lacks support for several features such as reverse framing, named graphs, re-embedding of the same data in the result [17], etc. Work was recently reactivated on the JSON-LD Framing specification 1.1 [8], focusing on changes regarding the node object embed options, support for the basic reverse framing, and framing of named graphs. The JSON-LD Framing 1.0 considers embedding of node objects as enabled (i.e. *true*) or disabled (i.e. *false*), but the latest version 1.1 also adds the following object embed flags @always, @never, @last and @link as the alternatives.

JSON-LD Framing is applied in specifications on the W3C recommendation track, such as Web Payments HTTP Messages 1.0 (uses JSON-LD Frame and JSON Schema for conformance check of JSON-LD objects of web payment messages

[18]) and Web Annotation Vocabulary (defines JSON-LD frames applicable to the graph of information that generate JSON output conforming to the serialization recommended by the Web Annotation Data Model [19]). In reference [20], a high performance cache for materialized graph views over large RDF graphs is created using JSON-LD Framing for denormalization of the materialized views into a tree data structure that is further indexed into a high performance tree indexing system. A similar approach is taken in [21] where JSON-LD Framing is used to represent RDF graphs of bibliographic data in JSON suitable for indexing with Elasticsearch.

In reference [22] the authors recognized the need for recursive prioritized inverse relationships in frames and developed an early implementation on top of node object filtering, as a basis of a web component which processes and displays machine readable CIM Profiles in a more human friendly form. This paper extends those results and addresses recursive application of prioritized reverse framing paired with node object filtering and introduces additional framing keywords to achieve more flexible results.

3 Problem Definition

In this section, we identify some deficiencies of the existing reverse framing solution, namely:

1. The inability to define a simple frame used for reverse framing.
2. The lack of multiple-relationship based reverse framing in a simple frame.
3. It is not possible to embed node objects on their first occurrence.
4. The lack of advanced filtering used for reverse framing.

3.1 Problem 1: Reverse Framing Complexity

Due to the fact that JSON-LD serializes directed graphs, each property (i.e. directed edge) points from one node to another node or value. Sometimes, it is necessary to express such a relationship in reverse direction. For instance, considering a *subClassOf* relationship, pointing from a subclass to a superclass, explicit definition of a property that designates the opposite direction – *superClassOf* relationship is typically omitted. Such a relationship can be expressed in a JSON-LD graph when the *@reverse subClassOf* property is defined. In this way, a reverse hierarchy is created.

When a long reverse hierarchy chain is needed, creating a desired framed output requires a deeply nested, complex frame, as the JSON-LD Framing specification currently supports only basic reverse framing. This means that a main frame and each embedded frame should define their own @reverse keyword section which may be inconvenient in situations when a creator of a frame is not aware of hierarchy depth in a JSON-LD input. A typical example represents the inheritance hierarchy (i.e. *rdfs:subClassOf*) in any RDF Schema (RDFS) vocabulary or Web Ontology Language (OWL) ontology.

An example of a JSON-LD frame that produces a tree hierarchy based on class inheritance and groups all related class properties based on a property domain is shown in Listing 1. It can be noticed that a subframe assigned to the *children* property must be repeated as many times as needed, depending on a JSON-LD document to be framed.

```
{
    "@context": {  ...
        "children": { "@reverse": "rdfs:subClassOf", "@container": "@set" },
        "properties": { "@reverse": "rdfs:domain", "@container": "@set" }
    },
    "@type": "rdfs:Class",
    "children": {
        "@type": "rdfs:Class",
        "properties": {
            "@type": "rdf:Property"
        },
        "children": {
            "@type": "rdfs:Class",
            "properties": {
                "@type": "rdf:Property"
            },
            "children": { ... }
        }
    }
}
```

Listing 1

Example of JSON-LD Frame Excerpt with Reverse Properties

There is certainly an option to create a custom suited reverse frame programmatically based on a document to be framed and the relationships of interest, but then each document requires its own frame and reverse framing becomes a two-step process. In the first step, a custom program creates a frame based on an input file and inverse relationships of interest, while in the second the frame is applied on the input file to produce a framed output.

3.2 Problem 2: Reverse Framing with Multiple Relationships

Frames with reverse properties become quite complex when additional properties are included or when there are multiple nested reverse properties of the same type creating corresponding hierarchies. For example, a frame with multiple nested reverse properties is shown in Listing 2. Assuming that family members work in the same company this frame can be used to create two hierarchies (company employee and parent-child hierarchies) starting from a person.

```
{
    "@context": { ...
        "employees": { "@reverse": "ex:employeeOf" },
        "children": { "@reverse": "ex:childOf" }
    },
    "@type": "Person",
    "employees": {
        "@type": "Person",
        "employees": {
            "@type": "Person",
            "employees": { ... },
            "children": { ... }
        },
        "children": {
            "@type": "Person",
            "employees": { ... },
            "children": { ... }
        }
    },
    "children": {
        "@type": "Person",
        "employees": { ... },
        "children": { ... },
    }
}
```

Listing 2

Example of JSON-LD Frame Excerpt with Multiple Nested Reverse Properties

Currently, there is no straightforward and standardized way for creating single-relationship or multiple-relationship based reverse tree hierarchies (thereafter referred to as tree hierarchies) of arbitrary depth starting from a flattened JSON-LD document and simple frame.

The order in which each inverse relationship is applied is important since JSON-LD framing uses the depth-first search algorithm when traversing related nodes to produce a framed output. By default, reverse properties are applied in order determined by the Expansion Algorithm, as an expanded frame is one of the inputs of the Framing Algorithm. Depending on how reverse properties are given in the original frame, whether using plain reverse properties or reverse properties with expanded term definitions or their combination, their order may vary in an expanded frame. The order in which inverse relationships are applied should be unambiguously determined and easily understood regardless of the way reverse properties are declared.

3.3 Problem 3: Embed on First Occurrence

Node objects are embedded on their last occurrence (i.e. the embedding rule ”@embed”: ”@last”) unless explicitly defined otherwise in a frame. Consequently, the result of reverse framing may not contain an explicitly expressed chain of full-length (i.e. longest hierarchy) if some element in the reverse chain appears later in the result. In order to overcome this deficiency, the embedding rule of @always could be applied, but then each referenced node and its subtree would be repeated in the result.

3.4 Problem 4: Advanced Filtering

The Framing Algorithm supports filtering based on strict-typing and duck-typing [17], [23]. The first type of filtering uses values of @type for matching. In its absence, filtering of nodes is based on matching included properties. Reverse framing is performed in conjunction with filtering, meaning that the framing process results in an array of node objects that satisfy filtering conditions and may also be roots of a reverse tree hierarchy if a node is related to some other nodes with reverse properties given in a frame. Some of these node objects may already be subtree roots in other reverse tree hierarchies, so there is a need to filter them out. In this way, a resulting framed output would contain only reverse tree hierarchies of full-length.

4 Solution

In order to address the identified problems, the authors propose extensions to the JSON-LD frame definition and the JSON-LD Framing Algorithm.

4.1 The Extended Frame

4.1.1 Recursive Application of Reverse Properties

By this proposal each reverse property is defined in the top-level frame of a JSON-LD frame. Each reverse property is applied recursively, meaning that they are all implicitly passed to any subframe (i.e. using the similar approach applied for object embed flags, the explicit inclusion flag and the require all flag). At the same time, a reverse property can be overridden in a subframe and as such passed to its subframes. In this way, deeply nested reverse properties are avoided in a frame.

4.1.2 Definition of Prioritized Reverse Properties

Problem 2 described in the previous section, regarding the order in which inverse relationships are applied, can be overcome if the order was explicitly declared. For this reason, a frame definition of a reverse property is extended with a new @priority framing keyword (e.g. *children* in Listing 3). The priority of a reverse property is defined with a number value of the @priority keyword (a lower value a higher priority). If the priority is not defined for a reverse property, a default priority is determined by the Expansion Algorithm. A priority does not determine the order in which relationships appear in a resulting tree layout because the layout is compacted using a context included in a frame.

4.1.3 Definition of a New Embedding Rule - First

The authors consider the embedding of node objects on their first occurrence in a JSON-LD graph as equally important to other alternatives. When used with recursive reverse framing it allows for the creation of tree hierarchies of full-length, while leaving node references to already traversed nodes. Therefore, a new value of the @embed framing keyword is proposed, defined with the @first object embed flag. Listing 3 illustrates how the new flag is used in a frame to globally define that node objects are embedded on their first occurrence.

4.1.4 Definition of Hierarchy Roots

In order to keep only the tree hierarchies of full-length in resulting framed outputs the authors introduce the @reverseRoots framing keyword (Listing 3). This keyword acts as a flag. The value of the @reverseRoots keyword is boolean. Setting its value to *true* enables filtering. If @reverseRoots is not specified, its value defaults to *false*.

```
{
    "@context" : {
        "ex" : "http://example.com/",
        "employees" : {"@reverse" : "ex:employeeOf"},
        "children" : {"@reverse" : "ex:childOf"}
    },
    "@type": "ex:Person",
    "@embed": "@first",
    "@reverseRoots": true,
    "employees": {"@priority" : 1},
    "children": {"@priority" : 2}
}
```

Listing 3
Example of Extended JSON-LD Frame

4.2 The Extended Framing Algorithm

The Extended Framing Algorithm (EFA) represents an extension of the Framing Algorithm [8] which supports the proposed, extended frame definition. In addition to the existing framing capabilities, it creates a tree hierarchy on each filtered node based on multiple prioritized reverse properties provided in an input frame.

The very process of creating prioritized reverse tree hierarchies in a JSON-LD tree layout can be split into two portions. The first portion of the EFA (Listing 4) accepts an expanded JSON-LD input file (i.e. *graph*) and expanded frame (i.e. *frame*) together with the global framing options. Essentially, this portion of the overall algorithm initializes the parameters used in the second, recursive portion. It includes:

- Initialization of the current state.
- Flattening of the input graph.
- Identification of relationships to be inverted from the input frame.
- Identification of graph nodes related with the identified relationships.

- Identification of all hierarchy roots and non-blank roots of each identified relationship based on the related nodes, and initialization of the current state.

Identification of non-blank roots is important as it is expected by this implementation that all blank (i.e. anonymous) nodes are embedded inside non-blank (i.e. named) nodes in a created hierarchy. For this reason, based on the identified hierarchy roots, if a root is a blank node, then its hierarchy is searched downstream for the nearest appearance of non-blank descendants which become new hierarchy roots.

```
function Frame(graph, frame, options)
        state       = CreateState(options)
        fGraph      = Flatten(graph)
        revRels     = GetReverseRelationships(frame)
        forEach node in fGraph do
                forEach revRel in revRels do
                        if node has revRel then
                                add node.id into revRel.domain
                                add node.revRel.value into revRel.range
        forEach revRel in revRels do
                forEach id in revRel.range do
                        if not id exists in revRel.domain then
                                add id into revRel.roots
        forEach revRel in revRels do
                forEach id in revRel.roots do
                        if IsBlank(id) then
                                descs = FindNearestNBDescs(id, revRel)
                                add descs into revRel.nonBlankRoots
                        else
                                add id into revRel.nonBlankRoots
        state.revRels       = revRels
        state.subjects      = fGraph.nodes
    return RecurFrame(state, IDs(fGraph.nodes), frame, false, undefined)
end function
```

Listing 4

Extended Frame Algorithm – First Portion

The second, recursive portion of the EFA (pseudo-code in Listing 5) includes:

- Flag initialization using the current frame and state – flags ensure that a property (namely *embed*, *explicit*, *requireAll*, *reverse* and *reverseRoots*) is passed from the current frame to a subframe if the subframe does not override it;
- Filtering of subjects that satisfy the current frame and flags (i.e. matches);
- Prioritization of matches using the identified non-blank roots – meaning that non-blank root matches have precedence over the rest of the matches that are sorted ascending by their ids;
- Each match is processed in the following way:
    - A match is skipped if it is a top-level node that is already traversed in another reverse hierarchy and only the hierarchies of full-length are of interest;
    - Depending on the current *embed* value and state, the way in which the match is referenced in the output is determined or the framing process is continued;

- Based on the content of the current frame, inverse relationships are identified, ordered by their priorities and used to build a tree hierarchy with the match as its root. For each relationship, the match's related nodes are prioritized and traversed recursively with the appropriate subframe, taking care that the related node is skipped if it is already traversed and hierarchies of full-length are of interest. The match and related node are marked as traversed when they are recursively processed;
- The match's own properties are processed;
- Default properties, defined in the current frame, are processed;
- The output is set as a value of the current *parent's property*;
- If the recursive framing of a top-level node is completed and hierarchies of full-length are of interest, then all traversed nodes are globally stored to be checked when a new top-level node is processed.

```
function RecurFrame(state, subjects, frame, parent, property)
        flags      = GetFlags(frame, state)
        matches    = FilterSubjects(state, subjects, frame, flags)
        matches    = PrioritizeNonBlankRoots(matches, state, frame)

        forEach match in matches do
                if property == undefined and flags.reverseRoots and
                match in state.traversedAll then
                        continue

                output = create(match)
                if ProcessEmbedValues(flags.embed, state, output) then
                        continue

                revRels = GetReverseRelationships(frame)
                revRels = OrderByPriority(revRels)

                forEach revRel in revRels do
                        rs = GetRelated(id, revRel)
                        rs = PrioritizeNonBlankRoots(rs, state, frame)
                        implicitFrame      = CreateImplicitFrame(flags)
                        subframe   = GetSubframe(frame, revRel)
                        subframe   = MergeFrames(implicitFrame, subframe)

                        forEach r in rs do
                                if subframe.reverseRoots and
                                r in state.traversed then
                                        continue

                                RecurFrame(state, r, subframe, output.reverse, revRel)

                                if not id in state.traversed then
                                        add id into state.traversed
                                if not r in state.traversed then
                                        add r into state.traversed
                ProcessOwnProperties(match, flags, frame, output)
                ProcessDefaultProperties(frame, output)
                AddFrameOutput(parent, property, output)

                if property == undefined and flags.reverseRoots then
                        add state.traversed into state.traversedAll
end function
```

Listing 5

Recursive Portion of the Extended Frame Algorithm

5 Testing Methodology

Initial testing was conducted against the set of created new reverse API tests included in the JSON-LD Test Suite provided with the implementation of the Extended Framing Algorithm [24]. These tests basically validate a framed output against the expected output for a given input and frame.

For the detailed testing, the authors searched for convenient data sources that are sufficiently large and complex to evaluate the proposed extensions providing at the same time verifiable results. The CIM Profiles which are part of the CGMES defined by the European Network Transmission System Operators for Electricity [14] (ENTSO-E) were chosen as the testing data source. These profiles were used for the 5th interoperability tests conducted by the European Transmission System Operators (TSO) in 2014.

In order to clarify the connections between input and output data, the following terms are defined:

- CIM Profile – a subset of CIM classes, properties and associations including CIM extensions. It may be defined using the CIM RDF Schema [25].
- CIM RDF Schema – an IEC standard, which relies on the subset of RDF classes and properties and set of CIM RDF Schema extensions [25].
- RDF/XML – an XML syntax for RDF graphs.
- CIMXML model exchange format – an IEC standard, defines a CIM Profile serialization using the RDF/XML [26].

CIMXMLs of the ENTSO-E CIM Profiles were used as a starting data source in two test scenarios. In the first test scenario, the profiles were transformed into JSON-LD syntax and used as a testing input. As a CIM Profile does not contain blank nodes related with RDFS properties, it was decided to conduct additional testing using the representation of CIM Profiles in a more expressive OWL 2 (the latest version of OWL). For this reason, the profiles were mapped into the OWL 2 representation in RDF/XML syntax, transformed into JSON-LD syntax afterwards, and as such used as a testing input in the second test scenario. In both test scenarios, the same input frame is applied to create a CIM Profile tree hierarchy.

5.1 The RDFS Test Scenario

In this scenario, the CIMXML files containing the RDFS representation of CIM Profiles were used as a starting data source. Those files were converted into JSON-LD syntax since both RDF/XML and JSON-LD are capable to serialize an RDF graph. The translation was done using the RDF Translator [27]. The frame

shown in Listing 6 was used together with a translated CIM Profile as an input to the Extended Framing Algorithm.

5.2 The OWL 2 Test Scenario

Based on the authors' previous experiences (an analysis of CIM Profile conversion into OWL was presented in reference [28]), a custom converter was implemented in order to transform CIMXML of CIM Profiles into the OWL 2 format. The conversion was accomplished in the following steps:

- RDFS class and property constructs were transformed into corresponding OWL 2 class and property constructs (i.e. *rdfs:Class* into *owl:Class*; *rdfs:Property* into *owl:DatatypeProperty* or *owl:ObjectProperty* depending on a relation designated by *rdfs:Property*).
- The RDFS extensions (i.e. constructs that share *cims* namespace) were transformed into corresponding OWL 2 constructs where possible. *cims:multiplicity* was replaced with OWL object and data property restrictions, *cims:inverseRoleName* was mapped to *owl:inverseOf*, and *cims:dataType* was replaced with *rdfs:range* of an *owl:DatatypeProperty*.
- The rest of the RDFS extensions (namely, *cims:AssociationUsed*, *cims:stereotype*, *cims:isFixed*, *cims:ClassCategory* and *cims:belongsToCategory*) were preserved as meta data of defined classes and properties.
- Classes that model primitive datatypes, such as String, Date, Integer, etc., were skipped and corresponding data types from XML Schema Definition (XSD) namespace were used instead.

In addition to the subset of RDF properties applied in CIM RDFS, the authors used *rdfs:isDefinedBy* property to designate that each defined *owl:Class*, *owl:DatatypeProperty* and *owl:ObjectProperty* is defined by the created *owl:Ontology*. In this way, one more hierarchical level was created in the CIM profile ontology compared to the corresponding profile RDF Schema.

The created CIM Profiles in OWL2 form were validated in Protégé ontology editor (Figure 1). JSON-LD serialization of a CIM Profile is used as an input in the Extended Framing Algorithm together with the frame shown in Listing 6 (see 5.3).

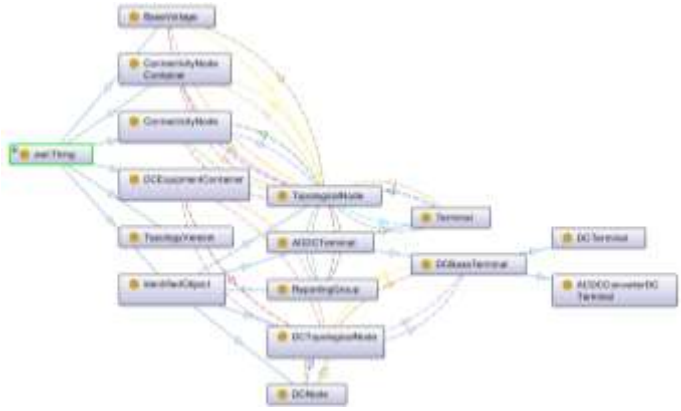

Figure 1
OntoGraph Vizualization of Topology Profile Ontology in Protégé

5.3 Test Frame

The input frame (Listing 6) shapes the initially provided JSON-LD document into hierarchy trees starting from an ontology or class, groups related classes and properties, embeds subclasses based on their inheritance relationship, groups all properties that belong to a class. It ensures that a hierarchy tree is not a subtree of another tree that only explicitly declared properties are included in the output and that node objects are embedded when they are first encountered. The same resulting framed output can be achieved by creating a simpler frame in each test scenario. For instance, in the RDFS test scenario OWL constructs and inverse *rdfs:isDefinedBy* property can be avoided in the frame. However, the authors wanted to keep the same input frame not affecting the framing process. At the same time, the results of such framing served as a confirmation of properly implemented profile conversion.

```
{
    "@context": {
        // owl, rdf, rdfs, xsd, cim, entsoe, tp, ...
        "children": { "@reverse": "rdfs:subClassOf", "@container": "@set"},
        "properties": { "@reverse": "rdfs:domain", "@container": "@set"},
        "defines": { "@reverse": "rdfs:isDefinedBy", "@container": "@set"}
    },
    "@type": ["owl:Ontology", "rdfs:Class", "owl:Class"],
    "@embed": "@first",
    "@reverseRoots": true,
    "@explicit": true,
    "defines": {
        "@priority": 1,
        "@type": ["owl:Class", "owl:DatatypeProperty", "owl:ObjectProperty"]},
    "children": {
        "@priority": 2,
        "@type": ["rdfs:Class", "owl:Class"]},
    "properties": {
        "@priority": 3,
        "@type": ["rdf:Property", "owl:ObjectProperty", "owl:DatatypeProperty"]}
}
```

Listing 6
Frame for CIM Profiles

6 Results and Discussion

The performance testing of the Extended Framing Algorithm implementation based on forked version of jsonld.js, available at [24], was conducted on a computer with an Intel Core i5-4300M/2.60 GHz CPU with 16 GB of RAM and 500 GB HDD running Microsoft Windows 8.1 Enterprise (64-bit) with Node v6.8.1. The testing was done in two scenarios, where 1000 iterations of the framing were executed for each file, and an average time was calculated.

Tables 1-2 present model metrics of the input and output data, and average framing times in RDFS and OWL 2 test scenarios, respectively.

Table 1
Data ModelMetrics per ENTSO-E CIM Profile document in RDFS

Profiles	Input				Output				
	#Classes	#Properties	#Triples	Size [bytes]	#Triples	Size [bytes]	#Hierarchy trees	Longest hierarchy [length]	Average framing time [ms]
GeographicalLocation	10	23	240	25332	58	5015	6	2	6.981
TopologyBoundary	10	28	284	32533	67	5680	7	2	8.075
Topology	18	31	355	36790	88	7085	8	3	10.182
DiagramLayout	21	46	585	58819	118	8789	14	3	15.769
EquipmentBoundary	25	39	567	59392	116	9517	10	5	15.583
StateVariablesProfile	33	63	770	74973	166	11775	24	3	21.362
SteadyStateHypothesis	75	84	1232	125892	292	24728	24	7	35.453
EquipmentProfileCore	177	412	4483	444497	1107	85889	69	7	171.210
EquipmentProfileCore-ShortCircuit	183	399	4417	448550	1093	86127	69	7	154.307
EquipmentProfileCore-Operation	222	417	4799	480497	1207	90106	69	7	187.640
EquipmentProfileCore-ShortCircuitOperation	226	624	6451	633842	1629	126640	69	7	281.026
Dynamics	252	2802	21655	2057966	6067	440244	39	7	1233.987

In order to have better understanding of results, input and output files of both test scenarios are compared and analyzed. The OWL 2 representation of profiles has a slightly smaller number of defined classes due to usage of XSD primitive data types when compared with RDFS representation, while the number of properties is the same. A conversion of CIM Profile representation in RDFS into OWL 2 had a significant impact on number of triples in OWL 2 profiles as some RDFS extensions are represented with several triples in OWL 2 (e.g. *cims:multiplicity* into OWL 2 restrictions). The number of triples in OWL 2 representation is increased for ~48.9% on average in comparison with the RDFS representation, while the size of the input file is increased for ~24.9% on average.

Table 2
Data Model Metrics per ENTSO-E CIM Profile document in OWL 2

Profiles	Input				Output				
	#Classes	#Properties	#Triples	Size [bytes]	#Triples	Size [bytes]	#Hierarchy trees	Longest hierarchy [length]	Average framing time [ms]
GeographicalLocation	7	23	334	30019	59	5635	1	3	8.854
TopologyBoundary	7	28	404	36236	69	6389	1	3	10.512
Topology	16	31	500	43085	93	8100	1	4	13.612
DiagramLayout	16	46	867	72747	123	9987	1	4	22.829
EquipmentBoundary	20	39	821	69265	119	10738	1	6	22.477
StateVariablesProfile	29	63	1158	95947	183	13744	1	4	32.877
SteadyStateHypothesis	70	84	1795	155311	307	28066	1	8	54.195
EquipmentProfileCore	169	412	6830	574211	1161	98731	1	8	311.856
EquipmentProfileCore-ShortCircuit	175	399	6745	584335	1143	99826	1	8	273.016
EquipmentProfileCore-Operation	214	417	7235	617918	1261	104570	1	8	343.537
EquipmentProfileCore-ShortCircuitOperation	218	624	9882	854579	1683	148120	1	8	522.892
Dynamics	247	2802	36035	2833795	6097	509551	1	8	2441.795

The number of output triples in OWL 2 representation is increased for ~4.2% on average when compared with its counterpart in RDFS representation, while the size of an output file is increased for ~14.6% on average. This is the consequence of the introduced *rdfs:isDefinedBy* property and conversion of *rdfs:Property* into *owl:DatatypeProperty* and *owl:ObjectProperty*. The number of hierarchy trees in each OWL 2 output is one, as there is a single ontology root object that defines all class and property node objects in contrast to the corresponding RDFS output in which each hierarchy tree has a class as a root object. This is illustrated in Figure 2 in which the JSON Tree Viewer web component displays the frames of the Topology Profile in RDFS and OWL 2 respectively. At the same time, this is a good example how JSON-LD Framing is used to shape input data to ease further JSON to DOM rendering. The length of the longest hierarchy tree in a framed OWL 2 profile was one level longer than the length of the corresponding RDFS counterpart.

As for the average framing time (later referred to as framing time), Figure 3 illustrates how it relates to other measured values. Figure 3 (a) shows the linear dependence of framing time (presented in a logarithmic scale) with respect to the number of ontology, class and property triples in both test scenarios. Framing time was slightly longer in OWL 2 case. The reason for this was the greater number of other types of triples in input files which is confirmed with results shown in Figure 3 (b). Also, Figure 3 (b) shows that the framing time advances with a linear dependence of the number of input triples in both test scenarios. Similarly, Figure 3 (c) shows the linear dependence of framing time with respect to input file size in

both test scenarios. As for the number of output triples in both scenarios, shown in Figure 3 (d), it is nearly the same since RDFS output contains some triples that were related to defined primitive data types which were removed from OWL 2 input, while in the OWL 2 input a *rdfs:isDefinedBy* relationship was introduced and preserved in the output triples (Figure 2). It should be noted that there is a linear dependence between data shown in Figure 3 (a) and Figure 3 (d) as the output contains ontology, class, property triples together with reverse property triples. The linear dependence was present between output file sizes and framing time in both test scenarios as well Figure 3 (e), which is the consequence of the size of input files, Figure 3 (c).

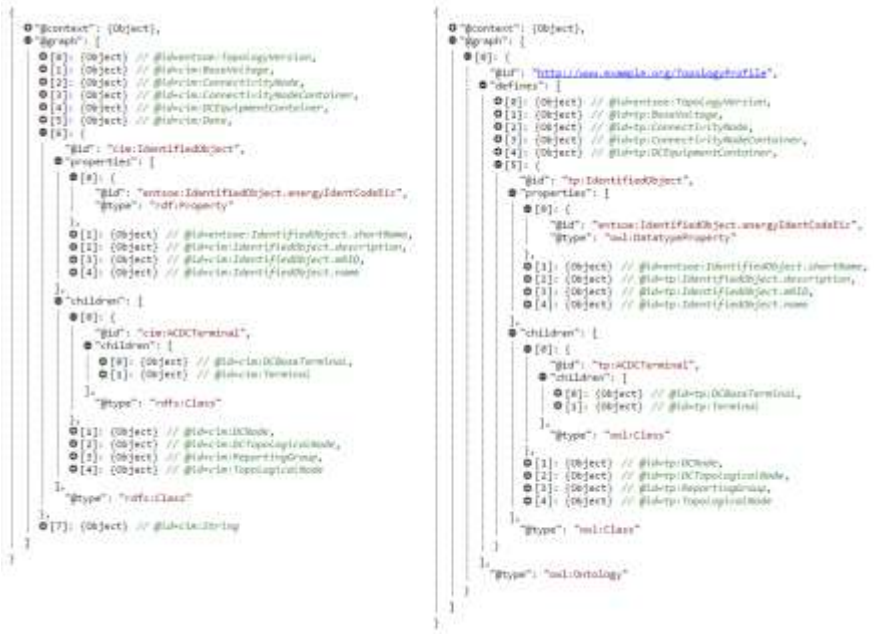

Figure 2
JSON Tree View of TopologyProfile in RDFS (left) and OWL 2 (right)

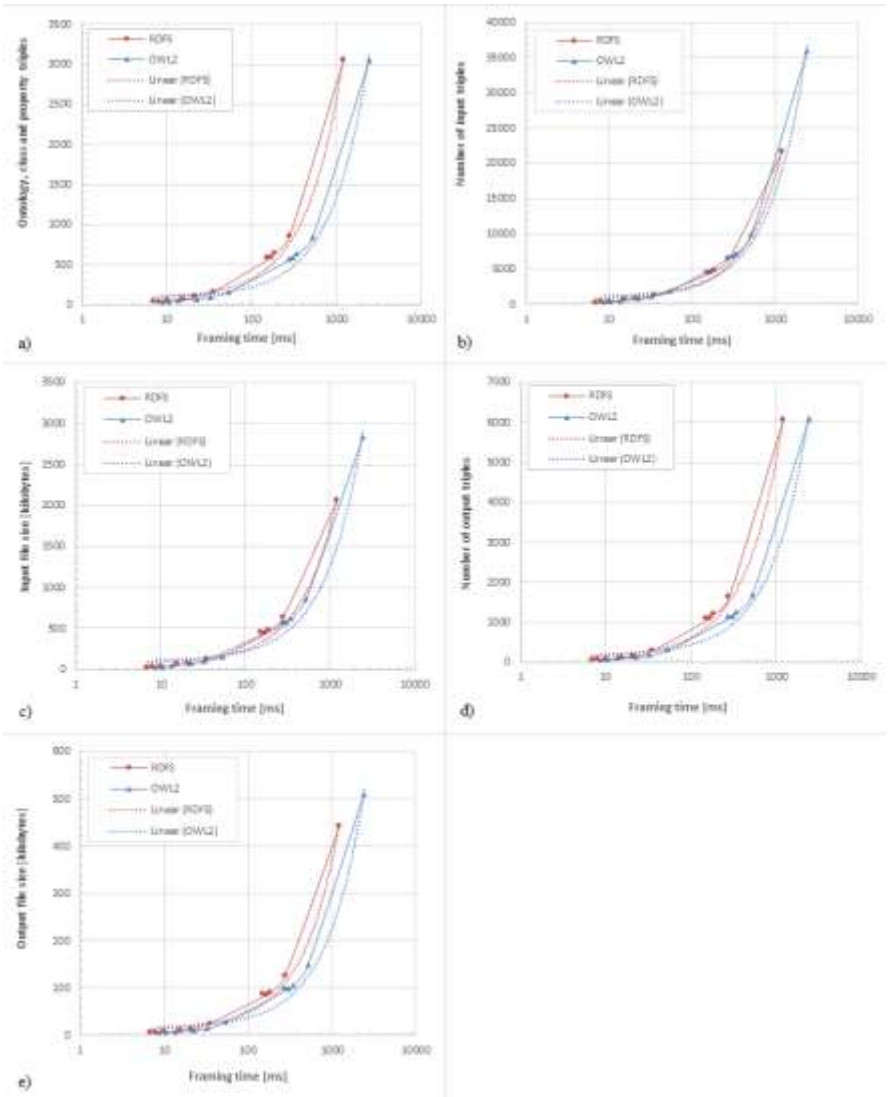

Figure 3
(a) Number of Input Ontology, Class and Property Triples vs. Framing Time,
(b) Number of Input Triples vs. Framing Time, (c) Input File Size vs. Framing Time,
(d) Number of Output Triples vs. Framing Time, (e) Output File Size vs. Framing Time

Conclusion

This paper extends the existing JSON-LD Framing specification with recursive prioritized reverse framing. Defined extensions allow definition of a frame which can be applied on an arbitrary number of input files regardless of the length of a reverse hierarchy chain of the given inverse relationship from the frame. Otherwise, a custom suited frame must be created for each input file. It also

allows combination of multiple inverse relationships in reverse tree hierarchies based on defined priorities, which the authors find suitable for grouping of related nodes using different inverse relationships. The set of existing embedding rules is extended with a new rule which enables embedding of node objects on their first occurrence. This rule, when combined with recursive reverse framing, enables creation of reverse tree hierarchies of full-length. Additionally, one of the proposed extensions enables filtering of such reverse tree hierarchies.

The proposed Extended JSON-LD Framing Algorithm is designed and implemented, and results of its application on a set of complex RDFS vocabularies and OWL 2 ontologies, using a single frame, are analyzed showing overall linear dependence of the number of input triples with respect to framing time when multiple inverse relationships are defined in a frame. The framing applied in the test scenarios shows how an arbitrary ontology can be transformed into a tree and used as input for other processes as well as how framing can be used in the validation of properly implemented RDFS into OWL 2 conversions on the examples of ENTSO-E CIM Profiles.

The future work is intended towards research of more advanced property value filtering (e.g. a property value equal to, less than, greater than some value) that could be used in conjunction with recursive prioritized reverse framing.

Acknowledgment

We would like to thank to our colleagues from Schneider Electric DMS NS LLC Novi Sad, Serbia for their support.

References

[1] U. Aguilera, O. Peña, O. Belmonte, and D. López-de-Ipiña, "Citizen-centric data services for smarter cities," *Futur. Gener. Comput. Syst.*, pp. 1-14, 2016.

[2] M. Lanthaler and C. Gütl, "Model Your Application Domain, Not Your JSON Structures," in *Proceedings of the 4th International Workshop on RESTful Design WSREST 2013 at the 22nd International World Wide Web Conference WWW2013*, 2013, pp. 1415-1420.

[3] M. Lanzenberger, J. Sampson, and M. Rester, "Ontology Visualization: Tools and Techniques for Visual Representation of Semi-Structured Meta-Data," *J. Univers. Comput. Sci.*, Vol. 16, No. 7, pp. 1036-1054, 2010.

[4] B. Fu, N. F. Noy, and M.-A. Storey, "Indented Tree or Graph? A Usability Study of Ontology Visualization Techniques in the Context of Class Mapping Evaluation," in *Proceedings of the 12th International Semantic Web Conference - Part I*, 2013, pp. 117-134.

[5] E. S. Alatrish, "Comparison of Ontology Editors," *e-RAF J. Comput.*, Vol. 4, pp. 23–38, 2012.

[6] K. Machová, J. Vrana, M. Mach, and P. Sinčák, "Ontology evaluation based on the visualization methods, context and summaries," *Acta Polytech. Hungarica*, Vol. 13, No. 4, pp. 53-76, 2016.

[7] M. Lanthaler and C. Gütl, "On using JSON-LD to create evolvable RESTful services," in *Third International Workshop on RESTful Design*, 2012, No. April, pp. 25-32.

[8] M. Sporny, G. Kellogg, D. Longley, and M. Lanthaler, "JSON-LD Framing 1.1 An Application Programming Interface for the JSON-LD Syntax," *Draft Community Group Report 04 October 2016*, 2016. [Online]. Available: http://json-ld.org/spec/latest/json-ld-framing/. [Accessed: 06-Oct-2016].

[9] D. Longley, G. Kellogg, M. Lanthaler, and M. Sporny, "JSON-LD 1.0 Processing Algorithms and API, W3C Recommendation 16 January 2014," *W3C*, 2014. [Online]. Available: https://www.w3.org/TR/json-ld-api/. [Accessed: 24-Feb-2016].

[10] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD 1.0, A JSON-based Serialization for Linked Data, W3C Recommendation 16 January 2014," *W3C*, 2014. [Online]. Available: https://www.w3.org/TR/json-ld/. [Accessed: 26-Feb-2016].

[11] W3C, "RDF Current Status," 2016. [Online]. Available: https://www.w3.org/standards/techs/rdf#w3c_all. [Accessed: 24-Nov-2016].

[12] J. Weaver and P. Tarjan, "Facebook Linked Data via the Graph API," *Semant. Web*, Vol. 4, No. 3, pp. 245-250, 2013.

[13] K. Furdík, M. Tomášek, and J. Hreňo, "A WSMO-based framework enabling semantic interoperability in e-government solutions," *Acta Polytechnica Hungarica*, Vol. 8, No. 2, pp. 61-79, 2011.

[14] Schema.org, "About Schema.org," *Schema.org*, 2014. [Online]. Available: https://schema.org/docs/faq.html. [Accessed: 13-May-2016].

[15] P. Mika, "On Schema.org and Why It Matters for the Web," *IEEE Internet Comput.*, vol. 19, no. 4, pp. 52–55, Jul. 2015.

[16] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD Implementation Report," *json-ld.org*, 2015. [Online]. Available: http://json-ld.org/test-suite/reports/.

[17] M. Sporny, G. Kellogg, D. Longley, and M. Lanthaler, "JSON-LD Framing 1.0, An Application Programming Interface for the JSON-LD Syntax," *W3C Community Group Draft Report*, 2012. [Online]. Available: http://json-ld.org/spec/ED/json-ld-framing/20120830/. [Accessed: 03-Mar-2016].

[18] M. Sporny and D. Longley, "Web Payments HTTP Messages 1.0 W3C

First Public Working Draft 15 September 2016," *W3C*, 2016. [Online]. Available: https://www.w3.org/TR/webpayments-http-messages/. [Accessed: 30-Dec-2016].

[19] R. Sanderson, P. Ciccarese, and B. Young, "Web Annotation Vocabulary W3C Candidate Recommendation 22 November 2016," 2016. [Online]. Available: https://www.w3.org/TR/annotation-vocab/. [Accessed: 27-Nov-2016].

[20] T. Kim, S. Campinas, R. Delbru, H. Jung, and S.-P. Choi, "High Performance Indexing of Materialized Graph Views," in *Proceedings of the 12th International Conference on Business Innovation and Technology Management*, 2013, pp. 1-7.

[21] T. Johnson, "Indexing Linked Bibliographic Data with JSON-LD, BibJSON and Elasticsearch," *Code4Lib J.*, No. 19, pp. 1-11, 2013.

[22] K. Nenadić, M. Letić, M. Gavrić, and I. Lendak, "Rendering of JSON-LD CIM Profile Using Web Components," in *Proceedings of the 14th International Symposium on Intelligent Systems and Informatics (SISY)*, 2016.

[23] G. Kellogg, "More specific frame matching #110," *GitHub*, 2012. [Online]. Available: https://github.com/json-ld/json-ld.org/issues/110. [Accessed: 01-Jun-2016].

[24] K. Nenadić, "Fork of jsonld.js with Recursive Prioritized Embedding Using @reverse," 2016. [Online]. Available: https://github.com/knenadic/jsonld.js.

[25] IEC, "Energy management system application program interface (EMS-API) - Part 501: Common Information Model Resource Description Framework (CIM RDF) Schema. IEC 61970-501." 2006.

[26] IEC, "Energy management system application program interface (EMS-API) - Part 552: CIMXML Model exchange format. IEC 61970-552." 2013.

[27] A. Stolz, B. Rodriguez-Castro, and M. Hepp, "RDF Translator: A RESTful Multi-Format Data Converter for the Semantic Web." 2013.

[28] K. Nenadić and M. Gavrić, "Enhancing CIM with Linked Data Capability," in *Proceedings of the 24th Telecommunications Forum (Telfor)*, 2016.