JSON towards a simple Ontology and Rule Language

Kevin Angele[2][3] and Jürgen Angele[1]

[1] adesso, Competence Center Artificial Intelligence
juergen.angele@adesso.de
http://adesso.de
[2] Onlim GmbH
kevin.angele@onlim.com
http://onlim.com
[3] Semantic Technology Institute Innsbruck
Department of Computer Science, University of Innsbruck, Innsbruck, Austria
kevin.angele@sti2.at
http://sti2.at

Abstract. In recent years knowledge graphs gained much attention. Many big companies, like Amazon, Facebook, or Google, are building a knowledge graph. This movement also affected smaller companies, as a growing number of those ask for knowledge graph solutions or start to build them themselves. The Semantic Web provides a long list of knowledge representation formalisms and ontology languages to build knowledge graphs. Those formalisms and languages cover a broad range of computational complexity, from simple to even undecidable. Focusing on the manifoldness of companies building knowledge graphs, a language that many people understand and accept is intended. Therefore, we present an alternative approach to existing knowledge representation formalisms and ontology languages based on JSON. JSON is the most popular format for exchanging information between systems. Also, JSON comes with its schema language called JSON Schema, allowing to define the structure of a given JSON document. Our paper provides an ontology language based on JSON Schema and extends it by inheritance, arbitrary relationships between different JSON documents (classes), and rules. The rules are based on OO-logic, a successor of F-logic, and are seamlessly embedded. As we see in our current projects, this language is quickly adopted and used by software developers.

Keywords: Ontologies · JavaScript Object Notation (JSON) · JSON Schema · JSON-LD · OO-logic.

1 Introduction

Data has become an essential asset for many companies worldwide. Gartner[1] predicts that "Data and analytics will become the centerpiece of enterprise strategy, focus and investment" [9]. When it comes to managing and integrating data, maybe also from heterogeneous sources, knowledge graphs come into play. In Gartner's Hype Cycle from 2020, knowledge graphs were listed on top of the peak ("Peak of Inflated Expectation")[12]. Many big companies (like Amazon, Facebook, Google, or Microsoft) build knowledge graphs containing vast amounts of data. Also, smaller companies want to use knowledge graphs to power their systems. For building knowledge graphs, the Semantic Web provides many ontology languages and knowledge representation formalism. Besides simple concepts also undecidable ones are provided. For many use cases, companies face, those ontology languages or knowledge representation formalisms might be overly complex with a steep learning curve. Therefore, an ontology language based on an existing widespread format is required with a flat learning curve allowing companies to define their knowledge graphs.

A widespread format that is used for exchanging data between systems[2] is JSON [5]. JSON comes along with a schema language called JSON Schema[3] [10]. JSON Schema is intended to be a vocabulary to annotate and verify JSON documents (JSON-Schema for JSON is like RDFS [6] for RDF [7]). JSON Schema is used as a basis for our proposal of a bottom-up enhancement of JSON towards a simple ontology language. JSON-Schema is still in design, and therefore the paper is based on the latest draft version (2020-12[4]).

In order to provide an alternative to existing ontology languages, we provide an ontology language based on JSON-Schema that extends JSON-Schema with complex constraints, inheritance, relationships, and rules. JSON Schema provides constraints for single property values only. In this paper, inheritance is introduced, as it is not provided by JSON Schema so far. Inheritance is divided into a top-down and a bottom-up part. Top-down specifies that subtypes inherit all properties from the supertype, and bottom-up defines that instances of subtypes are also instances of supertypes. JSON Schema can not yet handle relationships between different JSON documents (classes). However, for modeling ontologies, this is an essential requirement. Finally, to be able to reason over the given data, rules must be provided. Therefore, JSON Schema is extended to be usable with OO-Logic [2] rules. The ease of using the extension of JSON Schema towards an ontology language is motivated by a demonstration of a use case.

To begin with, we present preliminaries relevant to understand the concepts introduced within this paper. The preliminaries include a brief introduction to

[1] `www.gartner.com`

[2] `https://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html#software`

[3] `http://json-schema.org`

[4] https://json-schema.org/specification.html

JSON, JSON-LD, and JSON Schema. In Section 3, extensions for JSON Schema towards a simple ontology language are introduced. Then, rules that make use of the JSON data are described. Afterward, we present a use case using the extensions of JSON Schema and the rules. The last but one section (Section 6) mainly introduces SHACL, which is comparable to JSON Schema and briefly also other rule languages present in the Semantic Web. Finally, we conclude our paper and give an outlook on future work.

2 Preliminaries

The proposal of this paper is based on an adaptation of JSON Schema, a schema language for JSON. In order to follow the extensions made, we will present the basic terms used within this paper. Therefore, we briefly introduce JSON and JSON-LD, and afterward, relevant concepts of JSON Schema are introduced.

2.1 JSON and JSON-LD

JavaScript Object Notation (JSON) [5] is designed to be a text format for the serialization of structured data. JSON is used as a lightweight, language-independent data interchange format on the Web. Nowadays, many Web APIs provide JSON as a response format. The JSON specification provides a small set of formatting rules for structuring the data. In general, JSON allows four primitive types and two structured types. The primitive types are *string, number, boolean*, and *null. Objects*, an unordered collection of name-value pairs, and *arrays*, an ordered sequence of values (either of primitive type or object), are the structured values. JSON was designed to be minimal, portable, textual, and to be a subset of JavaScript.

JSON-LD [13] is an extension of JSON to be used for serializing Linked Data. Machine interpretable data linked across different documents may also be stored in different Websites, referred to as Linked Data. Documents stored across the Web can be accessed by following links going out of the current document. JSON-LD is a lightweight syntax to serialize Linked Data based on the previously introduced JSON format. Only minimal changes are made to JSON, and therefore JSON-LD is 100% compatible with JSON. This allows reusing tools and libraries that already work with JSON. On top of JSON, JSON-LD introduces a universal identifier mechanism to identify JSON objects using IRIs[5]. Besides, to distinguish keys from different vocabularies, JSON-LD allows defining vocabularies via a so-called context. Additionally, a language is assignable to string values, and JSON-LD allows expressing directed graphs within a single document.

2.2 JSON Schema

JSON Schema [10] for JSON is comparable to XML Schema for XML documents, it defines and ensures the structure of JSON data. Specifications using

[5] `https://www.w3.org/International/O-URL-and-ident.html`

the JSON Schema syntax are themselves JSON documents. The keywords in JSON schema can be used to define constraints on JSON instances or annotate them with additional information. Assertions on instances can either be *true*, the given property value of the JSON instance complies with the defined constraint, or *false* if the property value is not compliant. For a JSON instance to be valid, it must fulfill all the previously defined assertions. Besides, annotations can provide additional information for application usage, e.g., a help text for a form created based on a JSON schema. Keywords can also be present in both categories simultaneously, specifying a constraint and providing additional information. Unknown keywords in a JSON Schema document are ignored and do not affect the validity of the validated instance. An empty schema, a schema without keywords or unknown keywords only, matches every instance. JSON Schema is still work in progress. Hence, there might be slight changes until the final version[6]. Listing 1 presents a simple JSON Schema defining the structure of JSON documents of type `Person`.

```
{
    "title":"Person",
    "@id":"http://example.org/Person",
    "@type":"Class",
    "properties":{
        "name":{
            "type":"string"
        },
        "age":{
            "type":"integer"
        }
    },
    "required":["name", "age"]
}
```

Listing 1: A JSON Schema defining the structure of instances of type person

This schema defines two required properties, namely `name` and `age`. Values given for property `name` must be `strings` and `integers` for property `age`. Since those properties are marked as *required*, a given instance must provide those properties to be valid. The keyword `additionalProperties` is not used in this schema. Hence, an instance validated against the given schema can also have more than the defined properties and is still valid. When setting `additionalProperties` to `false`, only the two defined properties are allowed.

JSON Schema supports linking different schemas to reuse parts of a schema used in many places. A reference to another schema is introduced using the `$ref` keyword. The value of this keyword is an identifier of the other schema that should be included in the specified place. When the schema is validated,

[6] we rely on the draft 2020-12 `http://json-schema.org/draft/2020-12/json-schema-core.html`

the reference is replaced by the referenced schema. Before evaluating, the content of the other schema is embedded into the source schema and then applied to the given instances. Listing 2 presents a schema including a reference to a schema representing the structure of a `DegreeCourse` (Listing 3).

```
{
"title":"Student",
"@id":"http://example.org/Student",
"type":"object",
"properties":{
    "matriculation_number":{
        "type":"integer"
    },
    "degree_course":{
        "$ref":"definitions.json#/degreeCourse"
    }
},
"required":["matriculation_number"]
}
```

Listing 2: Student schema referencing the DegreeCourse schema

```
{
"title":"DegreeCourse",
"@id":"http://example.org/DegreeCourse",
"type":"object",
"properties":{
    "name":{
        "type":"string"
    },
    "course_number":{
        "type":"integer"
    }
},
"required":["name"]
}
```

Listing 3: Schema representing the structure of DegreeCourses

For the evaluation the `DegreeCourse` schema is embedded into the root schema (see Listing 4).

Given instances must have the `DegreeCourse` embedded as well. Only a link to a `DegreeCourse` instance would be marked as invalid according to the given schema.

```
{
        "title": "Student",
        "@id": "http://example.org/Student",
        "type": "object",
        "properties": {
                "matriculation_number": {
                        "type": "integer"
                },
                "degree_course": {
                        "title": "DegreeCourse",
                        "@id": "degreeCourse",
                        "type": "object",
                        "properties": {
                                "name": {
                                        "type": "string"
                                },
                                "course_number": {
                                        "type": "integer"
                                }
                        },
                        "required": ["name"]
                }
        },
        "required": ["matriculation_number"]
}
```

Listing 4: Student schema with the DegreeCourse schema embedded

3 Extending JSON Schema to an ontology language

JSON Schema usually is used for validating JSON data which have to be structured according to the schema. In our case, we use JSON schema for describing classes and properties of classes of an ontology. So, in this case, JSON schema is not used as a constraint language. On the other hand, using JSON schema for describing ontologies has a nice side effect, that it can be used in other applications for validation purposes. Using JSON Schema as ontology language requires inheritance as an extension and adapting the semantics of relationships. In this section, we first introduce inheritance for JSON Schema and then the adapted semantics of relationships. In the following, we refer to a JSON Schema describing the structure of a JSON object as `class`.

3.1 Inheritance

For referring to a superclass relevant for the inheritance, a new keyword is introduced. In the current JSON Schema draft, keywords have the *$* as the prefix. Our keywords are based on the syntax of JSON-LD with @ as the prefix. The keyword for representing the relationship between a class and its superclass is `@subClassOf`. A class with a `@subClassOf` property will inherit all the properties and definitions from its superclass. The following example (Listing 5) shows a subclass of `Person` describing the structure of a `Student`. It defines the superclass `Person` via the `@subClassOf` keyword and therefore inherits all the properties from the superclass. In addition to the inherited properties the `Student`

```
{
    "title":"Student",
    "@id":"http://example.org/Student",
    "@type":"Class",
    "@subClassOf":"http://example.org/Person",
    "properties":{
        "matriculation_number":{
            "type":"integer"
        },
        "degree_course":{
            "type":"string"
        }
    },
    "required":["matriculation_number"]
}
```

Listing 5: Subschema

class adds another required property named `matriculation_number`. A JSON object of type `Student` must fulfill the requirements from `Person` and `Student` class to be valid.

3.2 Relationships

JSON Schema provides a reference mechanism that looks like a relationship definition at first glance. However, the original reference mechanism expects the referenced object to be embedded in the JSON instance. This is infeasible when modeling knowledge graphs. Hence a more flexible approach is needed to describe references between classes. These relationships will be defined in the `type` definition property. Instead of giving a primitive data type (like `integer` or `boolean`), the reference to another class will be given using the `@id` property. During the verification, the referenced class will not be embedded into the root class. However, the referenced object will be validated against the defined class by considering the hierarchy. The following example (Listing 6) will illustrate a relationship definition.

In that way, it is possible to refer to an object of a subclass of `DegreeCourse`.

3.3 Semantics

The semantics of the presented extensions is described by mapping JSON to Horn-Logic. Mapping means translating the incoming JSON to three types of predicates. First, the `member/2` predicate, describing the type of the given triple. Second, the `subclass/2` predicate and third, the `value/3` predicate. For this mapping the two keywords `@id` and `@type` are used. The value of the property `@id` is always the unique identifier of each triple, and the according class of the object is given using the `@type` property. Listing 7 shows the predicates resulting from the mapping of the schema of class `Person` (see Listing 1). In this example, we skipped the quotes of the predicate values for better readability.

```
{
"title":"Student",
"@id":"http://example.org/Student",
"@type":"Class",
"properties":{
    "matriculation_number":{
        "type":"integer"
    },
    "degree_course":{
        "type":"http://example.org/DegreeCourse"
    }
},
"required":["matriculation_number"]
}
```

Listing 6: Schema definition containing relationship

```
value(http://example.org/Person,title,Person).
value(http://example.org/Person,@id,http://example.org/Person).
value(http://example.org/Person,@type,Class).
member(http://example.org/Person,Class).
value(http://example.org/Person,properties,name).
value(http://example.org/Person,properties,age).
member(name,Property).
member(age,Property).
value(age,type,integer).
value(name,type,string).
value(Person,required,name).
value(Person,required,age).
```

Listing 7: Triple representation of class `Person`

```
subclass(?X, ?Z) :-  subclass(?X, ?Y), subclass(?Y, ?Z).
member(?O, ?C) :-  subclass(?C1, ?C), member(?O, ?C1).
value(?Sub,properties,?P) :-    subclass(?Sub,?Super),
                                value(?Super,properties,?P).
```

Listing 8: Inheritance closure rules

Listing 8 presents closure rules added for describing the inheritance.

The semantics is then defined by the minimal model of the resulting atoms and rules.

4 Rules for JSON

Our approach allows us to define the terminology and the structure of information with simple ontology means. As soon as we describe more complex relationships between data, we have to add additional means. We already mapped

JSON objects as well as our extended JSON schema to a logic-based representation. So it is pretty obvious to add a rule-based language for expressing complex relationships. Rules are not meant here to be used for constraints but for deducing additional facts. In [2] we have presented OO-logic as a rule and ontology language. We will show in the following that this rule language seamlessly embeds into our JSON ontology language. OO-logic is a successor of F-logic [3]. The major goal of developing F-logic was to have a rule language with a syntax easy to read and understand. Therefore conceptually different things are also represented with a different syntax. For instance, membership to a class is differentiated to a property value. We first introduce the essential ingredients of OO-logic in this chapter. Then we show how those rules can easily access JSON elements.

4.1 A brief introduction to OO-logic

OO-logic allows describing the essential parts of an ontology, i.e., classes, properties of classes with their ranges, hierarchies of classes, and relationship types between classes. In addition, instances of classes together with their property names and their relationships can be described in OO-logic easily (see Listing 9).

```
// every student is a person
Student :: Person.
// additional properties for students
Student[properties:/matriculation_number,degree_course/].

// DegreeCourse is also a class
degree_course[range:DegreeCourse].

// Sabine is a student
sabine : Student.
// Sabine has the matriculation number 5
sabine[matriculation_number:5].
// physicsA is the uid for the degree course
sabine[degree_course:physicsA].
```

Listing 9: Ontology with instances in OO-logic

Rules derive new facts from existing ones. We can now express that if a student is studying the course, then all course members can be derived. Afterward, a query to ask for all courses of Sabine can be executed (see Listing 10).

4.2 Accessing JSON by rules

OO-logic rules allow seamless access to all properties of our JSON objects. They use the same structure as the JSON objects. That a JSON object is element of a certain class (type) is expressed by colon ":", that a JSON object is a subclass

```
CourseMembers: ?C[member:?S] :- ?S:Student, ?S[degree_course:?C]:

?- ?C[member:sabine].
```

Listing 10: Query courses of sabine

of another class is expressed by two colons "::", and properties are declared by expressions in brackets ([..]). Sub-objects are addressed like related objects. A question mark precedes variables.

OO-logic provides a vast amount of so-called built-ins. All Java methods from the Math- and String libraries are available. Values can be compared; complex mathematical and boolean formulas can be expressed in rule bodies. Rule heads allow to derive new values for JSON objects' properties, create new JSON objects with their properties and relations to other JSON objects, and classify JSON objects.

```
// what are the sub-classes of class Person
?Subclass :: Person

// which members has class person, including the members of the sub-classes
?Person : Person

// the matriculation numbers of all students
?Person:Student, ?Person[matriculation_number:?MatriculationNumber]
```

Listing 11: Examples for conditions usable in rule bodies to access JSON objects

An example for creating new values for JSON objects and classifying them using built-ins to access the current time and to compute the age is given in Listing 12.

```
?P:YoungPerson, ?P[age:?Age] :-
    ?P:Person, ?Person[birthDate: ?Born], _now(?Now),
    ?Age := ?Now-?Born, ?Age < 25.

// compute age of a person by subtracting the birth date from the current time
// classify person as young person if the age is less than 25
```

Listing 12: Rule accessing JSON objects, classifying them and creating new property values

Together with the extensions of JSON Schema, those rules are implemented in a deductive database called SemReasoner (Semantic Reasoner). SemReasoner is the successor of OntoBroker [1]. In addition to OntoBroker, SemReasoner provides a powerful persistency layer and is also a stream reasoner. The syntax for the logic used by SemReasoner is OO-logic[7] [2]. Dissolving JSON objects into triples like described before (see 3.3) applies in the same way to JSON Schema as well. The internal representation of SemReasoner is based on pure Horn logic with negation. Inside SemReasoner, OO-logic is mapped to Horn logic with negation. So every Horn logic-based reasoner could be used to implement the same approach. SemReasoner also provides an inductive logic component that allows deriving rules from data similar to FOIL [11].

The JSON Schema extensions without the rules[8] can easily be implemented by extending an available open-source JSON Schema validator[9].

5 Use case

At adesso[10] we support the effort of our customers towards better digitalization processes in our projects. Input management is still an area where much manual work takes place. Our current use case for applying our technology is a KYC (know your customer) process in the banking field. If a corporate customer requests a new bank account, regulations enforce the bank to gather information about the customer to estimate the risks. Therefore, the bank will request information about the customer from risk assessment organizations like Schufa[11] or Credit Reform[12] in Germany, and the bank will analyze information delivered by the customer himself. One piece of information is the balance sheet of the company. Still, such information is often not delivered in structured form but as texts either by a PDF or even in paper form. Often, those documents are read, interpreted, and analyzed by humans, which is much effort. adesso has developed a system for analyzing balance sheets for those purposes. Classes describe different types of balance sheets with different properties. For instance, individuals and small businesses have more straightforward balance sheets while larger businesses have more complex balance sheets. Properties describe the assets and the financing. An example for a JSON schema describing an IFRS balance is given in Listing 13.

Different types of rules are used for different purposes. The terminology in such balance sheets is not standardized. In addition, it often contains typos, especially if OCR has created the balance document (if it has been received

[7] For a more detailed description of OO-Logic and the reasoning, we refer to [2], and in the future, we will publish a paper on SemReasoner.

[8] However, JSON as an ontology language can be implemented using any other reasoner and also other rule languages.

[9] `https://json-schema.org/implementations.html`

[10] `https://www.adesso.de/de/`

[11] `https://www.meineschufa.de/`

[12] `https://www.creditreform.com`

```
{
    "title":"IFRSBalance",
    "@id":"http://example.org/balance",
    "@type":"Class",
    "@subClassOf": "Balance",
    "properties":{
        "non-current assets":{
            "type":"float"
        },
        "current assets":{
            "type":"float"
        },
        "shareholders equity":{
            "type":"float"
        },
        "non-current liabilities":{
            "type":"float"
        },
        "current liabilities":{
            "type":"float"
        },
        "company":{
            "type":"Company"
        }
    }
}
```

Listing 13: A JSON Schema defining the structure of an IFRS balance

```
 // cell C1 contains the sum of the equity in the document
 // cell C1 is in the same row as the word "total shareholders equity"
 // the words are compared with similarity measure Levenshtein

 ?BS["shareholders equity": ?Sum] :-
        ?BS: Balance,
        ?T:Table,?T[cells:{?C2}], _levenshtein(?C2.value,"total shareholders
        ↪  equity",0.8), ?T[cells:{?C1}],?C1.row=?C2.row, ?C1.column=1.

// if the computed sum of all equity values is different from
// the sum written in the document a warning is given

plausibilityTableBalance("sum for shareholders equity is wrong") :-
        ?BS:Balance, sumEquity(?BS,?S), ?BS["shareholders equity":?E], ?E != ?S.
```

Listing 14: Rule to check for a plausibility

as a picture or a sheet of paper). Rules together with string similarity built-ins like Levenshtein are used to map the terms to our properties. Those rules also take the geometry, e.g., the table structure of the document, into account. Another type of rule is used for plausibility checks. For instance, a rule sums up the different positions for equities and compares it to the sum mentioned in the document. An example for a rule performing a plausibility check is given in Listing 14. Finally, the last type of rules do the analysis itself, i.e., determine whether the customer is in good financial shape or the risk with this customer is high.

At adesso this technology is also used in other applications, e.g., in a chatbot platform [4].

6 Related work

When considering the web developer community, most developers are more familiar with JSON and JSON Schema compared to standards provided by the W3C (e.g., RDF, RDFS, SHACL). However, a similar approach to JSON Schema is SHACL.

A standard of the W3C consortium for defining the structure of entities in a knowledge graph based on RDF is the **SHApes Constraint Language** (SHACL)[13]. Similar to JSON Schema, the SHACL definitions can be used to generate user interfaces.

For defining constraints over a knowledge graph, SHACL differentiates two types of shapes, *Node shapes* and *Property shapes*. A node shape defines constraints for instances, and property shapes define constraints for the properties of those instances. Node shapes together with property shapes are used to define the structure of instances of a specific type. Listing 15 shows the SHACL shape for the class `Person` (see Listing 1).

```
@prefix ex: <http://example.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Person
  a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    sh:path ex:name ;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path ex:age ;
    sh:datatype xsd:integer ;
  ] .
```

Listing 15: Node shape for the `Person` schema

Besides defining the structure of instances, SHACL allows the usage of inference rules for inferring new facts. The SHACL documentation presents SPARQL and Triple rules. Based on the implementation of the underlying reasoning engine, other rule types can be supported. For example, SHACL JavaScript Extensions[14] defines another rule type called JavaScript rules.

[13] `https://www.w3.org/TR/shacl/`

[14] `https://www.w3.org/TR/shacl-js/`

Rule Interchange Format (RIF)[15] is a rule language from the Semantic Web Community based on Horn rules without function symbols. The semantics of RIF is standard first-order semantics. Datalog extensions in RIF are used to support F-Logic features as objects and frames. RIF is not intended as a rule language for modeling but is intended as a common format for exchanging rules.

Rewerse Rule Markup Language (R2ML)[16] is a rule language based on XML supporting integrity, derivation, production and reaction rules.

Semantic Web Rule Language (SWRL)[17] combines OWL DL or OWL Lite with a subset of the Rule Markup Language. The expressivity of SWRL is equal to OWL DL, which comes at a price of decidability. Also, this makes practical implementations harder. SWRL rules consist of a body and a head. Whenever the conditions in the body are fulfilled, the condition in the head must hold as well.

OO-logic supports full Horn logic with negation. Despite using JSON Schema in contrast to SHACL, our approach is not a constraint language. In OO-logic, seamlessly embedded built-ins allow for procedural extensions. All Java built-ins from the Java packages Math and String are integrated systematically. Using a plugin API built-ins can easily be extended. RIF as an exchange format has different dialects that support negation and a restricted set of built-ins, namely functions and propositional logic [14], the same holds for SWRL. OO-logic built-ins also support aggregations like *sum, average, max*, which are not given by the other rule languages. A mapping between different rule languages in the Semantic Web (excluding OO-logic but including F-logic) is shown in [8].

7 Conclusion and future work

This paper presented a proposal for a bottom-up enhancement of JSON, especially JSON Schema, towards a simple ontology language. Therefore, it was necessary to extend the existing functionality of JSON Schema with inheritance and relationships. Relationships are a significant concept in Linked Data and therefore also for building knowledge graphs. Besides the extension of JSON Schema, we presented the mapping from JSON towards triples that can be stored in a triple store used to reason over them. This extension towards an ontology language allows describing classes, including properties. Those classes and properties are then accessible by the rules we introduced for JSON. Those rules are based on OO-logic, a successor of F-Logic. Both the extension of JSON Schema and the OO-logic rules are implemented in a deductive database called SemReasoner. However, the extensions without the rules are easily realizable by adapting an existing JSON Schema validator implementation. Finally, we demonstrated a use case using the JSON Schema extensions and OO-logic rules.

Currently, the described concepts and technology are used in several adesso projects. JSON is used in many modern software applications. This has famil-

[15] `https://www.w3.org/TR/rif-core/#Overview`

[16] `https://www.w3.org/2005/rules/wg/wiki/R2ML.html`

[17] `https://www.w3.org/Submission/SWRL/`

iarity, ease of use, and a short learning period for our software developers as a consequence. Indeed we achieved a high level of acceptance among our software developers in those projects. As a side effect, our JSON schema export is also used in other applications in forms and adessos form generator for validation purposes. So we see that JSON, together with related technologies, closes the gap between semantic technologies and conventional software engineering. From our projects, we also get valuable feedback for our further work. Currently, we are working on integrating GraphQL as an additional simple query language and a cluster version of SemReasoner.

In future, we plan to publish several papers on SemReasoners reasoning engine describing the reasoning algorithms used and specific features that are relevant for many of our current use cases.

References

1. Angele, J.: Ontobroker. Semantic Web **5**(3), 221–235 (2014)
2. Angele, J., Angele, K.: Oo-logic: a successor of f-logic. In: RuleML+ RR (Supplement) (2019)
3. Angele, J., Kifer, M., Lausen, G.: Ontologies in f-logic. In: Handbook on Ontologies, pp. 45–70. Springer (2009)
4. Angele, K., Angele, J.: Derool: A rule-based dialog engine. In: RuleML+ RR (Supplement). pp. 157–168 (2020)
5. Bray, T.: The javascript object notation (json) data interchange format pp. 1–16 (2017). https://doi.org/10.17487/RFC8259, `http://www.rfc-editor.org/info/rfc8259`
6. Brickley, D., Guha, R.V., McBride, B.: Rdf schema 1.1. W3C recommendation **25**, 2004–2014 (2014)
7. Lassila, O., Swick, R.R.: Resource description framework (RDF) model and syntax specification. World Wide Web Consortium Recommendation (October) (1999), `http://www.w3.org/TR/REC-rdf-syntax`
8. Ma, Z.M., Wang, X.: Rule interchange in the semantic web. Journal of information science and engineering **28**(2), 393–406 (2012)
9. Pettey, C.: Why Data and Analytics Are Key to Digital Transformation - Smarter With Gartner, `https://www.gartner.com/smarterwithgartner/why-data-and-analytics-are-key-to-digital-transformation/`
10. Pezoa, F., Vrgoč, D., Ugarte, M., Suarez, F., Reutter, J.L.: Foundations of JSON Schema pp. 263–273 (2017). https://doi.org/10.1145/2872427.2883029
11. Quinlan, J.R.: Learning logical definitions from relations. Machine learning **5**(3), 239–266 (1990)
12. Rimol, M.: Why Data and Analytics Are Key to Digital Transformation - Smarter With Gartner, `https://www.gartner.com/smarterwithgartner/6-trends-on-the-gartner-hype-cycle-for-the-digital-workplace-2020/`
13. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: Json-ld 1.0. W3C Recommendation **16**, 41 (2014)
14. Xiao, G., Rezk, M., Rodriguez-Muro, M., Calvanese, D.: Rules and ontology based data access. In: International Conference on Web Reasoning and Rule Systems. pp. 157–172. Springer (2014)