Finding and Avoiding Bugs in Enterprise Ontologies

Michael Uschold
uschold@semanticarts.com

Semantic Arts Inc.

Abstract. We report on ten years of experience building enterprise ontologies for commercial clients. We describe key properties that an enterprise ontology should have, and illustrate them with many real world examples. They are: correctness, understandability, usability, and completeness. We give tips and guidelines for how best to use inference and explanations to identify and track down problems. We describe a variety of techniques that catch bugs that an inference engine will not find, at least not on its own. We describe the importance of populating the ontology with data to drive out more bugs. We point out some common ontology design practices in the community that lead to bugs in ontologies and in downstream semantic web applications based on the ontologies. These include proliferation of namespaces, proliferation of properties and inappropriate use of domain and range. We recommend doing things differently to prevent bugs from arising.

Keywords: ontology evaluation, inference, enterprise ontology, ontology debugging, OWL

1 Introduction

In a manner analogous to software debugging, ontologies need to be rid of their flaws. The types of flaws to be found in an ontology are slightly different than those found in software, and revolve around the ideas of correctness, understandability, usability and completeness.

We report on our experience (spanning more than a decade) in building and debugging enterprise ontologies for large companies in a wide variety of industries including: finance, healthcare, legal research, consumer products, electrical devices, manufacturing and digital assets. For the growing number of companies starting to use ontologies, the norm is to build a single ontology for a point solution in one corner of the business. For large companies, this leads to any number of independently developed ontologies resulting in many of the same heterogeneity problems that ontologies are supposed to solve. It would help if they all used the same upper ontology, but most upper ontologies are unsuitable for enterprise use. They are hard to understand and use because they are large and complex, containing much more than is necessary, or the focus is too academic to be of use in a business setting. So the first step is to start with a small, upper, enterprise ontology such as gist [McComb 2006], which includes core concepts relevant to almost any enterprise. The resulting enterprise ontology itself will consist of a mixture of concepts that are important to any enterprise in a given industry, and those that are important to a particular enterprise.

An enterprise ontology plays the role of an upper ontology for all the ontologies in a company (Fig. 1). Major divisions will import and extend it. Ontologies that are specific to particular applications will, in turn, import and extend those. The enterprise ontology evolves to be the semantic foundation for all major software systems and databases that are core to the enterprise.

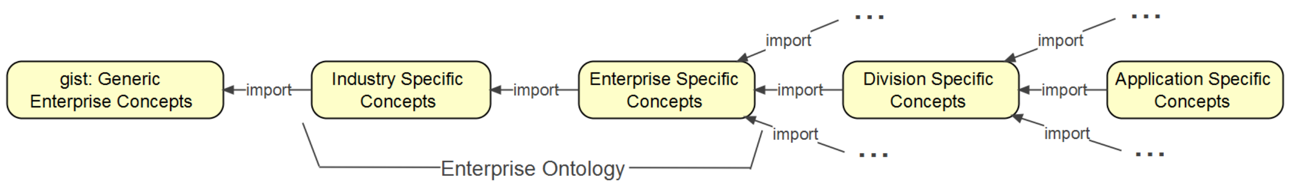

gist: Generic Enterprise Concepts
import
Industry Specific Concepts
import
Enterprise Specific Concepts
import
Division Specific Concepts
import
Application Specific Concepts
import
import
import
import
...
Enterprise Ontology

Fig. 1. Enterprise Ontology Layers

In this paper we give an overview of how we ensure a quality product when building an enterprise ontology. We use OWL as the ontology language; we use a proprietary plugin to Visio to author ontologies; and we examine them and run inference using Protégé. Much of what we describe applies to any ontology. However, there are a number of things that are less important for an enterprise ontology, other things are more important. For example, in all our commercial work, we have found that getting things done quickly and effectively trumps academic nicety.

We proceed by introducing the major ways that an ontology can be regarded as imperfect. These revolve around the ideas of *correctness*, *understandability*, *usability*, and *completeness*. We give tips for debugging errors, and explain what can be done to avoid bugs in the first place. We describe the importance of test data. We close with a discussion of related work indicating what things are more important for an enterprise ontology, as compared to any ontology in general. All the examples in this paper draw from enterprise ontologies with commercial clients.

2 Factors for improving the quality and usability of an ontology

Our focus will be broader than bugs, per se, encompassing anything that can be done to improve the quality and usefulness of an ontology. To be useful, an enterprise ontology should be not only correct (i.e. error-free) but also easy to understand. A major factor contributing to understandability is avoiding bad practices. Finally, the ontology needs to be sufficiently complete in scope to meet its intended requirements. An ontology that is hard to use will result in errors and bugs in downstream applications that depend on the ontology. We will consider these factors, roughly in order of importance.

2.1 Correctness

Does the ontology have any mistakes? When compiling software, it is common to distinguish an ERROR from a WARNING. The former means something is clearly wrong and must be fixed. The latter typically corresponds a known bad practice, or unintentional mistake. But there are countless other problems with software that may arise separately from those found by the compiler. The situation is similar for ensuring high-quality ontologies. First we consider errors. Running an inference engine over an ontology will identify logical inconsistencies that must be fixed.

Using inference to catch bugs.

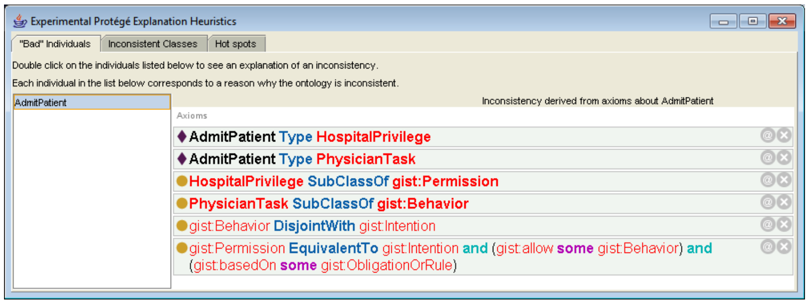
Below are two things useful to know when developing an ontology:

1. how to track down bugs that inference helps to catch;
2. how to build your ontology in a way that allows inference to catch even more bugs.

The first check is to run inference. There are two types of problems this can detect the ontology can be logically inconsistent or there may be unsatisfiable classes (i.e. that cannot possibly have any members). In both cases, Protégé provides an explanation facility to give insights into what went wrong.

Inconsistent Ontology

An inconsistent ontology means that there is a logical inconsistency somewhere – one or more statements are proven to be both true and false. In Protégé, a window will pop up and you can click on a question mark to get an explanation. Fig. 2 shows an explanation for why a healthcare ontology is inconsistent. There is an individual called `AdmitPatient` and that is an instance of both `HospitalPrivilege` and `PhysicianTask`. While individually, both seem to make perfect sense, it turns out that the former is a `Permission` (which is a subclass of `Intention`), and the latter is a `Behavior`. Since `Behavior` is disjoint from `Intention`, the ontology is inconsistent. This example is relatively easy to track down.



Double click on the individuals listed below to see an explanation of an inconsistency.
Each individual in the list below corresponds to a reason why the ontology is inconsistent.

AdmitPatient

Inconsistency derived from axioms about AdmitPatient

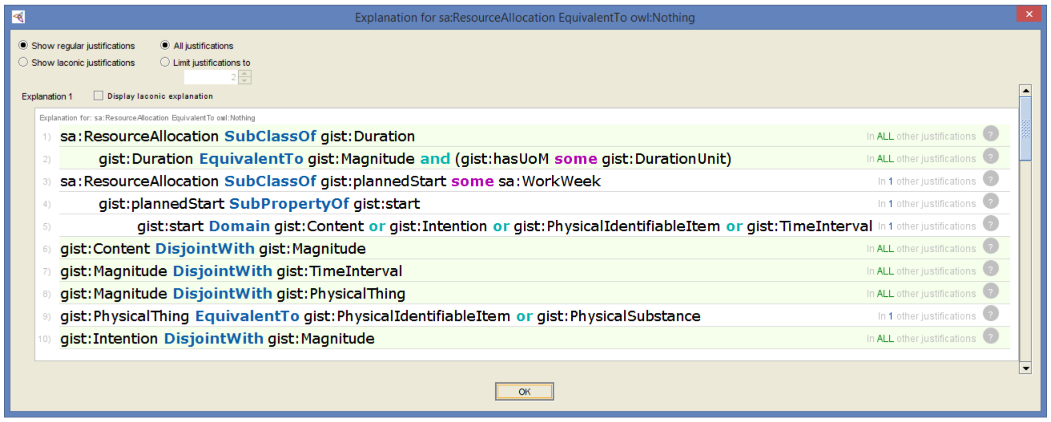
Axioms

```
♦AdmitPatient Type HospitalPrivilege
♦AdmitPatient Type PhysicianTask
●HospitalPrivilege SubClassOf gist:Permission
●PhysicianTask SubClassOf gist:Behavior
●gist:Behavior DisjointWith gist:Intention
●gist:Permission EquivalentTo gist:Intention and (gist:allow some gist:Behavior) and
 (gist:basedOn some gist:ObligationOrRule)
```

Fig. 2. Inconsistent Ontology Explanation

Unsatisfiable Class

Fig. 3 shows an example of an unsatisfiable class (i.e. one that is equivalent to `owl:Nothing`).



Explanation for sa:ResourceAllocation EquivalentTo owl:Nothing

(●) Show regular justifications (●) All justifications
( ) Show laconic justifications ( ) Limit justifications to 2

Explanation 1 ☐ Display laconic explanation

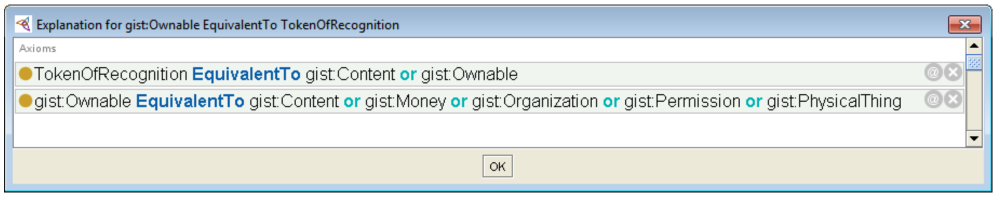
Explanation for: sa:ResourceAllocation EquivalentTo owl:Nothing

1) sa:ResourceAllocation **SubClassOf** gist:Duration — In ALL other justifications
2) gist:Duration **EquivalentTo** gist:Magnitude **and** (gist:hasUoM **some** gist:DurationUnit) — In ALL other justifications
3) sa:ResourceAllocation **SubClassOf** gist:plannedStart **some** sa:WorkWeek — In 1 other justifications
4) gist:plannedStart **SubPropertyOf** gist:start — In 1 other justifications
5) gist:start **Domain** gist:Content **or** gist:Intention **or** gist:PhysicalIdentifiableItem **or** gist:TimeInterval — In 1 other justifications
6) gist:Content **DisjointWith** gist:Magnitude — In ALL other justifications
7) gist:Magnitude **DisjointWith** gist:TimeInterval — In ALL other justifications
8) gist:Magnitude **DisjointWith** gist:PhysicalThing — In ALL other justifications
9) gist:PhysicalThing **EquivalentTo** gist:PhysicalIdentifiableItem **or** gist:PhysicalSubstance — In 1 other justifications
10) gist:Intention **DisjointWith** gist:Magnitude — In ALL other justifications

Fig. 3. Unsatisfiable Class Explanation

Other ways inference can help.

While using inference to detect logical inconsistencies and unsatisfiable classes is a powerful technique, there are many kinds of modeling mistakes that this technique cannot find. We find that it only catches 20-30% of the errors in an ontology. This figure could vary, depending on the specifics of the ontology, the modeling approach and the experience of the modeler. One important way to find more errors is to manually examine the inferred hierarchy. You will often find subclass or equivalence relationships that do not make sense, i.e. they do not accurately reflect the subject matter you are modeling. If you have an ontology with several hundred classes and plenty of inferred subclass relationships, it could take several tens of minutes to open up each level of the hierarchy, and identify problems.



Explanation for gist:Ownable EquivalentTo TokenOfRecognition

Axioms

TokenOfRecognition **EquivalentTo** gist:Content **or** gist:Ownable

gist:Ownable **EquivalentTo** gist:Content **or** gist:Money **or** gist:Organization **or** gist:Permission **or** gist:PhysicalThing

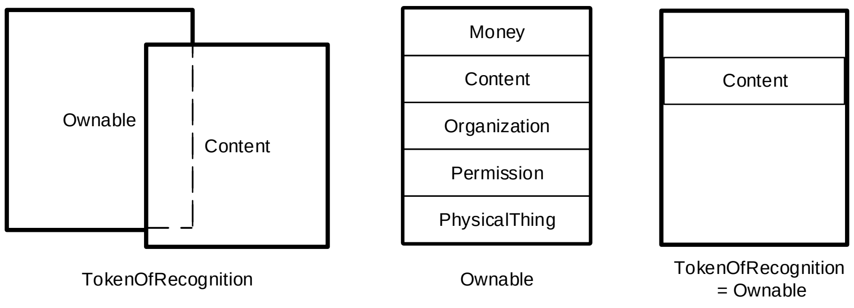
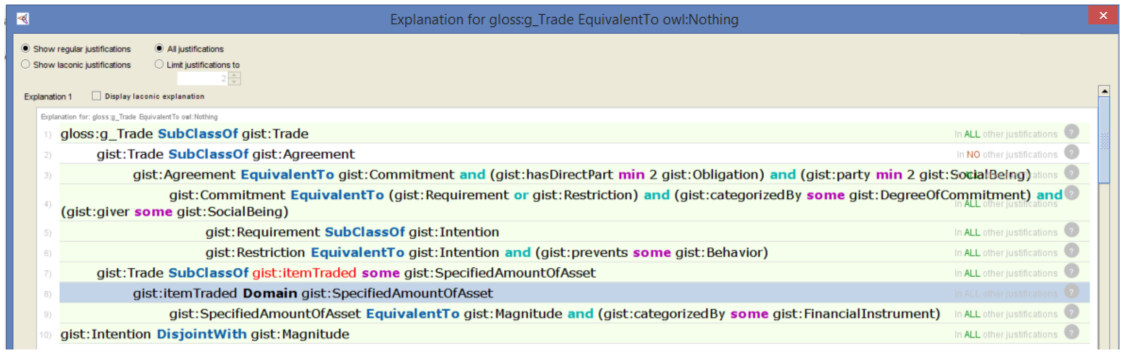


Fig. 4. Unexpected Equivalent Classes

Figure 4 shows a case where two classes are unexpectedly inferred to be equivalent. It took a while to find the problem, which turned out to be based on simple idea. It boils down to the fact that if class C1 is a subclass of class C2, then C2 is equivalent to C2 OR C1. Even for an explanation with only two lines, it can help to draw a Venn diagram to see what is going on. The Venn diagram on the left is the most general way to depict the union of Ownable and Content. Looking at the Venn diagram depicting Ownable as the union of 5 classes immediately shows that the diagram on the left is wrong, the diagram on the right is accurate. Since Content is ‘inside’ Ownable, taking the Union of Ownable with Content gives the original Ownable, which makes it equivalent to TokenOfRecognition. In this case, inference did not directly tell you there was a problem, but it did help find and track down a problem.

Debugging Hints

The explanation itself consists of a set of true statements (either by direct assertion, or by inference). If the explanation is merely a few lines long, the problem will often be easy to spot; otherwise, it can be a slow process to carefully follow the logic. One shortcut is to quickly scan each line to see if anything is clearly wrong. An easy-to-spot error is getting the domain and range mixed up. Fig. 5 explains why the class `Trade` is unsatisfiable. A `Trade` points to the item traded (e.g. 10 shares of Apple). `SpecifiedAmountOfAsset` should have been specified as the range of the property `itemTraded`, but it was mistakenly specified as the domain. This is a quick fix.



Explanation for gloss:g_Trade EquivalentTo owl:Nothing

[✓ Show regular justifications] Show laconic justifications
[✓ All justifications] Limit justifications to 2

Explanation 1 ☐ Display laconic explanation

Explanation for: gloss:g_Trade EquivalentTo owl:Nothing

```
1) gloss:g_Trade SubClassOf gist:Trade                                                    In ALL other justifications
2)     gist:Trade SubClassOf gist:Agreement                                               In NO other justifications
3)         gist:Agreement EquivalentTo gist:Commitment and (gist:hasDirectPart min 2 gist:Obligation) and (gist:party min 2 gist:SocialBeing)
4)             gist:Commitment EquivalentTo (gist:Requirement or gist:Restriction) and (gist:categorizedBy some gist:DegreeOfCommitment) and
(gist:giver some gist:SocialBeing)                                                        In ALL other justifications
5)                 gist:Requirement SubClassOf gist:Intention                             In ALL other justifications
6)                 gist:Restriction EquivalentTo gist:Intention and (gist:prevents some gist:Behavior)   In ALL other justifications
7)     gist:Trade SubClassOf gist:itemTraded some gist:SpecifiedAmountOfAsset             In ALL other justifications
8)         gist:itemTraded Domain gist:SpecifiedAmountOfAsset                             In ALL other justifications
9)             gist:SpecifiedAmountOfAsset EquivalentTo gist:Magnitude and (gist:categorizedBy some gist:FinancialInstrument)   In ALL other justifications
10) gist:Intention DisjointWith gist:Magnitude                                            In ALL other justifications
```

Fig. 5. Domain and Range Mixed Up

If you cannot spot anything obvious, then consider what you changed in the ontology since you last successfully ran inference. This helps you know where to look for the problem, and can save time. Making too many changes in between running inference will reduce the effectiveness of this aid.

Often, a simple mistake somewhere can result in large swaths of inconsistent classes. There is often a single problem that had major ripple effects. The trick is how to find it. Start with the obvious: consider what most recently changed in the ontology. If all the explanations are very long, try semi-randomly clicking at a bunch of different classes to find one with a short explanation that will be easier to debug. Most roads lead to Rome. If you are stuck with only long explanations, then find a set of axioms that are related to a single theme; this can help narrow the search. Try drawing some Venn diagrams; it can help you see what is going on. When all else fails, it can be necessary to somewhat arbitrarily remove different sets of axioms and then running inference each time to see if the same error persists. Start by removing roughly half of the axioms, and then another half etc. This can get you there faster. It can be slow and painful, but is effective as a last resort. Fig 6 summarizes these hints. Next we consider ways to find bugs that do not employ inference.

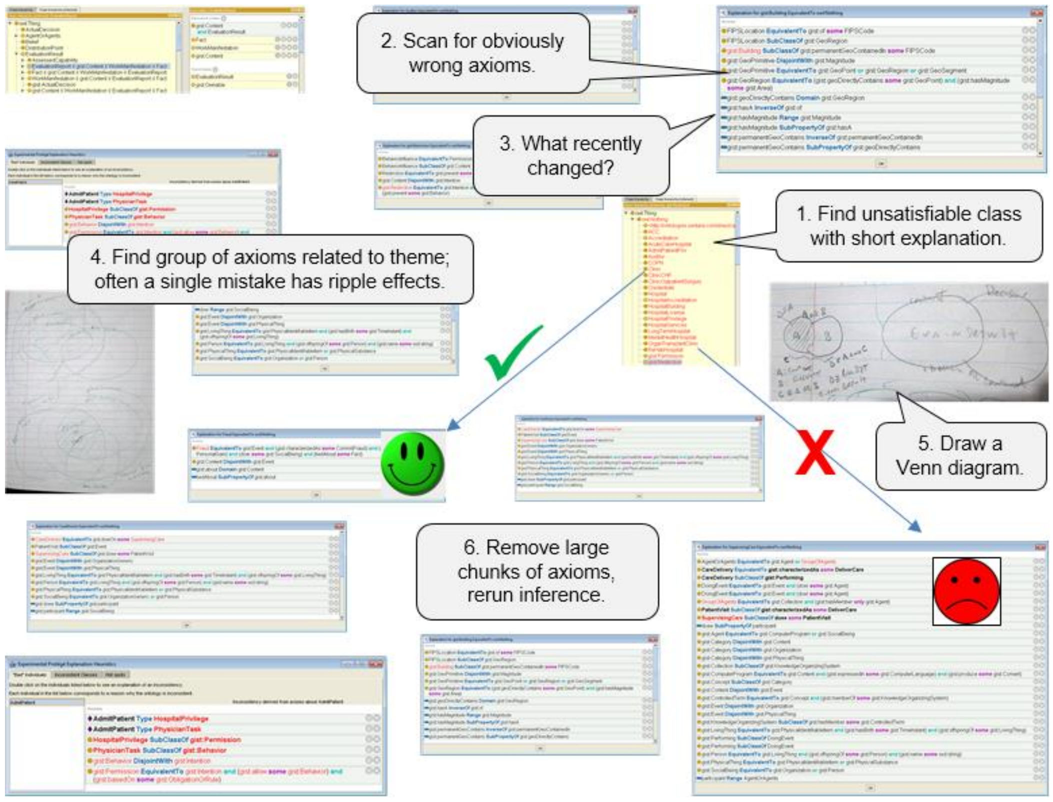
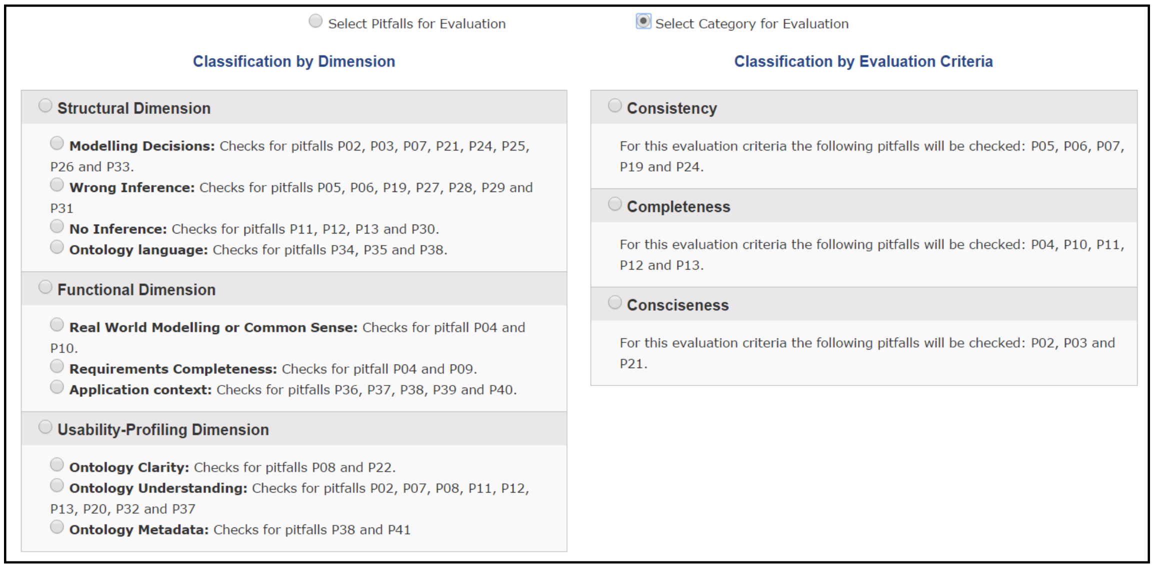

2. Scan for obviously wrong axioms.
3. What recently changed?
1. Find unsatisfiable class with short explanation.
4. Find group of axioms related to theme; often a single mistake has ripple effects.
5. Draw a Venn diagram.
6. Remove large chunks of axioms, rerun inference.

Fig. 6. Debugging Hints Using Explanations

Other ways to find bugs.

Although much has been written about what constitutes a 'good ontology' [Obrst et al 07; Duque-Ramos 11], there are no standard guidelines, nor are there any standard ontology compilers that will produce a wide variety of standard warnings. An important step in that direction is an online tool called the Ontology Pitfall Scanner (OOPS!) [Poveda-Villalón et al 2014]. It has a catalog of several dozen pitfalls, which are measurable aspects of the ontology that are potentially undesirable. You can select just the ones you care about, avoiding needless warnings (Fig. 7). If you have your own pitfalls that you wish to avoid, you can write SPARQL queries to catch them. Often it is a matter of preference and/or agreement among a group. Identifying and eliminating all the pitfalls ensures good hygiene for the ontology. For example, the Financial Industry Business Ontology (FIBO) [Bennet 2013] is identifying a set of conventions that must be adhered to before ontologies are published. The idea is to write SPARQL queries to catch non-conformance, and to embed them in scripts that are run before a GitHub commit is accepted.

Exactly what is deemed to be pitfall (the opposite of good hygiene) is often highly subjective. There are some desiderata that are generally agreed upon, but they are often too high-level to be actionable. Ironically, the more actionable the guideline, the more likely it is that there will be disagreement about it. Sometimes there are two right ways to do something. If the ontology is being created by a single individual, then that is what we call 'designers choice'. It matters less whether to accept any particular guideline; the important thing is to choose and use a set of guidelines consistently throughout the ontology.


Select Pitfalls for Evaluation [✓ Select Category for Evaluation]

Classification by Dimension

Structural Dimension

Modelling Decisions: Checks for pitfalls P02, P03, P07, P21, P24, P25, P26 and P33.
Wrong Inference: Checks for pitfalls P05, P06, P19, P27, P28, P29 and P31
No Inference: Checks for pitfalls P11, P12, P13 and P30.
Ontology language: Checks for pitfalls P34, P35 and P38.

Functional Dimension

Real World Modelling or Common Sense: Checks for pitfall P04 and P10.
Requirements Completeness: Checks for pitfall P04 and P09.
Application context: Checks for pitfalls P36, P37, P38, P39 and P40.

Usability-Profiling Dimension

Ontology Clarity: Checks for pitfalls P08 and P22.
Ontology Understanding: Checks for pitfalls P02, P07, P08, P11, P12, P13, P20, P32 and P37
Ontology Metadata: Checks for pitfalls P38 and P41

Classification by Evaluation Criteria

Consistency

For this evaluation criteria the following pitfalls will be checked: P05, P06, P07, P19 and P24.

Completeness

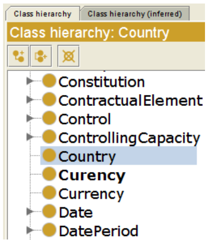
For this evaluation criteria the following pitfalls will be checked: P04, P10, P11, P12 and P13.

Consciseness

For this evaluation criteria the following pitfalls will be checked: P02, P03 and P21.

Fig. 7. Selecting pitfalls for OOPS! to scan

Another effective technique for spotting simple problems is to scan an alphabetized list of all the classes and properties. You will often find spelling errors (Fig 7). Depending on what conventions you want to have for your ontology, you may wish to create some SPARQL queries to detect non-adherence. For example, the Financial Industry Business Ontology, by convention, uses `skos:definition` rather than `rdfs:comment` to specify text definitions for ontology concepts. Every concept must have a definition, and `rdfs:comment` is not meant to be used for anything. This gives rise to two simple SPARQL queries, one to find any concepts that do not have a `skos:definition` defined, and the other to find any uses of `rdfs:comment`. You may have a policy about inverse properties that can also be checked using SPARQL.



Constitution
ContractualElement
Control
ControllingCapacity
Country
Curency
Currency
Date
DatePeriod

Fig. 8. Finding Spelling Errors

After performing all these checks, you should be able to rid the ontology of all the obvious mistakes. Next we consider how to avoid bugs in the first place.

Avoiding bugs .

As noted above, inference will only catch a modest percentage of the bugs. If you find an error that the inference engine did not find, it means the inference engine did not have enough information to go on. You don't want to keep adding more and more axioms, or else the ontology will become unwieldy. However, there are a few things that are worth doing so that inference will catch more errors:

- High-level disjoints,
- Careful use of domain and range.

High-Level Disjoints .

Recall the inconsistent ontology example in Fig. 2. The problem was that the individual `AdmitPatient` was a member of two disjoint classes. Fig 9 shows what was going on in more detail. The idea is to have a nicely structured hierarchy with relatively few classes that are mostly disjoint from one another. You need only say that `Behavior` is disjoint from

`Intention` for it to follow that no `PhysicianTask` can be an `Intention`, nor can any `Permission` or `HospitalPrivelege` be a `PhysicianTask` or `Behavior`. If the hierarchy was completely flat, then for N classes, you would have O(N**2) pairs of classes whose disjointness needs to be considered independently. 5*(5-1) / 2 =10 in this case. But with this hierarchy, we need only specify one disjointness axiom. This is an important pattern used in the development of gist [McComb 2006]. It can catch many errors that are otherwise hard to find.

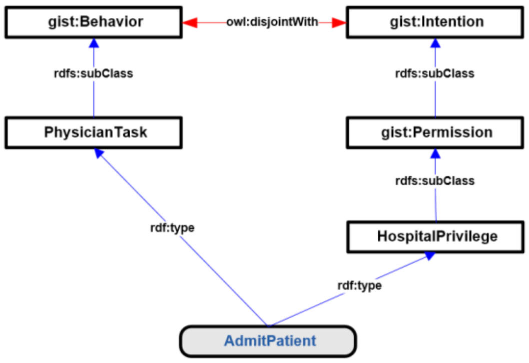


Fig. 9. High Level Disjoints

Domain and Range .
Sometimes just specifying disjoint classes is not sufficient to catch an error. In the above example, `AdmitPatient` was manually asserted to be an instance of two [inferred to be] disjoint classes. However, what if membership was only asserted into the class, `HospitalPrivilege`? In addition, let's say there is a triple: [`AdmitPatient hasPerformer JaneJones`], and that `hasPerformer` has domain equal to `Behavior`. Now `AdmitPatient` is inferred into the class `Behavior` which is disjoint from `HospitalPrivilege` as in the above example. It turns out that the combination of high-level disjoints and domain and/or range is extremely powerful. Indeed, our experience suggests that virtually all of the errors caught by the inference engine are due to one or both of these.

You must be careful when using domain and range. All too often ontology designers specify the domain or range of a property too narrowly. For example, the W3C Media Ontology[1] has many generic-sounding properties that can only be used for a `MediaResource` (e.g. `hasPolicy` and `hasLanguage`). That means if you want to attach a language to a country, you have to make up a new property that means essentially the same thing. Lots of things other than media resources have policies. This over-specialization is a real barrier to reuse of standard ontologies. A similar example from another ontology being developed as a standard is a property called `hasCount` with a domain of `Schedule`. Its intended use was to count the number of entries in a `Schedule`. This is fine, until you decide you want to count something else, like the number of entries in a report. This property can be very broadly used, so it probably did not need a domain in the first place. Below we learn that this is not always so easy to fix.

Namespaces.
There is another common source of bugs that can be avoided, this time having nothing to do with inference. Currently, it is very common practice to liberally create new namespaces, in many cases, one for every ontology. For example, a highly modular ontology might have several dozen ontology modules. Even though all of the ontologies are modeling the same subject area, a different namespace is introduced for each sub-topic modeled as a separate ontology. This causes problems. Why? First, it is time-consuming and error-prone to be checking all the time to ensure the correct namespace is being used. But more importantly, if you have dozens of different namespaces and decide you want to do some major refactoring, it becomes a big headache for you and your users. You have to change the URIs of many different entities, which ideally you should never do.[2] Also, if someone else was relying on that URI, their models, queries, and/or applications can suddenly break.

[1] https://www.w3.org/TR/mediaont-10/. Accessed on 2016-03-2.
[2] Avoiding a Semantic Web Roadblock: URI Management and Ontology Evolution; http://www.slideshare.net/UscholdM/uschold-michael-semanticwebroadblock

If all the ontologies for the same major topic use the same namespace, it is a trivial matter to move the definition from one ontology to the other. No need to change all the references to that URI that are scattered across dozens of other ontologies. The original idea of namespaces is to solve collision problems, such as *bank* the financial institution vs. a *bank* in the road or a *bank* along the side of a river. These are three entirely different concepts in three different subjects, so three different namespaces are warranted. There is little to be gained by having multiple namespaces in the same topic, and a lot to lose. The exception is if the sub-topic is under the governance of a different person or group. In that case, it works best if they can mint their own URIs.

By contrast with coding, where there are IDEs that make re-factoring relatively easy and reliable, tool support for refactoring ontologies requires a lot of manual error-prone work. If you are lucky, this is the only damage caused by this namespace proliferation. In the worst case, important refactoring is so difficult and unreliable, that you are locked into harmful design choices. Consider the irony. This namespace proliferation is re-creating the inflexible situation when building an ontology that ontologies are supposed to solve in the broader IT infrastructure in an enterprise.

Let's see a walk through the `hasCount` example mentioned above. In principle, the fix is straight-forward: just remove the domain. In practice, because there is a network of importing ontologies, we have to be careful (see Fig. 10). There are a few ontologies: core properties, agents, dates and schedules, contracts and loans. The `hasCount` property was conceived to be relevant only in the dates-and-schedules ontology. However, it is a perfectly general property that should be available to many ontologies that are not interested in dates. So `hasCount` belongs in the core properties ontology. This has ripple effects that must be addressed, mainly that the URI for `date:hasCount` has been changed to `prop:hasCount`. Therefore, all references to the original URI must be found and changed. These include references not only in the dates-and-schedules ontology, but also references in any ontology that imports the dates-and-schedules ontology. In addition, there may be ontologies that refer to `date:hasCount` without importing the ontology. So all the ontologies being developed and maintained must be checked. How will you minimize their disruption? This is one trivial change, moving a single URI to a different ontology. What if there was some major refactoring to be done where whole chunks of axioms were being moved around and rationalized? For example, maybe the schedules portion of the dates-and-schedules ontology grows so that it should be a separate ontology. This would usually entail creating yet another namespace, and changing the URI from, say, `date:Schedule` to `sched:Schedule`. You can see how much work this is, and how easy it is to get it wrong. The ontologies in Fig. 10 with the dotted borders are the ones that need to be checked for possible URI changes (all of them!). If the ontology is being used by others, you may have no way of knowing about it. This involves having a mechanism for informing the user community of new versions with release notes and update instructions.

Contrast this with a convention where you use just a single namespace: `fin:`, for the finance domain (Fig. 11). You still have to move the property into the new ontology, but there is no need to change any of the URIs. This is dramatically easier, less error-prone, and not a barrier to cleaning up the ontology.

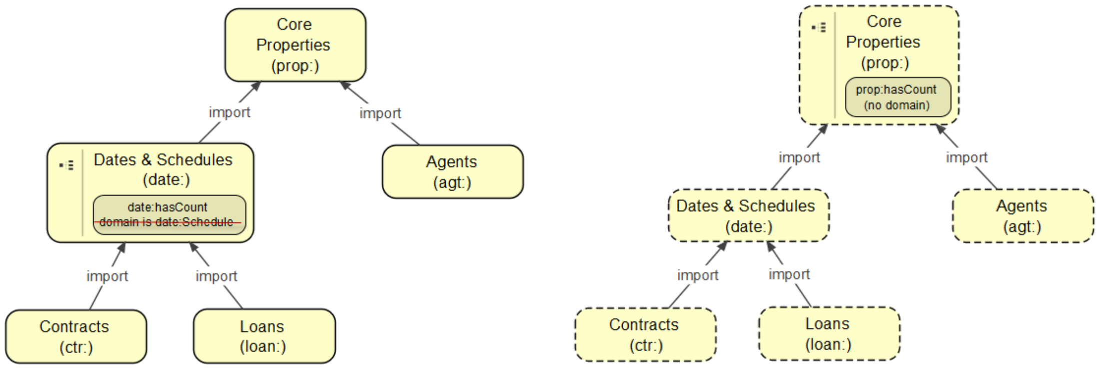


Fig. 10. Refactoring with Multiple Namespaces. Dotted ontology modules need to be checked.

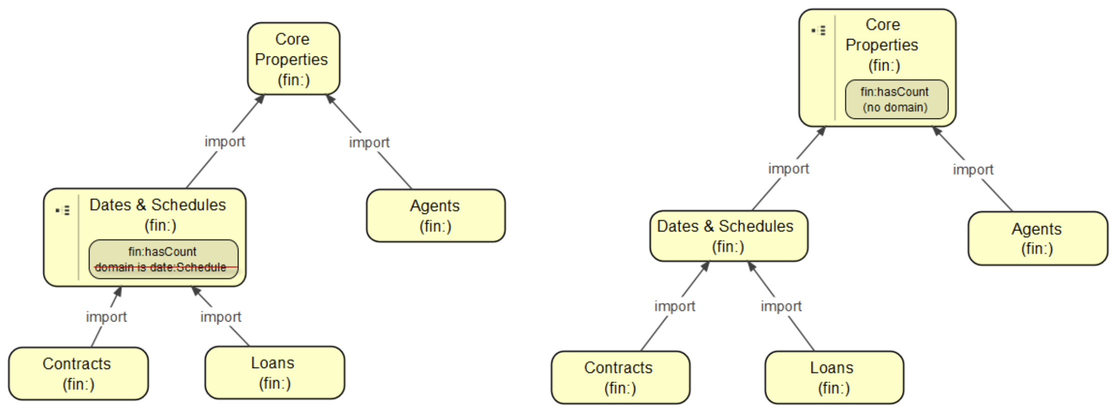


Fig. 11. Refactoring with a Single Namespace. None of the ontologies need to be checked.

Property proliferation.

The astute reader may have noticed that removing the domain from `date:hasCount` actually changes the semantics. In the event that the change in semantics could cause a problem, the least amount of disruption will be to introduce a more general property, `prop:hasCount`, in the core properties ontology and make it a super-property of `date:hasCount`. This works, but is not a very elegant solution. You now have two different properties whose core meaning is the same. It's very unlikely that users of the original property relied on the axiom requiring that the subject of any triple using the property must be a `date:Schedule`. Far more likely, they would see the name 'hasCount' and think that it applied to other things, like the count of how may stocks were in a portfolio. Any portfolio instance using that property would be inferred to be a Schedule, which should be disjoint from a Portfolio and an error would be raised. Generally speaking, you should not introduce two different properties when the meaning is essentially the same. In this case, only the domain was different.

A more common situation where properties proliferate is when the only difference in meaning is the range. Consider a series of properties like: `hasWheel`, `hasEngine` & `hasWindshield` which are sub-properties of `hasPart`. There is no reason to do this, and there is a really good reason to not do so: it causes bugs downstream. Why? Because repeated use of this pattern will multiply the number of properties. There will be so many that finding the right one will not be very easy, and mistakes will be made. This will cause bugs in downstream applications. A better solution is to keep the number of properties to a minimum. The ontology will be much easier to understand and use, which in turn, will result in fewer bugs in downstream applications.

If you really want to assert that say a Car has 4 wheels, an engine, and a windshield, you can use using qualified cardinality to make Car a subclass of the restrictions: [`hasPart exactly 4 Wheel`], [`hasPart exactly 1 Engine`], [`hasPart exactly 1 Windshield`]. This, in effect, specifies the range locally to the class Car.

Fidelity to the Subject Matter.

After a systematic pass to identify all the mistakes, it is necessary to get to more fundamental question: does the ontology faithfully represent the subject matter? Test data helps to do this, but it is also important to have the ontology vetted by subject matter experts and stakeholders. There are countless ways to do this, and it is often dictated by the preferences of the client. It can involve a series of meetings where different part of the ontology are presented as the ontology evolves. We have experimented with many kinds of visualization aids. Clients are getting more and more sophisticated – it is no longer surprising for them to want to just look at the owl in their favorite ontology editor. This phase is where understandability is critically important.

2.2 Understandability & Usability

Per the example above, it is important to keep the total number of concepts to a minimum to help ensure that the ontology is easy to understand and use. A good rule is to make sure that each new concept is importantly different from every other concept from the perspective of the intended uses of the ontology. Having a large number of concepts with very fine distinctions makes it difficult for an individual to find the right concept for a particular purpose. What are they to do if there are four choices that all seem to be ok? They might choose one on one day, and two weeks later forget about it and inadvertently choose a different one of the four options. All bets are off if there are different individuals using the ontology for the same project, as it is a statistical certainty that usage will not be consistent among the individuals.

In a collaborative ontology development project, the ability to use inference to check consistency can be very important. The more people involved, the more potential there is for error and the more need there is for careful coordination. Similar problems should be solved using similar modeling patterns. It is not unusual for there to be two valid ways to model something, so choose one and stick with it. Naming conventions should be established and adhered to. Such consistency helps in two ways. First, it makes it easier for new ontologists coming on board the project to understand the ontology, so they can work with it correctly making changes and additions. Secondly, end users will find it easier to understand, which reduces bugs in downstream applications.

Another barrier to understandability is redundancy. Why is something there twice? Is it a mistake, or is it intentional? This can cause confusion and hinder progress. Even aside from understandably, redundancy can be a source of bugs in its own right. Why? If something is defined in more than one place or something is directly asserted that is already easy to infer, then if something changes that no longer warrants that inference, it is likely that the direct assertion needs to be removed. This is error prone. The general rule is that having redundancy makes something harder to maintain.

However, this is not always the case. Consider the following example relating to ontology import. Is it ever acceptable, or a positively good idea, to have O1 explicitly import both O2 and O3, when O2 already imports O3? O1 only needs to import O2, and O3 gets imported because import is transitive. As the ontology evolves, someone may change which ontologies O2 imports, and it no longer imports O3. Another ontology that needs O3 and only imports O2 will no longer be importing O3. This creates a maintenance overhead. This can be avoided if the logically redundant import was explicit. No harm, logically and no impact to performance. An alternative is to have a maintenance discipline whereby whenever imports are changed in any ontology, then the ripple effects should be examined, and updated accordingly. For complex networks of ontologies being maintained by many people, this may be difficult in practice. There may not be one right way that covers all circumstances. There are tradeoffs and reasonable people disagree on these things.

There is a similar tradeoff for range and restriction filters, if something changes elsewhere, then there are ripple effects that need to be checked for, thus increasing maintenance work. For example, if the range of a property is `Currency` and you use that property in a qualified cardinality restriction, then specifying the filter class to be `Currency` is strictly redundant. However, it can be handy for communicating to the person looking at the model because they don't have to look up the range of that property. Or is it better to remove redundancy and put that information in a comment? Again, reasonable people may disagree.

A problem could arise if someone changes the domain of the property to be something broader. Then, in order to make sure things are okay, you would have to check for every restriction using that property with unqualified `min 1` cardinality to see if it needs to be updated to be the more specific class that was the original range of the property. Usually, redundancy incurs higher maintenance costs. In this case, there may be a tradeoff, where redundancy potentially can reduce maintenance errors. Again there is no one right way.

Terminology is also important for understandability. Different parts of a company use terms differently. Sometimes term usage is subtly influenced by the terminology of a product or application. We use the following guideline when choosing terms: a term should be readily understood by anyone in the enterprise who needs to know what is going on and who is unfamiliar with terms tied to specific applications. This mostly corresponds to standard industry terms, which are common across different enterprises in the same industry. Another guideline is to err in favor of having long terms that convey meaning in preference to shorter ones that are ambiguous (e.g. `FormallyConstitutedOrganization`).

2.3 Completeness

Completeness means that the ontology includes everything that is in scope. For an enterprise ontology, this is more difficult to ascertain than when building an ontology for a highly specific use in some corner of an organization. The guideline is to have all and only concepts that (1) are central to the enterprise, (2) are stable over long periods of time, and (3) are substantially different from each other. This is the starting point for building the enterprise ontology. When it is starting to take shape and seems reasonable complete, there are additional steps to aid completeness. You should identify the kinds of things that are hard to do now that the enterprise ontology should be able to support. Also, identify ideas for future prototypes. Flesh out specific use cases and go through them with the stakeholders. Focus on the inputs and outputs, because these tell you about the data requirements. Does the ontology have concepts to correctly represent the data? It does not have to have the exact detail, just a more general class or property that can be specialized.

3 Test Data

It is surprisingly easy to convince yourself that the ontology is okay. But until you encode some test data as triples, it is, at best, educated speculation. Real data is where the rubber hits the road. Only by looking at detailed examples that populate the core elements and relationships in the ontology can you be sure that it is up for the job. It can help find problems related to correctness, completeness, and understandability.

Correctness: When you create a bunch of triples to model real-world examples, you will often find errors. For example, say you have a restriction that says you need to have at least one connection, and then you find an example where it does not apply. So you remove the restriction or change it so it has no logical force (e.g. making it `min 0` cardinality). You may find that a subclass relationship does not hold like you thought it did: the two classes are, in fact, overlapping.

When debugging an ontology, it is possible to not notice an unsatisfiable class. The ontology is consistent so long as the class has no members. Using test data will help surface these problems.

Completeness: Test data is an important check for completeness. This will not be optional if the purpose of the ontology is to integrate databases – it will be your main focus. You may be required to convert a relational database into triples, either by writing scripts or using R2RML[3]. Start with a small ontology scoped to an initial subset of data. Then in an agile manner, iteratively extend the ontology to include more and more data. This process usually uncovers bugs, as noted above, but also discovers things that are missing.

Understandability: Don't go overboard on completeness, adding lots of things "just in case". If you do that, you get cruft which hinders understandability. If in doubt, leave it out. This helps keep the ontology small. Creating test triples also is important for to help others understand the ontology. Often the fastest way to get up to speed on using an ontology is to see examples of it in use. "No one reads long specifications. Most developers tend to copy and edit examples. So, the documentation is more like a set of recipes and less like a specification." [Guha 2016]. Seeing how an ontology is actually used is hugely important for getting ontologies in more widespread use. Without sufficient aids to understanding an ontology, future users will either not bother, or they will start using it incorrectly. So there will be bugs in semantic web applications that are not due to the ontology itself, but to the fact that it was hard to understand and use.

4 Related Work

To our knowledge, the most comprehensive work that is directly applicable to industry is OOPS, as discussed above [Poveda-Villalón et al 2014]. There has been a lot of work on ontology evaluation, mostly from an academic or scientific perspective. Each contributes something important, yet falls short in one of two ways. The first shortcoming is not being based on extensive commercial experience aimed at the needs of industry. If the semantic web and linked data are going to have more than a minor impact, it is essential to focus on the enterprise. Second, they tend to report on the technical details of one or more particular techniques, and are not attempting to take a comprehensive look at improving ontologies more generally. For example, [Köhler 06] reports on automated techniques for detecting circularity and intelligibility in text definitions. This would certainly be of value for many enterprise ontology development projects, but these things are not readily available for use as packaged software or plugins to standard ontology tools. This particular work is also focused on terms rather than the axioms that formally define the concepts that the terms are naming. In a terminology, circularity is a major flaw. But in an ontology, the term is just a name for a concept; the formal definition is what matters. If URIs are used that are not human-readable, then circularity will not even arise. Changing the URI of the concept does not suddenly render a definition circular. Competency questions are another way to identify scope and ensure completeness [Gruninger 1995; Uschold 1996]. Interestingly, we have not found them to feature largely when building an enterprise ontology. Perhaps it is because competency questions are appropriate for more specialized ontologies whose purposes are more specific than an enterprise ontology.

[3] R2RML: RDB to RDF Mapping Language (https://www.w3.org/TR/r2rml/)

5 Summary and Conclusions

We described the distillation of a decades' worth of experience in building enterprise ontologies for commercial use. Throughout, we illustrate our findings with real-world examples drawn from our commercial work. There are no made up examples just to illustrate a point. The key properties that an ontology should have are *correctness, understandability, usability, and completeness.* Inference is a very important aid, and may be able to catch up to 30% of the bugs in an ontology. We suggest various ways to help the inference engine catch more bugs. You need to judiciously add more axioms. The most powerful is the use of high level disjoints in combination with cautious use of domain and range. There are a variety of other techniques that can be used that do not involve inference. For example, you can identify what constitutes good hygiene for you are ontology and enforce it with SPARQL queries. The simple practice of looking at an alphabetized list of classes and properties can identify spelling errors. Populating the ontology with real world data is an important way to drive out bugs and establish completeness. Overall there is a mix of automated, semi-automated and fully manual techniques to deploy.

Three common practices in the ontology development community are causing problems and we recommend that people change. These are:

1. proliferation of namespaces cause the very problem that ontologies are designed to solve (inflexibility)
2. proliferation of properties that mean essentially the same thing makes an ontology hard to use, and will tend to create bugs in downstream applications
3. overly restrictive use of domain and range reduces reusability, which is the whole point of an ontology

Although our focus has been on enterprise ontology, most of our findings are relevant of other ontology development efforts. One area which is different is how to identify scope and when to identify use cases. Competency questions do not seem to play as central a role for enterprise ontology development than for more targeted ontologies.

We place very high importance on usability which is hugely dependent on understandability. The major way to keep an ontology understandable is to keep it small and to adopt conventions and use them consistently.

Another important development in the semantic web that helps ensure bug-free applications is SHACL. You can keep the ontology clean, separate from the concerns of the application, and at the same time give specific requirements to UI developers on how to use the ontology in a given application.

Acknowledgements

I am grateful to Dave McComb for providing the opportunity to work on so many commercial enterprise ontology projects and for many ideas on how best to do that. I am grateful to Dan Carey and Dave McComb for comments on a prior draft of this paper.

References

[Poveda-Villalón et al 2014] Poveda-Villalón, M., Gómez-Pérez, A., & Suárez-Figueroa, M. C. (2014). Oops!(ontology pitfall scanner!): An on-line tool for ontology evaluation. International Journal on Semantic Web and Information Systems (IJSWIS),10(2), 7-34. http://oops.linkeddata.es/

[Obrst *et al* 07] Obrst, L., Ceusters, W., Mani, I., Ray, S., & Smith, B. (2007). The evaluation of ontologies. In Semantic Web (pp. 139-158). Springer US.

[Duque-Ramos 11] Duque-Ramos, A., Fernández-Breis, J. T., Stevens, R., & Aussenac-Gilles, N. (2011). OQuaRE: A SQuaRE-based approach for evaluating the quality of ontologies. *Journal of Research and Practice in Information Technology*, 43(2), 159.

[Köhler 06] Köhler, J., Munn, K., Rüegg, A., Skusa, A., & Smith, B. (2006). Quality control for terms and definitions in ontologies and taxonomies. *BMC bioinformatics*, 7(1), 212.

[Guha 2016] Guha, R. V., Brickley, D., & Macbeth, S. (2016). Schema. org: evolution of structured data on the web. *Communications of the ACM*, 59(2), 44-51.

[Gruninger 1995] Grüninger, M., & Fox, M. S. (1995). The role of competency questions in enterprise engineering. In *Benchmarking—Theory and Practice* (pp. 22-31). Springer US.

[Uschold 1996] Uschold, M. (1996) Building Ontologies: Towards a Unified Methodology. In *Proceedings of the 16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems*

[Bennet 2013] Bennett, M. (2013). The financial industry business ontology: Best practice for big data. *Journal of Banking Regulation*, 14(3), 255-268.

[McComb 2006] McComb, D. (2006, March). Gist A Minimalist Upper Ontology. In *Semantic Technology Conference*. http://semanticarts.com/gist/

[Vrandečić 2009] Vrandečić, D. (2009). *Ontology evaluation* (pp. 293-313). Springer Berlin Heidelberg.

[Duque-Ramos 2011] Duque-Ramos, A., Fernández-Breis, J. T., Stevens, R., & Aussenac-Gilles, N. (2011). OQuaRE: A SQuaRE-based approach for evaluating the quality of ontologies. *Journal of Research and Practice in Information Technology*, 43(2), 159.