Ontology Design Principles for Model-Driven Applications

How Not to Shoot Yourself in the Foot with OWL

Karl Hammar[0000−0001−8767−4136]

Jönköping AI Lab
Jönköping University, Sweden
karl.hammar@jail.ai

Abstract. This position paper presents a set of design principles for ontology engineering for model-driven applications. Ontologies sometimes need to be translated into less expressive languages and be used by software developers with limited ontology experience. In such scenarios, one may wish to refrain from using OWL features or design patterns that increase interpretation or software implementation complexity. I introduce practical considerations inherent in those scenarios, and discuss their modeling consequences.

1 Introduction

Over the past years, the author has been involved in several projects where commercial partners have used OWL to model domain-specific terminology and relations, but where the developed ontologies have thereafter not been deployed as-is, but rather been translated into other representations, e.g., JSON schema[1], Microsoft's DTDL[2], OpenAPI specifications[3], Excel skeletons [2], etc. The motivation for such translation efforts vary, but generally emanate from the perceived lack of maturity in Semantic Web deployment tooling, coupled with a greater developer or user familiarity with these alternative representation formats.

In such projects, a recurring issue has been how to best translate OWL-specific designs. Often the target runtime environments do not support some OWL language features, or those features are costly to implement or otherwise difficult for developers to use. In ontology modeling we have thus had to consider how to maintain as much of the intended meaning of an OWL ontology as possible, when translating the ontology, data expressed per the ontology, or both, to alternate representations. This has led to the development of the set of design principles that this paper proposes.

Two projects in particular have informed this work: the RealEstateCore initiative [3], focusing on Smart Building interoperability, and a collaboration with the Swedish National Veterinary Institute on syndromic surveillance data integration using the AHSO [1] ontology.

[1] https://json-schema.org/
[2] https://github.com/Azure/opendigitaltwins-dtdl
[3] https://dev.realestatecore.io/OWL2OAS/

2 Design Principles

The eight design principles provided below are grouped into three categories; those relating to classes and the class hierarchy, those relating to properties and the property hierarchy, and those relating to general reuse and reusability issues.

2.1 Classes and the Class Hierarchy

Most modeling languages support the notion of classes of things that are specialized by extension, e.g., through subclasses in OWL ontologies and OOP programming languages, extension via the `allOf` keyword in JSON schema, the `extends` keyword in DTDL, etc. In this manner, the functionality (semantics) of super-concepts can be reused by more specific sub-concepts. However, some class-modeling practices do not translate well from OWL to the latter group of languages:

Principle 1 – No Cognitive Clustering Classes. Ontologists tend to develop class hierarchies that make up reasonably balanced trees that look good in an ontology IDE such as Protégé. Sometimes this desire to construct an easy-to-interpret hierarchy leads the ontologist to create classes that are strictly speaking not necessary per the project requirements, in order to *cognitively cluster* related concepts in the ontology. For instance, a class `Information` might be created as an joint superclass of all manner of schemas, response codes, configurations, etc. that occur as classes in the ontology.

This may be sound ontological modeling (possibly aligned with some top-level ontology's design). However, in practice, the tree view of ontology classes is not necessarily used in system deployment; so designing for it can be counter-productive. When an ontology is used to create models in other languages, it is not uncommon that each class becomes its own artifact. Consequently, such an abstract class (that will not be used to represent any real data) needs to have tests written for it, to be documented, to be version-managed, etc. This should be avoided when possible.

Principle 2 – Only Create Classes for Concepts with Identity. In the Semantic Web mindset, it is not uncommon to discuss and model data without identity, i.e., data that is represented as anonymous individuals in OWL ontologies and blank nodes in RDF graphs. Examples include geometries, time intervals, etc. In other systems and formalisms, such anonymous data is typically represented as data fields on some identity-carrying entity, i.e., the building that the geometry represents, or the event that is bounded by the time interval. Adhering to the same principle for ontology modeling makes translation to other formalisms significantly easier – i.e., the ontologist should design data properties as opposed to object properties for anonymous data. Ensuring a consistent representation of the target data structure can be done either explicitly (through the use of a suitable standard or custom datatype as range) or implicitly (through

the use of `xsd:String` as range, paired with an established and documented standard string representation, e.g., WKT for geometries, or ISO8601 for time intervals).

Principle 3 – Use OWL Classes for Typing. The *Explicit Typing* [4] ontology design pattern proposes that typing of instance data be done through designated object properties and type classes, rather than the `rdf:type` and `owl:Class` language primitives. The advantage of the Explicit Typing design is that it allows types to be manipulated as instance data, such that types can be added without modifying the ontology, and types can be related using object or data properties to other entities in the dataset. This design works and can be translated into other modeling languages *as long as the ontology does not come pre-populated with a set of such type instances* – if it does, and if the target language treats model and data as disjoint (which is common), such type instances can typically not be easily translated. Furthermore, a translation method or tool taking Explicit Typing into account needs to be customized for each ontology that uses this design pattern, and the typing object property(ies) defined in it, whereas the code for translating native `owl:Class`-based typing can be made entirely generic and reusable.

2.2 Object Properties

In OWL ontologies *properties* are first-order constructs that exist independently of the classes and datatypes that they relate; whereas in many other data modeling languages, properties have no individual identity or existence in some global namespace – rather, they are local slots or fields on some classes. This has several implications when translating OWL ontologies into such languages:

Principle 4 – Avoid Property Inheritance. Given that most other data modeling languages do not support the notion of sub-properties (the instances of which imply instances of their super-properties), translating OWL models into such languages without using some sort of reification treatment, invariably loses these semantics. Arguably, even in a semantic web deployment context, the use of OWL reasoning to infer property relations via the property subsumption hierarchy, is relatively rare. Like with **Principle 1**, it is not uncommon to see ontologists create property subsumption hierarchies for the sake of ordering the world, as opposed to being driven by real application needs. This, and `rdfs:subPropertyOf`, should be avoided if possible.

Another common design that is problematic when translating OWL models into other languages is subclasses which narrow property restrictions from their superclasses. Consider the case of a model that includes a class `Collection` and a property `hasMember`, and a cardinality restriction stating each `Collection` must have at least one member. Such a model might in OWL be specialized by defining a new class, `BookCollection` with a universal restriction stating that all `BookCollections` must have only `Books` as members. Translating this design

into a modeling or programming language that does not allow property/field overloading (which many do not) is non-trivial and the design is best avoided [4].

Principle 5 – Use Short Names. Ontologies tend to use rather long and expressive names for properties, e.g., `isActuationInterfaceOf`, `hasSubComponent`, etc. This is helpful when browsing properties in ontology documentation or an IDE, where properties are typically not ordered per the classes they apply to; but in a modeling language where properties are slots on classes, such verbosity is less useful. Typically software developers use and are used to shorter names, e.g., `interfaceOf` or `subComponent` – emulating this practice is one small and easy way of bridging the divide and supporting developers in adopting ontology-backed models.

Principle 6 – Avoid Property-Defined Semantics. Taken together, **Principle 4** and **Principle 5** imply that the same translated properties may sometimes need to be used in a generic manner to represent slightly different or ambiguous semantics, depending on the class that they are used with. To exemplify: if both persons and companies have names, with different semantics or restrictions, the most semantically precise model might be one that established three properties: `hasName`, and its two sub-properties `hasPersonName` and `hasCorporateName`. Adhering to principles 4 and 5, we would instead end up with a more ambiguous property called simply `name`. Given this ambiguity, defining semantics *on the property itself* (e.g., `rdfs:domain`, `rdfs:range`, `owl:FunctionalProperty`), is unwise, as it restricts property reusability. Instead, one should strive to use restrictions on the classes with which the property is used.

2.3 Reuse and Reusability

Principle 7 – Be Opinionated. Ontologies are meant to be reused across systems and settings, which is generally a good thing and helps further interoperability across the semantic web. For the sake of reusability, ontologists often generalize their solutions a-priori, defining somewhat flexible semantics. For instance, in modeling a building topology, an ontologist might end up with very generic classes such as `Zone`, `Space`, and properties relating these such as `contains`, which allows users of the ontology to represent a specific building in multiple different ways, depending on how they view the topology.

A developer who is constructing a software system on top of an ontology-based model generally needs a language that is closer to their practice, i.e., one that mentions `Rooms` and `Floors`. Such developers ask for more rigid semantics – for instance, defining unambiguously that a building is divided into floors, which in turn are divided per building wing, which in turn are divided into individual rooms. Providing such a developer with an open-ended and overly

[4] Arguably, the `Collection` class should per **Principle 1** not be modeled at all.

flexible model is in the author's experience rarely helpful. It is in such cases better to take an opinionated position (guided by the case requirements) and nail down and remove any model ambiguity – while being open to adapting the model to changing requirements in the future.

Principle 8 – Avoid External Imports. Ontology engineering projects tend to adhere to certain modeling principles and designs (e.g., those discussed in this paper, naming principles, documentation style, a set of agreed-upon design patterns, etc.). Reusing existing ontologies via `owl:Imports` brings with it not only the semantics of those other ontologies, but their design principles as well. This can be usability-wise quite jarring to a software developer who expects a consistent experience. Undoubtedly the benefits brought by reusing an existing model can outweigh the drawbacks – but in those cases, the author recommends that reused models be imported into the target project and are homogenized with the project in terms of design and style. This also brings the added benefit of reducing the risk of link rot.

3 Summary

This paper has presented eight design principles for ontologies that are translated and used in model-driven applications. In such a context, ontology reasoning and classification is only rarely useful. Rather, the typical goal is to enable data integration through ontology-based shared data schemas and APIs. While beneficial in that case context, the principles do restrict ontology expressivity, and consequently they may not be as helpful in other scenarios.

None of these principles are absolutes – in any real-world modeling case they all need to be weighed and considered, including the drawbacks and restrictions that they impact on the resulting model.

References

1. Dórea, F.C., Vial, F., Hammar, K., Lindberg, A., Lambrix, P., Blomqvist, E., Revie, C.W.: Drivers for the development of an animal health surveillance ontology (ahso). Preventive veterinary medicine **166**, 39–48 (2019)
2. Hammar, K.: Linked Data Creation with ExcelRDF. In: The Semantic Web: ESWC 2020 Satellite Events. Springer (2020)
3. Hammar, K., Wallin, E.O., Karlberg, P., Hälleberg, D.: The RealEstateCore Ontology. In: The Semantic Web – ISWC 2019. pp. 130–145. Springer (October 2019), `http://dx.doi.org/10.1007/978-3-030-30796-7_9`
4. Shimizu, C., Hirt, Q., Hitzler, P.: MODL: A Modular Ontology Design Library. In: Proceedings of the 10th Workshop on Ontology Design and Patterns (WOP 2019) co-located with 18th International Semantic Web Conference (ISWC 2019) (2019)