JSOnto: A Modular JSON-Based Syntax for Human-Friendly Ontology Engineering

David Jilg[1,2][a], and Ralph Bergmann[1,2][b]

[1]*German Research Center for Artificial Intelligence (DFKI), Branch Trier, Behringstraße 21, 54296 Trier, Germany*
{david.jilg, ralph.bergmann}@dfki.de

[2]*Artificial Intelligence and Intelligent Information Systems, Trier University, Universitätsring 15, 54296 Trier, Germany*
{david.jilg, ralph.bergmann}@uni-trier.de

Keywords: Ontology Syntax, Ontology Modeling, Knowledge Engineering, JSON

Abstract: We present JSOnto, a JSON-based ontology syntax for human-friendly, text-based knowledge engineering. Existing syntaxes and tools, such as RDF/XML, Turtle, and GUI-based editors like Protégé, are either hard to edit manually, lack modularity, or do not scale well to large ontologies. JSOnto addresses these issues with a modular multi-file design grounded in standard JSON and validated via JSON Schema. It provides concise encodings for common RDF and OWL/OWL2 constructs, supports inline annotations, and integrates with the Knowledge Transformation Framework for validation and conversion to established syntaxes. JSOnto was developed in an iterative, user-centered process and evaluated in a within-subjects user study with 26 participants, comparing JSOnto, Turtle, and Protégé on ontology modeling tasks. Less experienced users tended to prefer Protégé, whereas experienced ontology engineers clearly favored text-based modeling; among text-based options, JSOnto was preferred over Turtle and rated as more usable on several dimensions. These results suggest that JSOnto is a practical complement to existing ontology tools, particularly for large-scale ontology development and maintenance.

1 INTRODUCTION

In the context of Industry 4.0 (I4.0) (Gilchrist, 2016) and Business Process Management (BPM) (Marrella, 2019), Artificial Intelligence (AI) methods increasingly support industrial processes to improve automation, efficiency, and flexibility. Knowledge-based AI techniques such as AI planning (Green, 1969; Haslum, 2006) and Case-based reasoning (CBR) (Aamodt and Plaza, 1994; Bergmann, 2002) require extensive domain knowledge to be effective (Lenat and Feigenbaum, 1991; Alnazer and Georgievski, 2023). Ontologies are a widely used semantic data structure for representing such knowledge (Chen et al., 2024). They allow domain experts to model and maintain formal domain descriptions that can be exploited by knowledge-based methods, for instance by automatically deriving planning domains (Malburg et al., 2023). Beyond I4.0 and BPM, ontologies are a key technology for the Semantic Web (Hitzler et al., 2010), where they provide shared vocabularies and formal semantics for publishing, integrating, and reasoning over heterogeneous data and underlie many large-scale knowledge graphs and linked datasets.

Domain experts can model ontologies with GUI-based tools such as Protégé/WebProtégé (Musen, 2015; Horridge et al., 2014), Onto4ALL (Mendonça et al., 2021), or Semaphore[1]. However, modeling complex ontologies via graphical interfaces is often tedious and does not scale well (Vigo et al., 2014; García-Peñalvo et al., 2012). In Protégé, for example, large class hierarchies and forms quickly become overwhelming, and interaction is limited to navigating trees, forms, and menus. From our own work, we observe that many knowledge engineers with a computer science background prefer to edit ontologies directly in text editors instead of using GUI tools. Yet most ontology syntaxes are poorly suited for manual editing. RDF/XML (Gandon and Schreiber, 2014), still widely used for modeling with Resource De-

[a] https://orcid.org/0000-0002-4841-3512
[b] https://orcid.org/0000-0002-5515-7158

[1] https://allegrograph.com/semaphore/

scription Framework (RDF) (Cyganiak et al., 2014) and Web Ontology Language (OWL) (Motik et al., 2006), inherits the verbosity and limited readability of Extensible Markup Language (XML) (Bray et al., 2008; Matassoni et al., 2014). Moreover, ontologies are usually encoded as a single file; large ontologies easily reach tens of thousands of lines (e.g., FTOnto[2] (Klein et al., 2019)). More human-oriented syntaxes such as Terse RDF Triple Language (Turtle) (Beckett et al., 2014) or Manchester (Horridge and Patel-Schneider, 2012) improve readability but still lack modularity and cannot profit from tooling around mainstream data formats such as the JavaScript Object Notation (JSON). Overall, there is substantial potential to improve text-based ontology modeling.

In this paper, we introduce *JSOnto*, a JSON-based ontology syntax designed from scratch for manual knowledge engineering. JSOnto aims to be easy to read and write, supports modular multi-file ontologies, and leverages JSON tooling. Section 2 discusses related work, and Section 3 covers the methodology and initial observations that lead to JSOnto's development. Section 4 presents the syntax and its main design choices. Section 5 describes the user study comparing JSOnto, Turtle, and Protégé. Section 6 discusses limitations, and Section 7 concludes and outlines future work.

2 RELATED WORK

Several syntaxes exist to serialize ontologies in RDF and/or OWL, each with trade-offs in readability, expressiveness, modularity, and tool support. We focus on those most relevant for manual knowledge engineering.

The Terse RDF Triple Language (Turtle) (Beckett et al., 2014) is a popular RDF syntax designed for human readability. It offers a compact text-based notation with prefixes and list constructs and is much more readable than XML-based formats. However, Turtle lacks native support for modularity: a single ontology is typically written in a single file. Moreover, it is a bespoke syntax and does not directly map to standard data formats such as JSON or XML, limiting reuse of existing tooling such as generic validators or schema languages. The Manchester OWL Syntax (Horridge and Patel-Schneider, 2012) is another human-oriented syntax, with a frame-based, class-centric presentation well suited for documentation and editing. It shares Turtle's limitations regarding modularity and lacks the benefits of being grounded in a widely used data representation. JSON-LD (Sporny et al., 2020) is a JSON-based RDF serialization that integrates well with web technologies. It allows developers to serialize RDF in familiar JSON form while mapping keys to Uniform Resource Identifiers (URIs) using contexts and benefits from the JSON ecosystem, including validation and transformation tools. However, JSON-LD still represents a single RDF graph and does not provide a simple way to split one ontology across several files in a way that reflects a clear modular structure. Its structure is also not optimized for manual editing of ontologies. RDF/JSON (Davis et al., 2013) is another JSON-based serialization encoding triples as nested objects. It is deterministic and easy to parse but less expressive than JSON-LD. Like JSON-LD, it treats an ontology as a monolithic document and is not designed for human-friendly editing. The Distributed Ontology Language (DOL) (Lange et al., 2012) provides a framework for combining and relating heterogeneous ontologies, possibly in different formats. It can support modularity at the level of separate ontologies and imports, but is more complex than OWL imports and inherits the same difficulties when trying to partition a single ontology across many files.

In summary, Turtle and Manchester are geared towards human readers, while JSON-LD and RDF/JSON benefit from JSON-based tooling. None of these syntaxes, however, offers a simple, native mechanism to split a single ontology into multiple files while keeping a coherent namespace and avoiding complex import structures. OWL imports (*owl:imports*) are intended for reusing separate ontologies, not for structuring a single one, and managing many imports quickly becomes unwieldy. Furthermore, the easy to read syntaxes (i.e. Turtle and Manchester) not grounded in common data formats cannot directly leverage tools such as JSON Schema, JSONPath, or JSON-based APIs. These constraints motivate our design of JSOnto.

3 OBSERVATIONS AND METHODOLOGY

The design of JSOnto was grounded in observations of ontology modeling practices in our research department and refined through an iterative, user-centered process (Figure 1). This section summarizes the practical background and our methodology.

In our departments at the German Research Center for Artificial Intelligence (DFKI) and Trier University , semantic data structures, especially ontologies,

[2]https://gitlab.rlp.net/iot-lab-uni-trier/ftonto

are frequently modeled to enable knowledge-based AI methods (e.g., (Klein et al., 2019; Malburg et al., 2023; Malburg et al., 2020)). We observed that many ontology engineers in our team and collaborating groups prefer text editors or Integrated Development Environments (IDEs) over GUI tools such as Protégé for day-to-day modeling. Interviews with these researchers revealed that GUI tools are helpful for getting started but become slow and cumbersome for complex or large ontologies. In text-based environments, users benefit from features such as copy–paste, syntax highlighting, and powerful search–replace operations, which are efficient, especially for repetitive modeling tasks. In practice, RDF/XML and Turtle are the syntaxes most frequently used in our projects. RDF/XML fits well into IDE- and XML-oriented workflows but is verbose and hard to read (Matassoni et al., 2014), making manual editing difficult. Turtle is more readable but lacks a connection to standard data formats and their tooling. Both formats become unwieldy as ontologies grow; in some of our projects, single files exceed 60,000 lines (e.g., FTOnto). These experiences motivated the development of a new syntax combining human readability with better integration into existing development environments, leading to JSOnto.

We followed an iterative, design-oriented process combining user observation, feedback, and refinement cycles (Figure 1). An initial version of JSOnto was proposed based on our experience with ontology modeling and on observations of how team members used existing syntaxes. To gain a deeper understanding of user requirements, we conducted interviews with ontology modelers to identify pain points and usability issues in current tools and syntaxes. Development then proceeded in cycles: each iteration involved releasing a new syntax version, observing its use in modeling tasks, collecting feedback via follow-up interviews, and revising the syntax accordingly. This continuous feedback loop allowed us to improve readability and expressiveness step by step. The first evaluations involved a small group within our department. As JSOnto matured, we included researchers from other departments and institutions, such as SmartFactory^{KL} [3]. The scope of use evolved from small modeling exercises to real research projects. At the time of writing, JSOnto has become our preferred syntax for ontology development.

[3]https://www.smartfactory.de/

4 JSONTO

This section introduces *JSOnto*. The language specification and references to tooling are maintained in a public GitHub repository[4]. Due to space constraints, we focus on core design decisions and features most relevant for practical ontology engineering; the full specification is available in the repository.

JSOnto targets ontology modeling with the RDF/RDF Schema (RDFS) and OWL/OWL 2 stacks. It currently supports the axioms we found most common in real-world projects, rather than the full theoretical expressiveness. A JSOnto ontology consists of one or more JSON files; all must be valid JSON conforming to ECMA-404 (International, 2017). Six file types are defined, each with a dedicated JSON Schema: a mandatory *main* file (Section 4.1) and optional files for *classes*, *annotation properties*, *object properties*, *data properties*, and *individuals* (Section 4.2).

4.1 Main File

The *main* file is the only mandatory artifact of a JSOnto ontology. It contains core metadata and references to all other files. Listing 1 shows an excerpt of a main file for an exemplary family ontology used throughout this paper; the full example is available in JSOnto's repository.

Listing 1: Example for a JSOnto main file.

```
Note: URIs have been shortend.
{
  "JSOntoVersion" : "1.0",
  "ontologyUri":
      ↪ "dfki.de/jsonto/family",
  "baseNamespace": {
    "family": "dfki.de/jsonto/family#"
  },
  "namespaces":
  {
    "rdf": "w3.org/22-rdf-syntax-ns#",
    [...]
  },

  "terms:created": "2025-11-07",
  [...]

  "classes" : "exampleDirectory/",
  "objectProperties":
      ↪ "objectProperties.json",
  "dataProperties":
      ↪ "dataProperties.json",
  "annotationProperties":
      ↪ "annotationProperties.json",
  "individuals": "individuals.json"
```

[4]https://github.com/DavidJilg/JSOnto

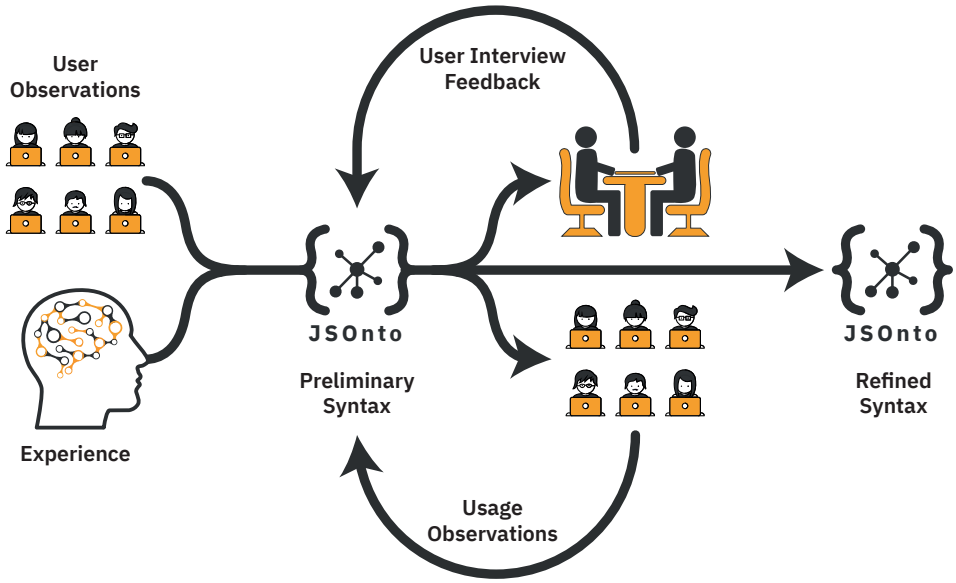


Figure 1: Overview of the methodology used to create JSOnto.

```
}
```

Because JSOnto builds on JSON, information is represented as key–value pairs. The main file defines a small set of reserved keys (e.g., `JSOntoVersion`) and a general rule for arbitrary keys. At the top level, any arbitrary key is treated as the URI of an annotation property, and the ontology itself is asserted as the subject, with the key as predicate and the value as object. Arrays can be used to express multiple annotation assertions compactly. One exception is comments: since JSON has no native comments, any key beginning with `//` is ignored by JSOnto software and may be used for human-readable comments without violating JSON validity.

In addition to annotations and comments, the main file contains structural metadata: the ontology URI, namespace prefixes, imports, and the JSOnto version. Crucially, it also specifies the locations of remaining JSOnto files using dedicated keys (`classes`, `annotationProperties`, `objectProperties`, `dataProperties`, `individuals`). Values may be single paths or arrays, relative or absolute, and can point to directories from which all files are included recursively. For small ontologies or packaging purposes, the corresponding objects can also be embedded inline under these keys instead of using separate files.

4.2 Classes, Properties, and Individuals

JSOnto uses a consistent pattern to define classes, properties, and individuals. In each file type, top-level keys denote the URIs of the entities, and values are JSON objects describing them. Arbitrary keys in these objects are interpreted as annotation assertions on the entity, as in the main file. Each file type also defines reserved keys with entity-specific semantics. We briefly summarize the most important keys; listings 2–6 illustrate their use.

Class definitions may include *equivalent* (equivalent classes), *disjointWith* (pairwise disjoint classes), *disjointUnion* (disjoint union of classes), and *subClasses* (direct subclasses). These keys cover common RDFS and OWL axioms in a concise way.

Listing 2: Exemplary excerpt from a JSOnto classes file.

```
{
  "family:Person": {
    "rdfs:label": "Person",
    "rdfs:comment": "A class used to
        ↪ represent people.",
    "subClasses": [
      "family:Woman",
      "family:Man"
    ]
  },"family:Woman": {
      [...]
}
```

Annotation properties support *range* (admissible datatypes for objects) and *subProperties* (direct subproperties).

Listing 3: Exemplary excerpt from a JSOnto annotation properties file.

```
{
  "family:Comment":
  {
    "rdfs:label": "Comment",
    "rdfs:comment": "This is an
        ↪ example for how annotation
        ↪ properties can be defined
        ↪ in JSOnto",
    "range": "xsd:float"
  },
  [...]
}
```

Object property definitions support *domain*, *range*, *equivalent*, *disjointWith*, *subProperties*, and *inverseOf*. Role characteristics are specified via Boolean flags (*functional*, *inverseFunctional*, *symmetric*, *asymmetric*, *transitive*, *reflexive*, *irreflexive*) that map directly to OWL axioms.

Listing 4: Exemplary excerpt from a JSOnto object properties file.

```
{
  "family:HasChild":
  {
      "rdfs:label": "HasChild",
      "rdfs:comment": "Used to state
          ↪ that a person has a
          ↪ child",
      "domain": "family:Person",
      "range": "family:Person",
      "inverseOf" : "family:HasParent",
      "functional": false,
      "irreflexive": true
  },
  "family:HasParent": {
    [...]
}
```

Data properties mirror object properties but constrain literals. *domain* specifies admissible subjects, *range* lists admissible datatypes, and *equivalent*, *disjointWith*, and *subProperties* capture inter-property relations. The flag *functional* indicates at most one value per subject.

Listing 5: Exemplary excerpt from a JSOnto data properties file.

```
{
  "family:Age":
  {
      "rdfs:label": "Age",
      "rdfs:comment": "Used to specify
          ↪ the age of a person",
      "domain": "family:Person",
      "range": "xsd:integer",
      "functional": true
  },
  "family:Deceased": {
    [...]
}
```

Individuals specify their classes via the *class* key and two assertion blocks. The *objectProperties* block is a JSON object whose keys are object property URIs and whose values are (i) a single URI string, (ii) an array of URI strings, or (iii) an object with mandatory `targets` and optional Boolean `negative` to express negative property assertions. The *dataProperties* block follows the same pattern but with literal values.

Listing 6: Exemplary excerpt from a JSOnto individuals file.

```
{
  "family:JamesKirk":
  {
    "class": "family:Man",
    "rdfs:label": "James Tiberius
        ↪ Kirk",
    "dataProperties":
    {
      "family:Deceased": false,
      "family:Age": 32
    },
    "objectProperties":
    {
      "family:HasParent": [
          ↪ "family:WinonaKirk",
          ↪ "family:GeorgeKirk"],
      "family:HasChild": {
        "negative": true,
        "targets": [
            ↪ "family:JamesKirk",
            ↪ "family:WinonaKirk"]
      }
    }
  },
  "family:WinonaKirk": { [...] },
  "family:GeorgeKirk": { [...] }
}
```

4.3 Tooling

Since JSOnto is released together with this paper, at the time of writing, the Knowledge Transformation Framework (KTF) developed in our group is the only software with native JSOnto support. It is open source under the MIT license and provides validation and format conversion (Jilg et al., 2026) . Validation checks syntactic well-formedness against the JSOnto JSON Schemas and performs lightweight semantic checks, such as detecting redefinitions of URIs across files.

The lack of native support is a limitation for JSOnto. For example the SPARQL Protocol and RDF Query Language (SPARQL) (Harris and Seaborne,

2013) is the standard query language for RDF-based ontologies. Since existing SPARQL engines do not consume JSOnto directly, conversion to widely supported formats is typically required for querying or tool integration. KTF supports conversion from JSOnto into several syntaxes, including RDF/XML, Turtle, and Manchester Syntax, using the OWL API[5], which also underpins Protégé and ensures interoperability. Even without many dedicated tools, JSOnto benefits from the mature JSON ecosystem because every JSOnto file is valid JSON. This has practical implications throughout the knowledge engineering process. For example, any text editor can be used to author JSOnto files. Popular IDEs offer schema-aware editing with syntax highlighting, code completion, folding, and automatic formatting. Some tools visualize nested JSON structures as interactive trees, which is useful for large ontologies (e.g., JSON Crack[6]). Additionally, existing JSON query languages and processors, such as JSON Query[7] or JMESPath[8], and JSON-based document stores like MongoDB[9], can be applied directly to JSOnto files for inspection, reporting, and quick checks without first converting to another ontology syntax. There are also numerous tools for validating JSON against JSON Schema and enforcing consistent formatting (e.g., JSONLint[10], JSON Schema Validator[11]). Many IDEs integrate these capabilities, offering immediate feedback while editing JSOnto. Finally, general-purpose converters can transform JSON into other representations such as YAML or XML[12]. While these are not semantics-preserving ontology exports, they help integrate JSOnto with tools expecting different container formats.

4.4 Advantages

JSOnto combines the human-readable, hierarchical key–value structure of JSON with features addressing limitations we experienced with RDF/XML and, to a lesser extent, Turtle. In our experience, the concise encoding of axioms and the locality of related information make JSOnto files easier to read and reason about. Listing 6 and 7 compare the JSOnto and Turtle encodings of the same individual `James Kirk`. In JSOnto, negative property assertions are expressed in place, through dedicated keys, rather than via separate reified structures as in Turtle (and similarly in RDF/XML). This reduces syntactic overhead and cognitive load when reviewing or diffing changes.

Listing 7: Individual definition in Turtle for comparison to JSOnto through listing 6.

```
family:JamesKirk rdf:type
    ↪ owl:NamedIndividual ,
     family:Man ;
     family:HasParent family:GeorgeKirk
         ↪ ,family:WinonaKirk ;
     family:Age 32 ;
     family:Deceased
         ↪ "false"^^xsd:boolean ;
     rdfs:label "James Tiberius Kirk" .

[ rdf:type
    ↪ owl:NegativePropertyAssertion ;
    owl:sourceIndividual
        ↪ family:JamesKirk ;
    owl:assertionProperty
        ↪ family:HasChild ;
    owl:targetIndividual
        ↪ family:JamesKirk
 ] .

[ rdf:type
    ↪ owl:NegativePropertyAssertion ;
    owl:sourceIndividual
        ↪ family:JamesKirk ;
    owl:assertionProperty
        ↪ family:HasChild ;
    owl:targetIndividual
        ↪ family:WinonaKirk
 ] .
```

The syntax aligns with the expectations of practitioners familiar with JSON. Common OWL constructs appear under predictable keys, which helps bridge the gap between abstract semantics and concrete representation and lowers the entry barrier for developers and data engineers who are new to ontology modeling. JSOnto also promotes organizational clarity through explicit modularization. Large ontologies can be split into separate, typed files (classes, properties, individuals), improving navigability and separation of concerns. Achieving similar structure with traditional syntaxes typically requires complex import hierarchies. File-level modularity further benefits collaboration. Plain-text JSON diffs work well with version control systems such as Git; smaller, focused files reduce merge conflicts and make reviews easier (e.g., changes in an individuals file do not obscure updates to classes).

Finally, JSOnto leverages a broad JSON tooling ecosystem. Editors, schema-based validation, automatic formatting, and programmatic manipulation are available in most programming languages. For many

[5] https://github.com/owlcs/owlapi
[6] https://jsoncrack.com/editor
[7] https://jsonquerylang.org/
[8] https://jmespath.org/
[9] https://www.mongodb.com/
[10] https://jsonlint.com/
[11] https://www.jsonschemavalidator.net/
[12] https://www.site24x7.com/tools/json-to-xml.html

workflows, text editing augmented by inline assistance is faster than navigating GUI-based ontology editors. In preliminary tests, modern code assistants quickly learn JSOnto patterns and offer useful completions, further accelerating modeling tasks.

5 EVALUATION

JSOnto is intended as a human-friendly syntax for text-based ontology modeling, both as an alternative to GUI tools (e.g., Protégé) and as an improvement over existing text syntaxes (e.g., Turtle). Our evaluation had two main goals: (1) to compare GUI-based modeling with text-based modeling and understand users' preferences, and (2) to compare JSOnto and Turtle for manual editing. We therefore compared three modeling methods: (1) Protégé, the de facto standard GUI tool; (2) a text editor or IDE using Turtle; and (3) a text editor or IDE using JSOnto. In the following, we refer to these as the modeling methods. We also wanted to understand what users liked and disliked about each method and which aspects they would improve. We collected both quantitative ratings and qualitative feedback for each method in a user study. Section 5.1 describes the setup, and Section 5.2 presents the results.

5.1 Setup

We conducted a within-subjects user study with 26 participants. Each participant used all three modeling methods, allowing direct comparison. We created three closely related modeling tasks. For all tasks, participants were provided with an ontology describing a university teaching environment (teachers, students, lectures, lecture halls). Each task required modeling a different lecture scenario by extending the ontology, in particular modeling which teachers were responsible for a lecture, which students attended, and which lecture halls were used. Participants had to create new classes and properties (object, data, and annotation) and describe individuals, thus applying basic schema- and instance-level modeling operations. Each participant solved one task with Protégé, one with Turtle in a text editor or IDE, and one with JSOnto in a text editor or IDE.

After completing the tasks, participants filled out a survey reflecting on each modeling method. The survey also collected background information, including prior familiarity with semantic technologies (RDF, OWL, ontologies), experience with the three modeling methods, and experience with adjacent technologies such as JSON. This information was used to relate modeling experiences and preferences to participant backgrounds. The questionnaire combined closed-ended questions (i.e. Likert scales (Likert, 1932)) with free-text questions asking participants what they liked or disliked about each method and what they would improve.

5.2 Survey Results

We first summarize participant backgrounds and then analyze method ratings.

Of the 26 participants, 21 reported a computer science background; the remaining five came from digital humanities (three) or business informatics (two). The top part of Figure 2 shows self-assessed experience with semantic technologies. For semantic data structures, 30.77% reported little to no experience, 34.62% some experience, 19.23% considered themselves experienced, and 15.38% very experienced. Distributions for RDF, OWL, and ontologies are similar. Overall, the sample is slightly biased towards less experienced users, but still includes a substantial number of experienced participants, which is useful for evaluating JSOnto from both perspectives. The bottom part of Figure 2 shows prior experience with Protégé, Turtle, JSOnto, and JSON. As expected, some participants had used Protégé and Turtle before, whereas virtually none had used JSOnto. In contrast, most participants reported at least some experience with JSON, which likely eased the learning JSOnto.

Participants rated each modeling method on how intuitive, efficient, easy, and comfortable it was to use, using five-point Likert scales. Figure 3 summarizes the distributions. Across all participants, Protégé and JSOnto tend to receive more positive ratings than Turtle. Both are more frequently rated as intuitive, efficient, easy, and comfortable. Protégé is rated slightly more favorably than JSOnto regarding efficiency and difficulty, while both receive similar ratings for intuitiveness and comfort. Free-text comments provide additional insights. Participants criticized Turtle for its lack of visualization, syntactic demands, and error-proneness, especially for beginners. JSOnto was also noted to lack visual support, but, unlike Turtle, was not generally perceived as unintuitive or error-prone. Several participants stressed that good editor support (e.g., via JSON Schema) is important for an optimal modeling experience with JSOnto. Criticisms of Protégé focused mainly on its cluttered and sometimes overwhelming interface.

Participants also described what they liked about each method. Positive comments on Turtle came mainly from more experienced users, who appreciated that it is "better than RDF/XML", "feels like

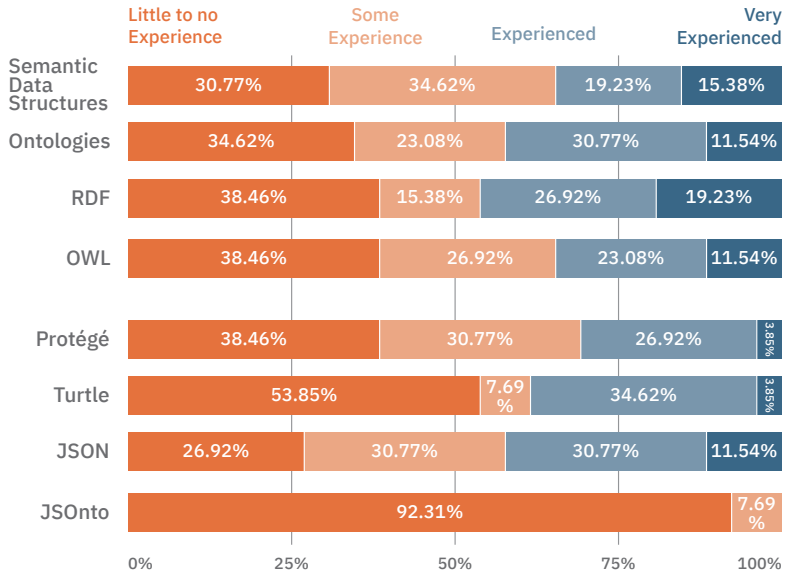


Figure 2: Experience of the participants with semantic data structures and the modeling methods that were part of the survey.

coding", and is familiar to programmers. Less experienced users, by contrast, found it unintuitive and error-prone. JSOnto was praised for its clear and modular structure, the separation of elements into different files, and its readable and editable JSON-based syntax. Protégé was valued for its graph-based visualization and for being widely used, with extensive documentation and community support. When asked which method they would choose for a future ontology modeling task, 57.69% chose Protégé, 26.92% JSOnto, and 15.38% Turtle. However, preferences strongly depend on prior experience with semantic data structures (Figure 4).

Grouping participants into those with little to some experience and those who are experienced or very experienced reveals a clear pattern. Among less experienced participants, 76.47% preferred Protégé, 11.76% Turtle, and 11.76% JSOnto. Among experienced participants, 55.56% preferred JSOnto, 22.22% Protégé, and 22.22% Turtle. Thus, experienced modelers tend to favor text-based workflows, and among text-based options, JSOnto is preferred over Turtle with experienced users.

This dependence on experience is also visible in the Likert ratings. For example, 77.78% of more experienced participants agreed that JSOnto is intuitive, compared to 55.82% among less experienced participants. Similar differences appear for efficiency (88.89% vs. 41.18%), ease of use (55.56% vs. 35.29%), and comfort (55.56% vs. 41.18%). Turtle exhibits a similar but weaker pattern. Overall, Protégé is generally preferred by less experienced users, while more experienced ontology engineers tend to prefer text-based modeling, with JSOnto clearly favored over Turtle. Figure 5 in the appendix visualizes these differences.

6 LIMITATIONS AND IMPLICATIONS FOR ADOPTION

The study and the current state of JSOnto and its tooling have several limitations that should be considered.

First, the sample size is modest (26 participants) and biased towards people with a computer science background. Many participants are comfortable with programming tools and JSON, which may favor text-based languages like JSOnto over a broader population of domain experts. At the same time, a familiarity bias in the opposite direction exists: many participants had prior experience with Protégé and Turtle but not with JSOnto, which can influence ratings and stated preferences. On the one hand, this may bias participants towards known tools; on the other hand, JSOnto still received positive evaluations and was preferred by a substantial fraction, especially among more experienced users, suggesting that its potential may be underestimated in this short-term study. Second, the experimental tasks focused on a single domain (university teaching), ontologies of moderate size, and tasks that fit into the study time frame. We therefore do not provide direct evidence on how JSOnto scales to very large ontologies, highly complex modeling patterns, or long-term maintenance.

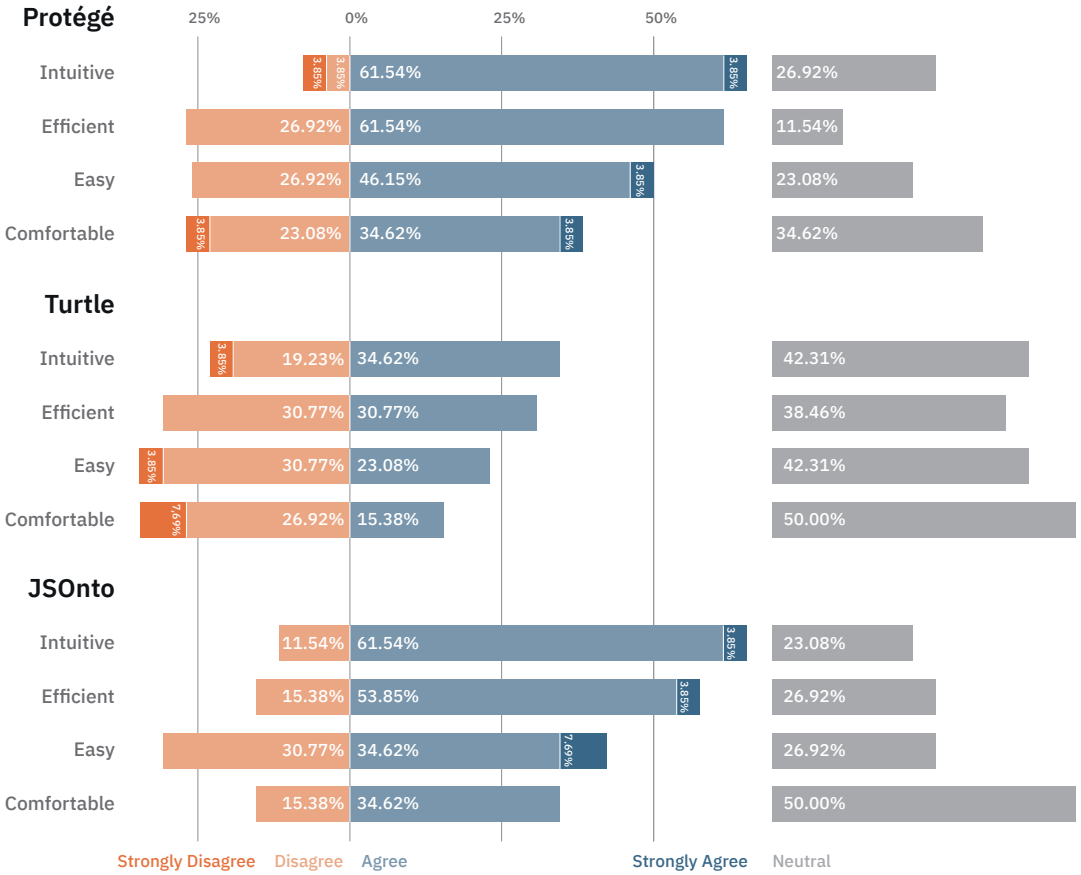


Figure 3: Survey participants opinions of modeling with the Protégé, Turtle, and JSonto.

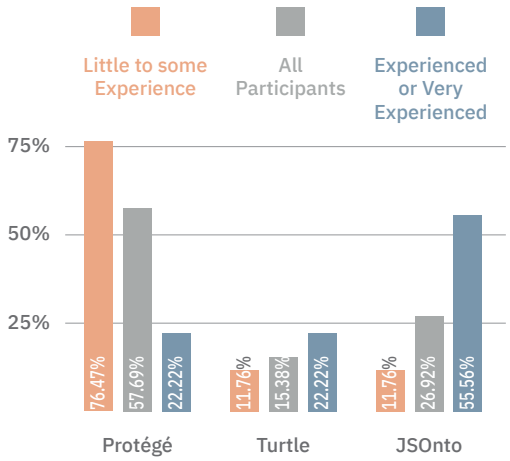


Figure 4: Preferred modeling method of the survey participants in relation to their experience with semantic data structures.

Furthermore, our evaluation emphasizes subjective usability and preferences rather than objective measures such as completion time, error rates, or ontology quality. Third, JSOnto currently concentrates on the most frequently used OWL and OWL 2 constructs observed in practice and does not cover all features of the formalisms. This reflects a trade-off between usability and coverage: focusing on common patterns simplifies the language and tooling but constrains applicability in domains that rely heavily on advanced constructs such as complex class expressions, property chains, or rare axiom types. Finally, dedicated tool support for JSOnto is still limited. While JSOnto benefits from the general JSON infrastructure, many ontology tools cannot yet handle it directly. Adoption will depend on how well JSOnto can be integrated into existing workflows, for example through import/-export in established editors or additional tooling on top of the KTF.

Overall, JSOnto should be seen as a complementary addition to the ontology engineer's toolbox. Our results suggest that hybrid workflows are promising: GUI tools such as Protégé remain useful as accessible entry points and for visual tasks, while JSOnto and other text-based syntaxes cater to experienced users and large-scale maintenance in IDEs. These implications motivate further work, discussed in the following section.

7 CONCLUSIONS AND FUTURE WORK

We introduced JSOnto, a JSON-based ontology language designed to make ontology modeling more accessible and maintainable for humans. We implemented tooling within the Knowledge Transformation Framework and evaluated JSOnto in a user study comparing it to Protégé and Turtle. The results indicate that JSOnto is perceived as more usable than Turtle and competitive with Protégé on several dimensions, particularly among participants with higher experience in semantic technologies, whereas less experienced users tend to prefer Protégé. This supports viewing JSOnto as a practical, engineer-friendly syntax for the OWL/RDF stack that can complement or, in some settings, replace existing tools.

Beyond the language itself, our work suggests how ontologies can be better integrated into software engineering practice. By aligning ontology modeling with technologies and workflows already used by software engineers—JSON-based data formats, modular file structures, and standard development tools—JSOnto can lower the barrier for adopting ontologies in real-world projects. If ontology files can be edited, reviewed, versioned, and tested alongside application code, the perceived gap between "knowledge engineering" and "software engineering" may shrink, making knowledge representation more attractive for practitioners.

The design principles behind JSOnto are not specific to OWL/RDF. A modular syntax based on structured data formats and supported by strong tooling, explicit separation of concerns, and good version-control integration could also be applied to other knowledge representation formalisms, such as rule languages, constraint systems, or domain-specific logics. Conversely, these principles might inspire the evolution of existing syntaxes and tools; for example, Turtle could benefit from more modular file organization or schema-driven tooling without changing its semantics. In this sense, JSOnto contributes both a concrete language and a design space for future knowledge engineering tooling.

Important directions for future work follow from our current limitations. A central question is how to integrate the full expressiveness of OWL and OWL 2 into JSOnto while keeping it human-friendly. Extending JSOnto to support advanced features such as complex class expressions or property chains will likely require carefully designed abstractions or templates rather than a direct transcription of existing syntaxes.

On the tooling side, we plan to extend KTF beyond the current JSOnto-to-RDF/XML direction. In particular, robust conversion from existing syntaxes such as Turtle and RDF/XML into JSOnto would allow legacy ontologies and vocabularies to be maintained in JSOnto without manual migration. This would enable incremental adoption: users could continue to publish and consume OWL/RDF in standard syntaxes while performing day-to-day maintenance in JSOnto.

Finally, future studies should broaden the empirical evaluation to more diverse participant groups, domains, and ontology sizes, and include objective measures such as task efficiency, error rates, and ontology quality. We are also interested in how JSOnto interacts with emerging assistance tools, such as LLM-based code completion and refactoring for JSON structures. Preliminary experiments in this direction have shown promising results. Together, these steps will help clarify the role of modular syntaxes like JSOnto in the broader ecosystem of ontology and knowledge representation technologies.

ABBREVIATIONS

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
BPM	Business Process Management
CBR	Case-based reasoning
DFKI	German Research Center for Artificial Intelligence
DOL	Distributed Ontology Language
I4.0	Industry 4.0
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KTF	Knowledge Transformation Framework
LLM	Large Language Model
OWL	Web Ontology Language
RDFS	RDF Schema
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
Turtle	Terse RDF Triple Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

ACKNOWLEDGEMENTS

This work is funded by the German Federal Ministry for Economic Affairs and Energy under grant No.

01MD22002C *EASY* (Schultheis et al., 2024).

USE OF GENERATIVE AI

ChatGPT (OpenAI, 2026) was used to polish the wording and correct spelling and grammar mistakes for the sections in this paper. Each section was first written by the authors without the use of any AI tools before being passed to the Large Language Model (LLM) for polishing. All LLM-adjusted texts were carefully checked sentence by sentence for accuracy by the authors. No graphs, figures, code, data, or citations were AI-generated.

REFERENCES

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59.

Alnazer, E. and Georgievski, I. (2023). Understanding real-world AI planning domains: A conceptual framework. In Aiello, M., Barzen, J., Dustdar, S., and Leymann, F., editors, *Service-Oriented Computing - 17th Symposium and Summer School, SummerSOC 2023, Heraklion, Crete, Greece, June 25 - July 1, 2023, Revised Selected Papers*, volume 1847 of *Communications in Computer and Information Science*, pages 3–23. Springer.

Beckett, D., Berners-Lee, T., Prud'hommeaux, E., and Carothers, G. (2014). Rdf 1.1 turtle. Technical report, W3C. http://www.w3.org/TR/2014/REC-turtle-20140225/.

Bergmann, R. (2002). *Experience Management: Foundations, Development Methodology, and Internet-Based Applications*, volume 2432 of *Lecture Notes in Computer Science*. Springer, Berlin, Germany.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible markup language (XML) 1.0 (fifth edition). Technical report, W3C. http://www.w3.org/TR/2008/REC-xml-20081126/.

Chen, J., Mashkova, O., Zhapa-Camacho, F., Hoehndorf, R., He, Y., and Horrocks, I. (2024). Ontology embedding: A survey of methods, applications and resources. *CoRR*, abs/2406.10964.

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). Rdf 1.1 concepts and abstract syntax. Technical report, W3C. https://www.w3.org/TR/rdf11-concepts/.

Davis, I., Steiner, T., and Hors, A. J. L. (2013). Rdf 1.1 json alternate serialization (rdf/json). Technical report, nov. https://www.w3.org/TR/rdf-json/.

Gandon, F. and Schreiber, G. (2014). Rdf 1.1 xml syntax. Technical report, W3C. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

García-Peñalvo, F. J., Palacios, R. C., Navarro, J. F. G., and Therón, R. (2012). Towards an ontology modeling tool. A validation in software engineering scenarios. *Expert Syst. Appl.*, 39(13):11468–11478.

Gilchrist, A. (2016). *Industry 4.0: The Industrial Internet of Things*. Apress, Berkeley, CA, 1st edition.

Green, C. C. (1969). Application of theorem proving to problem solving. In Walker, D. E. and Norton, L. M., editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 219–240. William Kaufmann.

Harris, S. and Seaborne, A. (2013). Sparql 1.1 query language. Technical report, W3C. https://www.w3.org/TR/sparql11-query/.

Haslum, P. (2006). *Admissible Heuristics for Automated Planning*. PhD thesis, Linköping University, Sweden.

Hitzler, P., Krötzsch, M., and Rudolph, S. (2010). *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press.

Horridge, M. and Patel-Schneider, P. F. (2012). Owl 2 web ontology language manchester syntax (second edition). Technical report, W3C. http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/.

Horridge, M., Tudorache, T., Nyulas, C., and Musen, M. A. (2014). Webprotégé: a web-based development environment for OWL ontologies. In Keet, C. M. and Tamma, V., editors, *Proceedings of the 11th International Workshop on OWL: Experiences and Directions (OWLED 2014) co-located with 13th International Semantic Web Conference on (ISWC 2014), Riva del Garda, Italy, October 17-18, 2014*, volume 1265 of *CEUR Workshop Proceedings*, pages 109–120. CEUR-WS.org.

International, E. (2017). The JSON Data Interchange Syntax. Technical Report ECMA-404, Ecma International, Geneva, Switzerland. 2nd edition, December 2017.

Jilg, D., Sehmer, F., Schander, R., and Bergmann, R. (2026). KTF: A framework for semantic knowledge transformation. Submitted for review at KEOD 2026. Preprint available at https://github.com/DavidJilg/Preprints.

Klein, P., Malburg, L., and Bergmann, R. (2019). Ftonto: A domain ontology for a fischertechnik simulation production factory by reusing existing ontologies. In Jäschke, R. and Weidlich, M., editors, *Proceedings of the Conference on "Lernen, Wissen, Daten, Analysen", Berlin, Germany, September 30 - October 2, 2019*, volume 2454 of *CEUR Workshop Proceedings*, pages 253–264. CEUR-WS.org.

Lange, C., Mossakowski, T., Kutz, O., Galinski, C., Grüninger, M., and Vale, D. C. (2012). The distributed ontology language (DOL): use cases, syntax, and extensibility. *CoRR*, abs/1208.0293.

Lenat, D. B. and Feigenbaum, E. A. (1991). On the thresholds of knowledge. *Artificial Intelligence*, 47(1):185–250.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55–55.

Malburg, L., Klein, P., and Bergmann, R. (2020). Semantic web services for ai-research with physical factory simulation models in industry 4.0. In Panetto, H., Madani, K., and Smirnov, A. V., editors, *Proceedings of the International Conference on Innovative Intelligent Industrial Production and Logistics, IN4PL 2020, Budapest, Hungary, November 2-4, 2020*, pages 32–43. SCITEPRESS.

Malburg, L., Klein, P., and Bergmann, R. (2023). Converting semantic web services into formal planning domain descriptions to enable manufacturing process planning and scheduling in industry 4.0. *Eng. Appl. Artif. Intell.*, 126:106727.

Marrella, A. (2019). Automated planning for business process management. *J. Data Semant.*, 8(2):79–98.

Matassoni, M., Rospocher, M., Dragoni, M., and Bouquet, P. (2014). Authoring owl 2 ontologies with the TeX-OWL syntax. In *Proceedings of the 11th OWL: Experiences and Directions Workshop (OWLED 2014)*, volume 1265 of *CEUR Workshop Proceedings*.

Mendonça, F. M., Emygdio, J. L., de Castro, L. P., and Felipe, E. R. (2021). Onto4alleditor: a graphic web ontology editor for information science. *Advances in Knowledge Representation*, 1(2):70–94.

Motik, B., Patel-Schneider, P. F., and Horrocks, I. (2006). Owl 1.1 web ontology language structural specification and functional-style syntax. Technical report, W3C. http://www.w3.org/submissions/2006/SUBM-owl11-owl_specification-20061219/.

Musen, M. A. (2015). The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12.

OpenAI (2026). Chatgpt (version 5.5). Large language model.

Schultheis, A., Alt, B., Bast, S., Guldner, A., Jilg, D., Katic, D., Mundorf, J., Schlagenhauf, T., Weber, S., Bergmann, R., Bergweiler, S., Creutz, L., Dartmann, G., Malburg, L., Naumann, S., Rezapour, M., and Ruskowski, M. (2024). Easy: Energy-efficient analysis and control processes in the dynamic edge-cloud continuum for industrial manufacturing. *KI - Künstliche Intelligenz*.

Sporny, M., Dave Longley, G. K., Lanthaler, M., Champin, P.-A., and Lindström, N. (2020). Json-ld 1.1 a json-based serialization for linked data. Technical report, W3C. https://www.w3.org/TR/json-ld11/.

Vigo, M., Jay, C., and Stevens, R. (2014). Design insights for the next wave ontology authoring tools. In Jones, M., Palanque, P. A., Schmidt, A., and Grossman, T., editors, *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 1555–1558. ACM.

APPENDIX

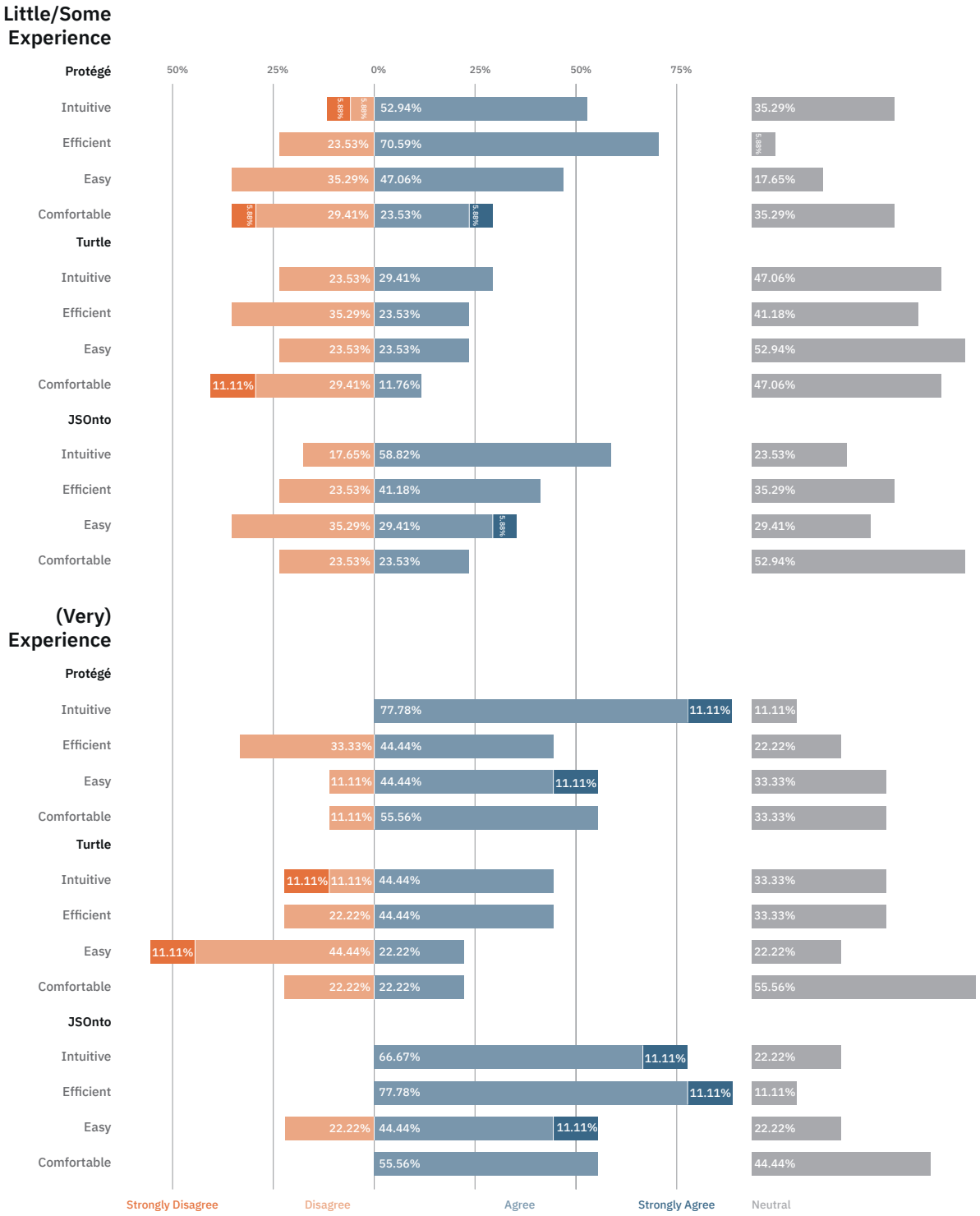


Figure 5: Survey participants opinions of modeling with Protégé, Turtle, and JSOnto in relation to their experience with semantic data structures.