Available online at www.sciencedirect.com

ScienceDirect

Procedia Manufacturing 00 (2019) 000–000

Procedia MANUFACTURING

www.elsevier.com/locate/procedia

International Conference on Changeable, Agile, Reconfigurable and Virtual Production

Translating JSON Schema logics into OWL axioms for unified data validation on a digital manufacturing platform

Hyunmin Cheong[a,]

[a]*Autodesk Research, 661 University Ave 200, Toronto, ON M5G 1M1, Canada*

Abstract

JSON (JavaScript Object Notation) is a prevalent data format used in cloud-based platforms that support composable digital manufacturing workflows. The current work presents a method to translate the logics found in JSON Schema into OWL axioms, in order to facilitate ontology-based unified data validation with JSON data. The specific contributions of this paper include the demonstration of using a formal ontology for the logic translation and data validation, a technique for disambiguating implicit relations found in JSON Schema as explicit OWL properties, and mapping JSON Schema validation keywords to equivalent OWL expressions.

Keywords: JSON, JSON Schema; OWL; Ontology; Mapping, Axiom translation; Data validation

1. Introduction

The motivation for the current work stems from an ongoing effort to integrate semantic technologies within a cloud-based platform that supports composable digital manufacturing workflows. For example, on such a platform, a user would create design and manufacturing data that are passed around different service applications to perform modeling, simulation, optimization, process planning, etc. In this scenario, we would like to use ontologies, specifically in the OWL (Web Ontology Language) format, to mainly validate the data before they are consumed by different applications. In addition, this approach can enable semantic integration of data coming from different sources and make new inferences about given data so that the data can be interpreted by each application.

One significant challenge identified was that existing applications had already committed to their own data models and formats, often with strong resistance to any change that would facilitate integrating semantic technologies to work with their data. Hence, we had to develop techniques to work with existing data models and formats.

In particular, JSON (JavaScript Object Notation) was the most prevalent data format used on our cloud platform to exchange data between applications. JSON is language-independent, based on a nested set of key-value pairs, and

[*] Corresponding author. Tel.: +1-647-846-1677.
E-mail address: hyunmin.cheong@autodesk.com

considered as an easier-to-use alternative to XML (Extensible Markup Language). JSON data can be validated using JSON Schema, which is also written in the JSON format but with a specific set of validation keywords.

To perform ontology-based data validation with JSON data, one must: 1) map a JSON Schema file to an OWL ontology and 2) translate JSON data as OWL individuals based on the mapping. During the first task, we realized that much of the logics expressed in JSON Schema could be translated into corresponding OWL axioms that can be used for data validation. The current paper presents the technique developed to make this translation.

Several efforts have been made to map different data formats to OWL, including from XML [1, 2], UML [3], EXPRESS [4, 5], relational database schema [6], etc. In contrast, our work addresses mapping between JSON and OWL. The previous efforts of [7] and [8] also worked on mapping between JSON and OWL, but their work focused on translating JSON data into ontology classes and instances, not the logics of JSON Schema. The most relevant work, [9], used a model transformation language to translate JSON Schema logics into OWL axioms. However, their work was done with the initial version of JSON Schema and hence did not deal with cardinality and data value restrictions that can now be translated from the latest version (draft-07). Also, our work automates part of the translation process, e.g., mapping implicit relations in JSON Schema to corresponding object or data properties in an OWL ontology.

With the capability to translate JSON Schema logics into OWL axioms, unified data validation can be achieved. Figure 1 (a) shows how data validation is currently performed. A particular JSON data file is validated in different stages, e.g., against a JSON Schema file and then a pre-application validation script, before it is sent to an application. In contrast, Figure 1 (b) shows how the logics from different sources could be combined under a unified ontology. For example, the ontology could be first extended from an existing formal ontology, carrying along its axioms, then logics from JSON Schema and other application-specific validators could be translated as additional axioms of the ontology. Then, a particular JSON data file can be validated in a single step using the unified ontology. This approach also separates out the logics embedded within different data models and applications into a more common, explicit, and discoverable form, which can facilitate the management and reuse of the logics.

The rest of the paper is organized as follows. First, we provide some background information that elucidates the translation challenge addressed by the current work. Next, we present the translation method developed, followed by an example scenario to demonstrate the application of the method. We end the paper with some conclusions.

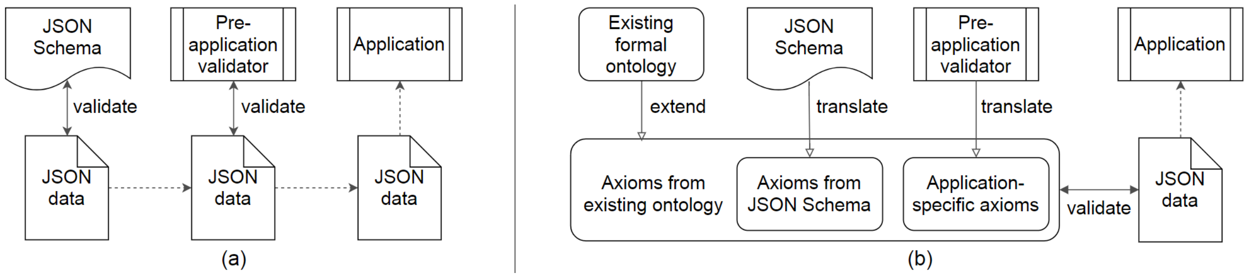
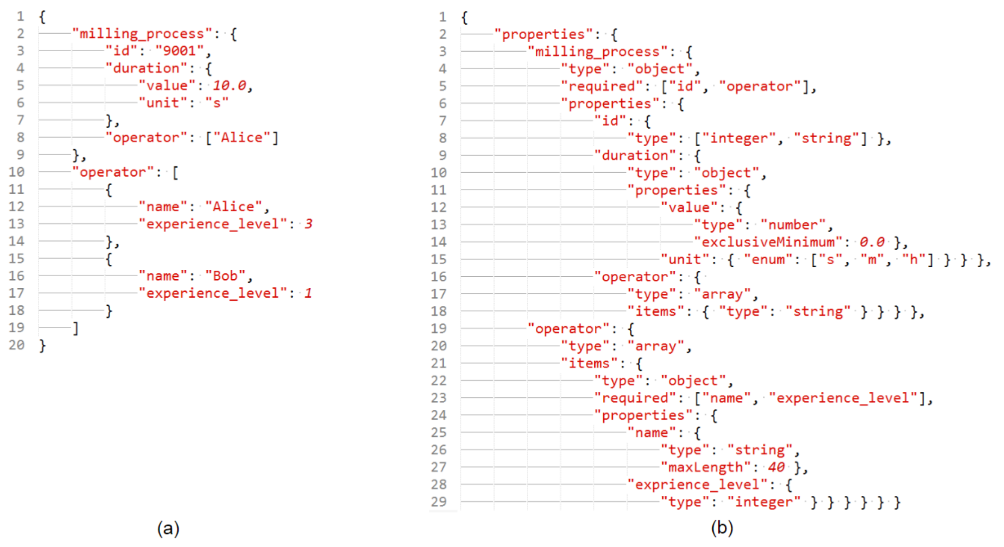


Fig. 1. (a) Current data validation process; (b) Unified data validation via an ontology.

2. Background information on JSON, JSON Schema, and OWL axioms

JSON data consist of a nested set of data objects (key-value pairs). Figure 2 (a) shows a JSON data example. Values can be one of the primitive data types such as *Boolean*, *string*, *integer*, etc., or another data object. Data objects can be categorized using hierarchical relations, but the semantics of those relations are not explicitly defined in the schema.

Valid JSON data can be specified with a JSON Schema file. Figure 2 (b) shows the corresponding JSON Schema file for the JSON data example. JSON Schema follows the same syntactic structure as JSON data, but uses a set of validation keywords to specify the corresponding data. To specify a particular data object, the keyword “properties” is used. Then, the allowed key name is stated, followed by a collection of additional specifications about that particular data object. For example, the keyword “type” specifies the data type allowed for the value of the data object or whether the value is another data object. If the latter is true, the keyword “properties” is used again to define the nested data object. Additional validation keywords are used to further specify restrictions on the data, e.g., “required”, “minLength”, “enum”, etc., a combination of which constitutes the logics used to validate a particular JSON data file.



```
{
    "milling_process": {
        "id": "9001",
        "duration": {
            "value": 10.0,
            "unit": "s"
        },
        "operator": ["Alice"]
    },
    "operator": [
        {
            "name": "Alice",
            "experience_level": 3
        },
        {
            "name": "Bob",
            "experience_level": 1
        }
    ]
}
```

(a)

```
{
    "properties": {
        "milling_process": {
            "type": "object",
            "required": ["id", "operator"],
            "properties": {
                "id": {
                    "type": ["integer", "string"] },
                "duration": {
                    "type": "object",
                    "properties": {
                        "value": {
                            "type": "number",
                            "exclusiveMinimum": 0.0 },
                        "unit": { "enum": ["s", "m", "h"] } } },
                "operator": {
                    "type": "array",
                    "items": { "type": "string" } } } },
        "operator": {
            "type": "array",
            "items": {
                "type": "object",
                "required": ["name", "experience_level"],
                "properties": {
                    "name": {
                        "type": "string",
                        "maxLength": 40 },
                    "exprience_level": {
                        "type": "integer" } } } } } }
```

(b)

Fig. 2. (a) Example JSON data; (b) JSON Schema for the data.

The types of OWL axioms to be translated from JSON Schema logics are of the following forms. We use the Manchester syntax [10] throughout this paper, with OWL classes italicized and OWL properties in bold.

- *Class* **Object Property** <cardinality expression> *Class*
- *Class* **Data Property** <cardinality expression> <datatype expression>
- *Class* **Data Property** value <literal>

3. Method

A method was developed to translate the logics of JSON Schema (draft-07) into the types of OWL axioms presented above. First, each data object specified in JSON Schema is mapped to a corresponding class in an ontology. Next, implicit relations found in JSON Schema are mapped to explicitly defined object or data properties in the ontology. Then, equivalent OWL expressions of cardinality restrictions for a set of JSON validation keywords are identified. Finally, for data property axioms, data type or data value restrictions are identified from JSON validation keywords.

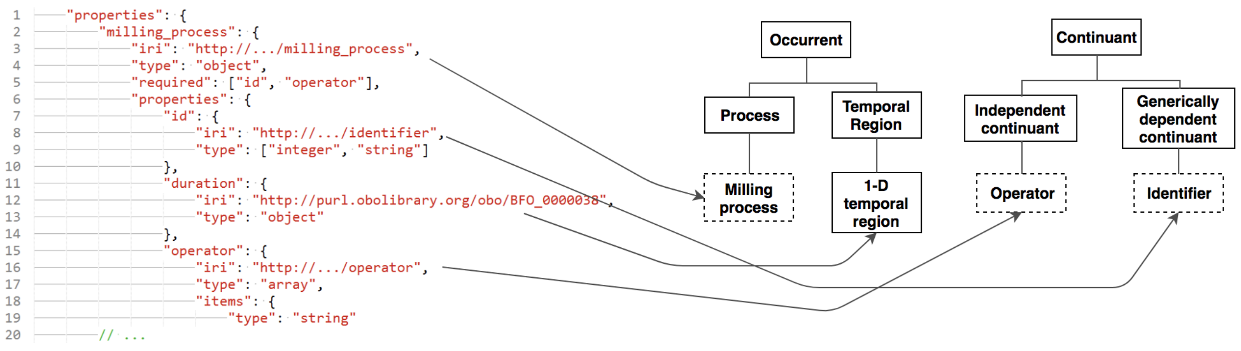
3.1. Mapping JSON Schema data objects to ontology classes

The first step is to map each JSON data object to a corresponding class in an ontology. Ideally, an existing upper ontology such as DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) [11] or BFO (Basic Formal Ontology) [12] can be extended to include application-specific classes to which mappings can be made. Starting with a formal, upper ontology ensures that all the application-specific classes can be extended from a single ontology, which is essential for unified data validation, and the logical theories that are part of the formal ontology can be reused.

The extension of a formal ontology and class mappings are performed manually. Mappings are annotated directly on a JSON Schema file, by using a keyword “iri” to specify the corresponding ontology IRI (Internationalized Resource Identifier) for each data object. Figure 3 illustrates this step.

3.2. Inferring OWL properties from implicit relations in JSON Schema

JSON Schema does not include explicit definitions of relations between data objects, but only has a single relation type via “properties” that implicitly carry different meanings depending on the data objects involved. A method was developed to disambiguate such relation and map it to a corresponding OWL property defined in an ontology. The goal



```
"properties": {
    "milling_process": {
        "iri": "http://.../milling_process",
        "type": "object",
        "required": ["id", "operator"],
        "properties": {
            "id": {
                "iri": "http://.../identifier",
                "type": ["integer", "string"]
            },
            "duration": {
                "iri": "http://purl.obolibrary.org/obo/BFO_0000038",
                "type": "object"
            },
            "operator": {
                "iri": "http://.../operator",
                "type": "array",
                "items": {
                    "type": "string"
    // ...
```

Fig. 3. Illustration of mapping data objects from JSON Schema to corresponding classes in an BFO-extended ontology

here was to automate part of the overall mapping process, which can be achieved by using the technique presented below and a set of generic relations already defined in a formal ontology.

If we assume that data objects have been mapped to corresponding OWL classes and the ontology contains a set of OWL properties defined with domain and range restrictions, the problem can be formulated as follows:

$$\underset{x \in P}{\arg\min}\; dist(domain(x), range(x)) \tag{1}$$

where P is a set of OWL object properties, x, defined in the ontology. $dist(a, b)$ computes the shortest path between the classes a and b where the ontology is transformed into a graph as follows: nodes represent classes and edges represent the existence of subclass axioms between the classes or object property axioms with the classes defined as its domain and range. This equation can be solved with any algorithm used to find the shortest path in a graph.

3.3. Finding equivalent OWL cardinality restrictions for JSON validation keywords

For a pair of data objects related in JSON Schema, depending on specific combinations of validation keywords used, the following rules can be used to identify equivalent OWL cardinality restrictions:

- If the child data object is "required" and part of an "array", the equivalent OWL expression is [min 1]
- If the child data object is "required" and not part of an "array", the equivalent OWL expression is [exactly 1]

For the example schema in Figure 2 (b), each of the two rules is applicable for identifying the cardinality restriction involved in the relations between "milling_process" and 1) "operator" and 2) "id", respectively.

3.4. Finding equivalent OWL data value/type restrictions for JSON validation keywords

As the last step, if the relation found in JSON Schema is between a data object and a data type, our method uses the following rules to map validation keywords to equivalent data value or data type restriction expressions in OWL. The acronyms rV and dT stand for restriction value and data type expressions, respectively.

- "maximum" → xsd:double[<= rV]
- "exclusiveMaximum" → xsd:double[< rV]
- "minimum" → xsd:double[>= rV]
- "exclusiveMinimum" → xsd:double[> rV]
- "maxLength" → xsd:double[maxLength rV]
- "minLength" → xsd:double[minLength rV]
- "enum" → **has value** value <literal$_1$> or **has value** value <literal$_2$> or ...
- "pattern" → xsd:double[pattern rV]
- "allOf" → dR$_1$ and dR$_2$
- "anyOf" → dR$_1$ or dR$_2$
- "oneOf" → ((dR$_1$ or dR$_2$) and not (dR$_1$ and dR$_2$))
- "not" → not dR
- "const" → **has value** value <literal>

4. Example scenario

The JSON data and JSON Schema shown in Figure 2 are considered for an example scenario. We took Basic Formal Ontology 2.0 [12] as the starting formal ontology and extended it to include application-specific classes, which are mapped to the data objects found in the schema. This resulted in the following set of subclass and equivalent class axioms. *GDC* stands for *generically dependent continuant*, which mainly subsumes information content entities.

1) "milling_process" ≡ *milling process*
2) *milling process* ⊆ *BFO: process*
3) "operator" ≡ *operator*
4) *operator* ⊆ *BFO: object*
5) "id" ≡ *identifier*
6) *identifier* ⊆ *BFO: GDC*
7) "value" ≡ *data value*
8) *data value* ⊆ *BFO: GDC*
9) "unit" ≡ *data unit*
10) *data unit* ⊆ *BFO: GDC*
11) "name" ≡ *name*
12) *name* ⊆ *BFO: GDC*
13) "experience_level" ≡ *experience level*
14) *experience level* ⊆ *BFO: Quality*
15) "duration" ≡ *BFO: 1-D temporal region*

After the class mappings are made, additional axioms are automatically translated from JSON Schema using our method. The ontology included a set of relations documented in [12] as part of BFO, e.g., **has participant**, **located in**, **bearer of**, etc., plus a relation between an information content entity and an entity it represents, **is about** [13] and its inverse relation **represented by**. These relations serve as the potential object properties to which the implicit relations found JSON Schema could be mapped. Axioms translated from the example schema in Figure 2 (b) are as follows.

16) *milling process* **represented by** exactly 1 *ID*
17) *milling process* **has participant** min 1 *operator*
18) *identifier* **has value** exactly 1 (xsd:int or xsd:string)
19) *data value* **has value** exactly 1 xsd:double[>0.0]
20) *data unit* **has value** value "s" or [...] "m" or [...] "h"
21) *operator* **represented by** exactly 1 *personal name*
22) *operator* **bearer of** exactly 1 *experience level*
23) *name* **has value** exactly 1 xsd:string[maxLength 40]
24) *experience level* **has value** exactly 1 xsd:int

To demonstrate unified data validation, shown below is an additional logical constraint that is currently checked at the application level, but could be included as part of the axioms in the ontology. The axiom states that a milling process must be carried out by an operator who has an experience level denoted higher than 2.

28) *milling process* **has participant** min 1 (*operator* and **bearer of** exactly 1 (*experience level* and **has value** exactly 1 *xsd:int*[>2]))

Also, the following axiom from BFO could be included, which states that every occurrent (hence every process) must occupy some temporal region [12].

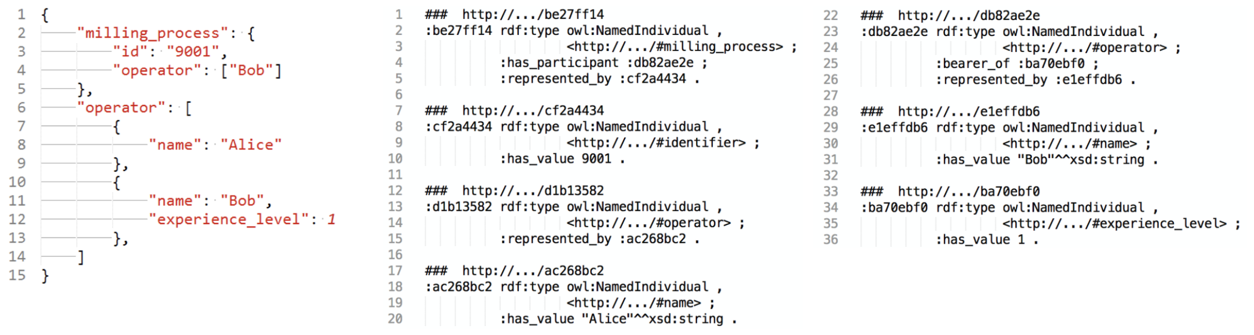
29) *process* **occupies temporal region** min 1 *temporal region*

To validate a JSON data file, it is translated into OWL statements about individuals based on the mappings established. Then, closed world assumption is enabled for reasoning. OWL-API [14] and HermiT reasoner [15] were used for data translation and reasoning. Due to space constraints, the exact translation steps are not presented here.

Figure 4 shows an example of JSON data and the translated OWL statements in the Turtle syntax, which would be inconsistent with the three types of axioms shown above. The data violates: Axiom #22 – the operator "Alice" does not have any experience level defined, Axiom #28 – the operator "Bob" does not have enough experience level for performing milling, and Axiom #29 – the milling process does not have a temporal region, e.g., "duration", defined.

5. Conclusions

The current work presented a method to translate JSON Schema logics into OWL axioms, for the goal of enabling unified data validation on a digital manufacturing platform. This was necessary to overcome the resistance to adopting new data models and formats, which is a prevalent challenge in the practical application of semantic technologies.



```json
{
    "milling_process": {
        "id": "9001",
        "operator": ["Bob"]
    },
    "operator": [
        {
            "name": "Alice"
        },
        {
            "name": "Bob",
            "experience_level": 1
        },
    ]
}
```

(a)

```
###  http://.../be27ff14
:be27ff14 rdf:type owl:NamedIndividual ,
                   <http://.../#milling_process> ;
          :has_participant :db82ae2e ;
          :represented_by :cf2a4434 .

###  http://.../cf2a4434
:cf2a4434 rdf:type owl:NamedIndividual ,
                   <http://.../#identifier> ;
          :has_value 9001 .

###  http://.../d1b13582
:d1b13582 rdf:type owl:NamedIndividual ,
                   <http://.../#operator> ;
          :represented_by :ac268bc2 .

###  http://.../ac268bc2
:ac268bc2 rdf:type owl:NamedIndividual ,
                   <http://.../#name> ;
          :has_value "Alice"^^xsd:string .

###  http://.../db82ae2e
:db82ae2e rdf:type owl:NamedIndividual ,
                   <http://.../#operator> ;
          :bearer_of :ba70ebf0 ;
          :represented_by :e1effdb6 .

###  http://.../e1effdb6
:e1effdb6 rdf:type owl:NamedIndividual ,
                   <http://.../#name> ;
          :has_value "Bob"^^xsd:string .

###  http://.../ba70ebf0
:ba70ebf0 rdf:type owl:NamedIndividual ,
                   <http://.../#experience_level> ;
          :has_value 1 .
```

(b)

Fig. 4. (a) Example data that is inconsistent with the ontology; (b) Translated OWL statements about individuals in the Turtle syntax.

Using an existing formal ontology was essential in the current work. A formal ontology, namely BFO, provided the common backbone for accommodating the mappings of data objects from JSON Schema, a set of generic relations that could be used to disambiguate implicit relations in JSON Schema, and axioms that are useful during data validation.

Not all the logics in JSON Schema were translated into OWL axioms. For example, JSON Schema supports quantifying over properties, e.g., “maxProperties”, which are not natively supported in OWL. Also, rules expressed with “if/then/else” keywords and data value arrays could only be translated using SWRL [16] and OWL LIST [17], respectively, which are not supported with many Description Logic reasoners. Nevertheless, the current work has contributed a method for integrating OWL ontologies with JSON data, a prominent data format used in cloud-based platforms.

References

[1] Ferdinand, M., Zirpins, C., & Trastour, D. (2004). Lifting XML schema to OWL. In International Conference on Web Engineering (pp. 354-358). Springer, Berlin, Heidelberg.

[2] Bohring, H., & Auer, S. (2005). Mapping XML to OWL Ontologies. Leipziger Informatik-Tage, 72, 147-156.

[3] Gasevic, D., Djuric, D., Devedzic, V., & Damjanovi, V. (2004). Converting UML to OWL ontologies. In Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters (pp. 488-489). ACM.

[4] Barbau, R., Krima, S., Rachuri, S., Narayanan, A., Fiorentini, X., Foufou, S., & Sriram, R. D. (2012). OntoSTEP: Enriching product model data using ontologies. Computer-Aided Design, 44(6), 575-590.

[5] Terkaj, W., & Pauwels, P. (2017). A Method to generate a Modular ifcOWL Ontology. In Proceedings of the 8th International Workshop on Formal Ontologies meet Industry.

[6] Cullot, N., Ghawi, R., & Ytongnon, K. (2007). DB2OWL: A Tool for Automatic Database-to-Ontology Mapping. In SEBD (pp. 491-494).

[7] Lampoltshammer, T. J., & Heistracher, T. (2014). Ontology evaluation with Protg using OWLET. Infocommunications Journal, 6(2), 12-17.

[8] Yao, Y., Wu, R., & Liu, H. (2014). JTOWL: A JSON to OWL Converto. In Proceedings of the 5th International Workshop on Web-scale Knowledge Representation Retrieval and Reasoning (pp. 13-14). ACM.

[9] Wischenbart, Martin, et al. (2013). Automatic data transformation: Breaching the walled gardens of social network platforms. In Proceedings of the Ninth Asia-Pacific Conference on Conceptual Modelling-Volume 143. Australian Computer Society, Inc.

[10] Horridge, M., Drummond, N., Goodwin, J., Rector, A. L., Stevens, R., & Wang, H. (2006). The Manchester OWL syntax. OWLed (Vol. 216).

[11] Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., & Schneider, L. (2002). Sweetening ontologies with DOLCE. In International Conference on Knowledge Engineering and Knowledge Management (pp. 166-181). Springer, Berlin, Heidelberg.

[12] B. Smith et al., (2005). Basic Formal Ontology 2.0: Specification and user’s guide. Retrieved from https://github.com/BFO-ontology/BFO/raw/master/docs/bfo2-reference/BFO2-Reference.pdf.

[13] Ceusters, W., & Smith, B. (2015). Aboutness: Towards foundations for the information artifact ontology. In Proceedings of the Sixth International Conference on Biomedical Ontology (pp. 1-5).

[14] Horridge, M., & Bechhofer, S. (2009). The OWL API: a Java API for working with OWL 2 ontologies. In Proceedings of the 6th International Conference on OWL: Experiences and Directions-Volume 529 (pp. 49-58). CEUR-WS. org.

[15] Glimm, B., Horrocks, I., Motik, B., Stoilos, G., & Wang, Z. (2014). HermiT: an OWL 2 reasoner. Journal of Automated Reasoning, 53(3), 245-269.

[16] Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., & Dean, M. (2004). SWRL: A semantic web rule language combining OWL and RuleML. W3C Member submission, 21, 79.

[17] Drummond, N., Rector, A., Stevens, R., Moulton, G., Horridge, M., Wang, H. H., & Seidenberg, J. (2006). Sequences in protg OWL. In 9th International Protege Conference (pp. 50-53).