

Debugging OWL-DL Ontologies: A Heuristic Approach

Hai Wang Matthew Horridge Alan Rector
Nick Drummond Julian Seidenberg

Department of Computer Science,
The University of Manchester,
Manchester M13 9PL, UK
{hwang,mhorridge,rector,ndrummond,jms}@cs.man.ac.uk

Abstract. After becoming a W3C Recommendation, OWL is becoming increasingly widely accepted and used. However most people still find it difficult to create and use OWL ontologies. One major difficulty is “debugging” the ontologies - discovering why a reasoner has inferred that a class is “unsatisfiable” (inconsistent). Even for people who do understand OWL and the logical meaning of the underlining description logic, discovering why concepts are unsatisfiable can be difficult. Most modern tableaux reasoners do not provide any explanation as to why the classes are unsatisfiable. This paper presents a ‘black boxed’ heuristic approach based on identifying common errors and inferences.

1 Introduction

One of the advantages of logic based ontology languages, such as OWL, in particular OWL-DL or OWL-Lite, is that reasoners can be used to compute subsumption relationships between classes and to identify unsatisfiable (inconsistent) classes. With the maturation of tableaux algorithm based DL reasoners, such as **Racer** [11], **FaCT** [8], **FaCT++** [7] and **PELLET** [4], it is possible to perform efficient reasoning on large ontologies formulated in expressive description logics.

However, when checking satisfiability (consistency) most modern description logic reasoners can only provide lists of unsatisfiable classes. They offer no further explanation for their unsatisfiability. The process of “debugging” an ontology - i.e. determining why classes are unsatisfiable - is left for the user. When faced with several unsatisfiable classes in a moderately large ontology, even expert ontology engineers can find it difficult to work out the underlying error. This is a general problem which gets worse rather than better with improvements in DL reasoners; the more powerful the reasoner the greater its capacity to make non-obvious inferences.

Debugging an ontology is a non-trivial task because:

- *Inferences can be indirect and non-local.* Axioms can have wide-ranging effects which are hard to predict.
- *Unsatisfiability propagates.* Therefore, a single root error can cause many classes to be marked as unsatisfiable. Identifying the root error from amongst the mass of unsatisfiable classes is difficult.

2 A Heuristic Approach to Ontology Debugging

In short, the current state of ontology development environments and reasoning services within these environments is akin to having a programming language compiler detect an error in a program, without explaining the location of the error in the source code.

Over the past five years we have presented a series of tutorials, workshops and post-graduate modules on OWL-DL and its predecessors. Based on our experience, a list of frequently made errors have been identified as reported in [10]. This catalogue of common errors has been used in turn to develop a set of heuristics that have been incorporated into debugging tool for Protégé-OWL [5]. The examples in this paper are all taken from these tutorials and use the domain of Pizzas used in the introductory tutorial.

The heuristic debugger treats the tableaux reasoner as a ‘black box’ or ‘oracle’. This ‘black box’ approach has the advantage that it is independent of the particular reasoner used. It works with any DIG [1] compliant reasoner, even ones which have been specially augmented or adapted.¹

Being independent of the reasoner has advantages even if only as single reasoner is to be used. Many modern reasoners transform the input ontology in order to optimise the reasoning process. Although logically equivalent, the internal representation may bear little resemblance to the ontology as it was constructed by the user. Given such transformations, even it were possible for the reasoner to ‘explain’ its actions, the explanation in terms of the transformed ontology would be unlikely to be of direct use to the user. An additional advantage of the ‘black box’ approach is that it is independent of such transformations.

3 Background

3.1 OWL overview

OWL [2] is the latest standard in ontology languages, which was developed by members of the World Wide Web Consortium² and Description Logic community.

An OWL ontology consists of classes, properties and individuals. Classes are interpreted as sets of objects that represent the individuals in the domain of discourse. Properties are binary relations that link individuals and are represented as sets of ordered pairs that are subsets the cross product of the set of objects.

OWL classes fall into two main categories – named classes and anonymous (unnamed) classes. Anonymous (unnamed) classes are formed from logical descriptions. They contain the individuals that satisfy the logical description. Anonymous classes may be sub-divided into restrictions and ‘logical class expressions’. Restrictions act along properties, describing sets of individuals in terms of the types of relationships that the individuals participate in.

¹ The DIG Interface is a standard DL reasoner communication protocol that sits between DL based applications and DL reasoners, thereby allowing these applications to communicate with different third party DL reasoners.

² <http://www.w3.org>

Logical classes are constructed from other classes using the boolean operators AND (\sqcap), OR (\sqcup) and NOT (\neg).

An important point to note from the point of view of debugging is that OWL allows the nesting of anonymous class expressions to arbitrary levels. For example, the expression:

$\text{Pizza} \sqcap \exists \text{hasTopping} (\text{PizzaTopping} \sqcap \exists \text{hasIngredient} (\text{SpicyIngredient} \sqcap \exists \text{hasColour} \text{RedColour}))$ describes the individuals that are pizzas that have pizza toppings that have ingredients which are spicy ingredients that are coloured red.

Disjoint Axioms All OWL classes are assumed to overlap unless it is otherwise stated that they do not. To specify that two classes do not overlap they must be stated to be *disjoint* with each other using a *disjoint axiom*. The use (and misuse) of disjoint axioms is one of the primary causes of unexpected classification results and inconsistencies [10]. Disjoint axioms are ‘inherited’ by their subclasses. For example if *Pizza* is disjoint from *PizzaTopping* then all subclasses of *Pizza* will be disjoint from all subclasses of *PizzaTopping*. This can make debugging difficult for ontologies that have deep taxonomical hierarchies.

Describing Classes Named OWL classes are described in terms of their named and anonymous super classes, equivalent classes and disjoint classes. When a restriction is added to a named class, it manifests itself as an anonymous superclass of the named class.

For example the named class *SpicyPizzaTopping* might have a named superclass called *PizzaTopping* and also the anonymous super class $\exists \text{hasSpicyIngredient} \text{SpicyIngredient}$. That is, things that are *SpicyPizzaToppings* are also *PizzaToppings* and things that have at least one *SpicyIngredient*. We refer to these super classes as *conditions*, as they specify the conditions for membership of a given class.

In summary, OWL has three types of class axioms:

- **Subclass axioms** – These axioms represent *necessary* conditions.
- **Equivalent class axioms** – These axioms represent *necessary & sufficient* conditions.
- **Disjoint axioms** – These axioms represent additional *necessary* conditions.

Domain and Range Axioms OWL also allows ‘global’ axioms to be put on properties. In particular, the *domain* and *range* can be specified for properties. In many other languages, domain and range are commonly used as constraints that are checked and generate warnings or errors if violated. Hence domain and range constraints can be used in inference and are a potential cause of unsatisfiability.

3.2 Unsatisfiable OWL Classes

An OWL class is deemed to be unsatisfiable (inconsistent) if, because of its description, it cannot possibly have any instances. While there are many different ways in which the axioms in an ontology can cause a class to be unsatisfiable, the key observation in

heuristic debugging is that there are limited number of root causes for the unsatisfiability.

In general, there are three categories of causes.

Local unsatisfiability The combination of directly asserted restrictions and named superclasses are unsatisfiable.

Propagated unsatisfiability The combination of directly asserted restrictions and named superclasses would be satisfiable except that some class used in them is unsatisfiable.

Global unsatisfiability There is some global constraint, usually a domain or range constraint, from which along with other information in the ontology it can be inferred that the class is unsatisfiable.

Local unsatisfiability is usually easy to spot. Section 4.5 describes the various reasons that may lead to a class being locally unsatisfiable. Propagated unsatisfiability is more difficult. There are two primary mechanisms for propagation:

Unsatisfiable ancestor classes All descendant classes of an unsatisfiable class are unsatisfiable. Therefore unsatisfiability propagates down the subclass hierarchy.

Unsatisfiable fillers of existential restrictions Any existential (or minimum cardinality) restriction with an unsatisfiable filler is itself unsatisfiable.

A single error can cause large swathes of the ontology to be unsatisfiable. The key strategy of the heuristic debugger is to collect all global conditions so that they can be treated as local.

4 Heuristic debugging process

Figure 1 gives a principled view of the heuristic debugging process; in practice this is optimised.

- Check that the selected class is indeed unsatisfiable
- Determine the *basic debugging necessary conditions*
- Identify the *unsatisfiable core*, or smallest set of unsatisfiable subset of the *basic debugging necessary conditions*
- Generate the *debugging super conditions*, which are the conditions that are implied by the conditions in the *unsatisfiable core*.
- Determine the *most general conflicting* class set based on the *unsatisfiable core*.
- Analyse the most general conflict in order to produce an explanation of why the class is unsatisfiable.

Each step is examined in detail below.

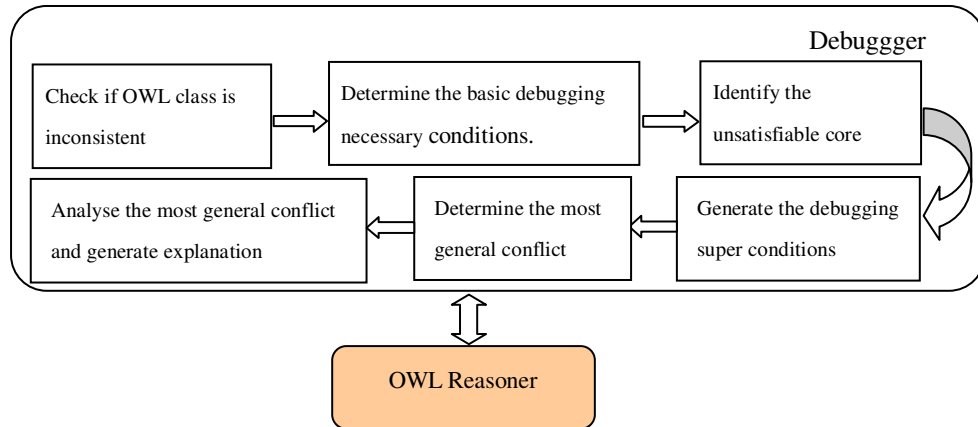


Fig. 1. The debugging process

4.1 Determining the *Basic debugging necessary conditions (BDNC)*

As discussed in section 3.1 OWL uses three kinds of class axioms:

- **Subclass axioms** – *necessary* conditions.
- **Equivalent class axioms** – *necessary & sufficient* conditions.
- **Disjoint axioms** – *necessary* conditions.

An OWL class is unsatisfiable if and only if a subset of the above conditions, which we refer to as the *basic debugging necessary conditions* is unsatisfiable. The first step of the debugging process is the generation the '*basic debugging necessary conditions*'. This is achieved by collecting together the *necessary*, and *necessary & sufficient* conditions of the class that is being debugged, and then adding a condition for each class that the given class is disjoint with, which represents the complement class of each disjoint class. For example, suppose the class in question was disjoint with class D . The condition $\neg D$ would be added to the set of basic debugging necessary conditions.

4.2 Identifying the Unsatisfiable core

After obtaining the set of basic debugging necessary conditions, they are refined and reduced to obtain the *unsatisfiable core*. The *unsatisfiable core* is the smallest unsatisfiable subset of the basic debugging necessary conditions. The *unsatisfiable core* is defined as follows:

Definition 1. Let $BDNC(C)$ be the '*basic debugging necessary conditions*' of a unsatisfiable Class C . An **unsatisfiable core** ($UC(C)$) of the class C is a set of OWL class descriptions, such that:

1. $UC(C) \subseteq BDNC(C)$
2. Intersection of all the concepts belonging to $UC(C)$ is unsatisfiable.
3. For every set of class descriptions CD :
 $CD \subset UC(C) \Rightarrow$ Intersection of all the concepts belonging to CD is satisfiable
 $CD = \emptyset$

Condition 3 ensures that an *unsatisfiable core* is the most minimal possible set of conditions. An unsatisfiable class could have more than one *unsatisfiable core*, in which case the first is analysed.

4.3 Generating the Debugging Super Conditions

The *unsatisfiable core* merely identifies the set of axioms which have resulted in the inconsistency. However, as described above, the inconsistency may have been caused by global conditions (section 3.1). The debugging process, therefore, ‘collects’ global axioms – primarily domain/range and disjoint axioms – and maps them into local axioms – i.e. sets of necessary conditions. These are the *debugging super conditions*. The set of *debugging super conditions* is expanded by recursive application of the rules in Figure 2, the most important of which are explained below.

Debugging Super Condition Generation Rules

Named class rule (Rule 1): If an OWL named class C_1 is added to the debugging super conditions, all its direct super classes are also added to the debugging super conditions. For each OWL class C_2 which is asserted to be disjoint with C_1 , $\neg C_2$ will be added to the debugging super conditions.

Complement class rule (Rule 2): If an OWL complement class $\neg C_1$ is added to the debugging super conditions, it will be converted to negation normal form (NNF) so that negations only appear directly before named classes. For example $\neg(\forall eats\ Plant) \equiv \exists eats\ \neg Plant$. Furthermore, if C_1 is a named class, the complement of all the subclasses of C_1 will be added. The complement of each necessary & sufficient conditions of C_1 will also be added.

Domain/Range axioms (Rule 3): As explained in section 3.1, in OWL, domain restrictions act as universal restrictions such that the all individuals to the property is applied can be inferred to be of the type indicated by the domain. Therefore, if an existential (someValuesFrom) restriction acting along the property P is added to the debugging super conditions, and P has a domain of C_d , then C_d is also added to the debugging super conditions. (Note that the domain of P might have been declared as the range of its inverse.)

Functional /Inverse functional property (Rule 4): If an existential restriction (someValuesFrom) or a hasValue restriction is added to the debugging super conditions, and the property that the restriction acts along is a functional property P^3 , then a $\leq 1 P$ restriction (max cardinality restriction) is added to the debugging super conditions.

Intersection Rule (Rule 8): If an OWL class $C_1 \sqcap C_2$ is added to, then both C_1 and C_2 are added to the debugging super conditions.

³ A functional property implies that an individual may only be related to at most one other individual via that property

4.4 Determining The Most General Conflict

Determining the most general conflict is based on a simple observation: If an OWL class C conflicts with another class D , then then it conflicts with any subclass of D . Therefore we can eliminate any classes that are subclasses of other classes already in the *Debugging super conditions*.

The most general conflict – $MGC(C)$ – is therefore defined as follows.

Definition 2. Let $DSC(C)$ to debugging super conditions of the unsatisfiable Class C . The **most general conflict** ($MGC(C)$) of C is a set of OWL class descriptions, such that:

1. $MGC(C) \subseteq DSC(C)$
2. Intersection of $MGC(C)$ of all the concepts belonging to is unsatisfiable.
3. $\forall C_1 : MGC(C), C_2 : MGC(C)$, such that $C_1 \sqsubseteq C_2 \Rightarrow C_1 = C_2$
4. $\nexists C_1 : DSC(C)$, such that
 $C_2 \ni MGC(C)$ and $\exists C_2 : MGC(C)$ such that $C_2 \sqsubseteq C_1$ and
Intersection of all the concepts belonging to $MGC(C) \cup \{C_1\} - \{C_2\}$ is unsatisfiable

Condition 3 ensures that no class in $MGC(C)$ is subclass of another class in $MGC(C)$. Condition 4 ensures that if we replace any class in $MGC(C)$ with one of its superclass in $DSC(C)$, the intersection of $MGC(C)$ will become satisfiable.

4.5 Analysing the Most General Conflict

Having determined the most general conflict set, the final step is to analyse it to find the route use of the conflict and provide the explanation to users about the reason these set of axioms are conflicted. Although there theoretically indefinitely many ways in which inconsistencies may arise, we have found empirically that most can be boiled down to a small number of ‘error patterns’ to be checked by the heuristic debugger.

There are two broad classes of reasons that the *Most general conflict set* can be unsatisfiable.

- It can contain one or more classes – including restrictions – that are themselves unsatisfiable
- The intersection of two or more classes could be unsatisfiable.

Each of these cases will be dealt with in turn; the debugger generates suitable error messages for each case.

Unsatisfiable Superclasses

Existential (someValuesFrom) restriction There are three common reasons for an existential restriction to be unsatisfiable:

- Its filler may be unsatisfiable, in which case the filler must be analysed to find the root cause from which the unsatisfiability propagated. In this case the debugger will suggest that the filler should be the next class to be debugged.

- The filler may be disjoint from the range of the property. In this case the debugger will suggest that the filler and property range should be examined to determine why they are disjoint.
- The property may have an unsatisfiable domain. In this case the domain class will have already been added to the debugging super conditions and will therefore be found to be the cause of unsatisfiability.

Universal (allValuesFrom) restriction A universal restriction alone will never be unsatisfiable. Since a universal restriction does not imply that anything actually exists, it can be trivially satisfied. A universal restriction only leads to an inconsistency when there is a corresponding existential restriction along the same property that has a filler which is disjoint from the filler of the universal restriction. However, universal restrictions that are only trivially satisfiable are usually errors. Later addition of existential restrictions are likely to cause classes to become unsatisfiable. Therefore, the debugger generates warnings for trivially satisfied restrictions.

Maximum/Minimum/Equality cardinality restriction In OWL, cardinality restrictions do not specify a filler⁴. Therefore, the only common situation in which they themselves can be unsatisfiable is if the restricted property has an unsatisfiable domain and the restriction is a minimum cardinality greater than zero restriction. In this case the domain class will have been added to the debugging super conditions and will therefore be found to be the cause of unsatisfiability.

Intersection condition An intersection condition will be unsatisfiable if at least one of the operand classes is unsatisfiable. All of conditions that represent the operand classes will have been added to the debugging super conditions and therefore the cause of unsatisfiability will be found by examining these operand conditions.

Union condition A union conditions will only be unsatisfiable if *all* of its operand classes are unsatisfiable. In this case the debugger will suggest that all of the operand conditions should be debugged by individually checking them.

Complement condition If a complement condition is unsatisfiable the operand class must be equal or be inferred equal to owl:Thing.

hasValue restriction If a hasValue restriction is unsatisfiable, the filler individual is a member of an unsatisfiable class or a class disjoint with the range of the property in question. In this case the debugger will suggest that the class which the filler is a member of should be debugged.

Contradictory Super Conditions The second common cause of a class being unsatisfiable is that two or more *debugging super conditions* contradict each other, i.e. their conjunction is unsatisfiable. This situation can arise for a variety of reasons as described below. Unless otherwise stated, the debugger generates an explanation for the user.

- The class in question has been asserted to be disjoint with one of its super conditions.
- A universal (allValuesFrom) and an existential (someValuesFrom) that act along the same property have disjoint fillers. In this case the debugger will suggest that the intersection of the two fillers should be debugged in order to determine why the fillers are disjoint from each other.

⁴ i.e. there are no “qualified cardinality restrictions”

- A universal (allValuesFrom) has an unsatisfiable filler (owl:Nothing) which conflicts with any existential or minimum cardinality restriction on the same property.
- The super conditions contain a maximum (or equality) cardinality restriction, limiting the number of relationships along property P to n , but there are more than n disjoint filler classes implied by existential and/or minimum cardinality constraints.
- The super conditions contain two or more cardinality restrictions that act along the same property but contradict each other. For example, $\leq 2P$ and $\geq 3P$.

5 Case Study

This section illustrates the use of the debugger with an example taken from an ontology about pizzas⁵. The pizza ontology contains the class hierarchy shown in Figure 3. The ontology also contains the property `hasTopping`, which has a domain of `Pizza`. The ontology contains the following class axioms:

`IceCreamWithChocolateSauce` \sqsubseteq \exists `hasTopping` `ChocolateSauce`

`Pizza` \sqsubseteq \neg (\exists `hasTopping` \neg `PizzaTopping`)

When the ontology is classified, it is found that the class `IceCreamWithChocolateSauce` is unsatisfiable. In order to debug this class, the debugger is started and the class is selected. With the debugger running, the user is lead through the steps shown in Figure 4. At the end of each debugging step, the debugger presents a tree of conditions, which represent the conditions that instances of the class being debugged must fulfil – the parent child relationships in the tree are *is-generated-from*. For example, at the end of the first debugging step depicted in Figure 4, all instances of `IceCreamWithChocolateSauce` must also be instances of `Pizza`, which was generated from \exists `hasTopping` `ChocolateSauce` due to the fact that `Pizza` is in the domain of the `hasTopping` property. Conditions that cause an unsatisfiability are boxed in red, and an explanation is generated. In this case the conditions \exists `hasTopping` `ChocolateSauce` and \forall `hasTopping` `PizzaTopping` contradict each other – the explanation being “The universal restriction means that all relationships along `hasTopping` must be to individuals that are members of `PizzaTopping`. However, the existential restriction means that there must be at least one relationship to an individual from `ChocolateSauce`, which is disjoint from `PizzaTopping`.” After the user has pressed the Continue button, the debugger suggests that the next step is to determine why `PizzaTopping` and `ChocolateSauce` are disjoint from each other. At the end of this final step the debugger explains that the classes `PizzaTopping` and `ChocolateSauce` are disjoint from each other because `PizzaTopping` is disjoint with `DessertTopping` which is an ancestor class of `ChocolateSauce` – the explanation being “The two classes `ChocolateSauce` and `PizzaTopping` are disjoint from each other. `DessertTopping`, which is an ancestor class of `ChocolateSauce`, has been asserted to be disjoint with `PizzaTopping`.”

⁵ We typically use the domain of pizzas as it is easily understood but rich enough to illustrate key principles and common errors [6]

6 Related Work and Conclusions

6.1 Related Work

Work in the area of reasoner explanation is still in its infancy. Other approaches were discussed at the 3rd International Semantic Web Symposium (ISWC 2004) held in Hiroshima, Japan, where the Pellet reasoner [4] team's future work includes the development of an explanation mechanism for concept satisfiability. The OWL-Lite ontology editor OntoTrack [9], is able to generate explanations for subsumption, equivalence and concept satisfiability using algorithms based on the work of Borgida et. al. in explaining subsumption in \mathcal{ALC} [3]. The OntoTrack team implemented their own tableaux based explanation generator, which is currently limited to working with unfoldable AL_{EN} ontologies, and generates an explanation corresponding to the stages of the tableaux algorithm expansion.

6.2 Limitations

As the title of this paper suggests, the debugger is based on heuristics and pattern matching. The debugger cannot determine the root cause of unsatisfiability in every case, and is therefore not complete. However, we have found that this does not have a serious impact on the usefulness of the debugger, since in most cases the mistakes made by ontologists, ranging from students to experts, can be described by a small number of error patterns that the debugger is adept at spotting.

6.3 Conclusions and Future Work

In this paper we have described a heuristic approach to ontology debugging that uses a DL Reasoner, treating the reasoner as a 'black box'. This means that the debugger is totally reasoner independent, thereby affording the user the benefits of being able to select a reasoner that is appropriate for their needs. The black box approach also helps to minimise any potential versioning problems between the debugger and future advancements in DL reasoners, since the debugger does not need to know the details of any internal tableaux algorithms, reasoner optimisations or capabilities. The debugger is useful for beginners constructing small ontologies, through to domain experts and ontology engineers working with large complex ontologies, as it reduces the amount of time and frustration involved in tracking down ontological inconsistencies.

Acknowledgements.

This work was supported in part by the CO-ODE project funded by the UK Joint Information Services Committee and the HyOntUse Project (GR/S44686) funded by the UK Engineering and Physical Science Research Council and by 21XS067A from the National Cancer Institute. Special thanks to all at Stanford Medical Informatics, in particular Holger Knublauch, for their continued collaboration and to the other members of the ontologies and metadata group at Manchester for their contributions and critiques.

References

1. Sean Bechhoffer. The dig description logic interface: Dig/1.1. Technical report, The University Of Manchester, The University Of Manchester, Oxford Road, Manchester M13 9PL, 2003.
2. Sean Bechhoffer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah McGuinness, Peter Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference, February 2004.
3. A Borgida, E Franconi, I Horrocks, D McGuinness, and P Patel-Schneider. Explaining alc subsumption. In *Description Logics*, 1999.
4. Bijan Parsia Evren Sirin. Pellet: An owl dl reasoner. In Ralf Moller Volker Haaslev, editor, *Proceedings of the International Workshop on Description Logics (DL2004)*, June 2004.
5. Alan Rector Holger Knublauch, Mark Musen. Editing description logic ontologies with the protege-owl plugin. In *International Workshop on Description Logics - DL2004*, 2004.
6. Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, and Chris Wroe. A practical guide to building owl ontologies using protégé-owl and the co-ode tools. Available from <http://www.co-ode.org/resources>, 2004.
7. Ian Horrocks. Fact++ web site. <http://owl.man.ac.uk/factplusplus/>.
8. Ian Horrocks. The fact system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, pages 307 – 312. Springer-Verlag, May 1998.
9. Thorsten Liebig and Olaf Noppens. Ontotrack: Combining browsing and editing with reasoning and explaining for owl lite ontologies. In S.A. McIlraith et al., editor, *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*. Springer-Verlag, 2004.
10. Alan L. Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. Owl pizzas: Practical experience of teaching owl-dl: Common errors and common patterns. In *Proceedings of Engineering Knowledge in the Age of the Semantic Web*, 2004 2004.
11. Ralf Moller Volker Haarslev. Racer system description. In *International Joint Conference on Automated Reasoning, IJCAR 2001*, 2001.

Rule 1: Named class rule

(a) **IF** $C_1 \in DSC(C) \wedge C_1 \sqsubseteq C_2$, where C_1 is a named OWL class
THEN $C_2 \in DSC(C)$

(b) **IF** $C_1 \in DSC(C)$ and $Disj(C_1, C_2)$, where C_1 and C_2 are named OWL classes
THEN $\neg C_2 \in DSC(C)$

Rule 2: Complement class rule

(a) **IF** $\neg C_1 \in DSC(C)$, where C_1 is a named OWL class
THEN IF $C_2 \sqsubseteq C_1$, **THEN** $\neg C_2 \in DSC(C)$
IF $C_1 \equiv C_2$, **THEN** $\neg C_2 \in DSC(C)$

(b) **IF** $\neg C_1 \in DSC(C)$, where C_1 is an anonymous OWL class
THEN $NORM(C_1) \in DSC(C)$

Rule 3: Domain/Range rule

(a) **IF** $\exists S.C_1 \in DSC(C) \vee \geq n S \in DSC(C) \vee = n S \in DSC(C)$,
where $n > 0$, and $DOM(S) = C_2$
THEN $C_2 \in DSC(C)$

(b) **IF** $\exists S.C_1 \in DSC(C) \vee \geq n S \in DSC(C) \vee = n S \in DSC(C)$,
and where $n > 0$, $INV(S) = S_1$ and $RAN(S_1) = C_2$
THEN $C_2 \in DSC(C)$

(c) **IF** $\exists S.C_1 \in DSC(C) \vee \geq n S \in DSC(C) \vee = n S \in DSC(C)$,
where $n > 0$, and $RAN(S) = C_2$
THEN $\forall S.C_2 \in DSC(C)$

Rule 4: Functional/Inverse functional property

(a) **IF** $\exists S.C_1 \in DSC(C)$ or $\geq n S \in DSC(C)$ or $= n S \in DSC(C)$,
where $n > 0$ and S is functional
THEN $\leq 1 S \in DSC(C)$

(b) **IF** $\exists S.C_1 \in DSC(C)$ or $\geq n S \in DSC(C)$ or $= n S \in DSC(C)$,
where $n > 0$ and $INV(S) = S_1$, S_1 is inverse functional
THEN $\leq 1 S \in DSC(C)$

Rule 5: Inverse Rule

IF $\exists S.C_1 \in DSC(C)$ and $INV(S) = S_1$,
and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall S_1 C_3$
THEN $C_3 \in DSC(C)$

Rule 6: Symmetric Rule

IF $\exists S.C_1 \in DSC(C)$ and S is a symmetric property,
and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall S C_3$
THEN $C_3 \in DSC(C)$

Rule 7: Transitive Rule

IF $\forall S.C_1 \in DSC(C)$ and S is a transitive property,
THEN $\forall S \forall S.C_1 \in DSC(C)$

Rule 8: Intersection Rule

IF $C \wedge C_1 \in DSC(C)$,
THEN $C \in DSC(C)$ and $C_1 \in DSC(C)$

Rule 9: Subproperty Rule

(a) **IF** $\forall S.C_1 \in DSC(C)$ and $S_1 \sqsubset S$, **THEN** $\forall S_1.C_1 \in DSC(C)$

(b) **IF** $\leq n S \in DSC(C)$ and $S_1 \sqsubset S$, **THEN** $\leq n S_1.C_1 \in DSC(C)$

(c) **IF** $\exists S.C_1 \in DSC(C)$ and $S_1 \sqsupset S$, **THEN** $\exists S_1.C_1 \in DSC(C)$

(d) **IF** $\geq n S \in DSC(C)$ and $S_1 \sqsupset S$, **THEN** $\geq n S \in DSC(C)$

Rule 10: Other inference Rule

IF C_1 can be inferred by any subset of $UC(C)$, where C is a named class
THEN $C_1 \in DSC(C)$

Fig. 2. Rules for the membership of Debugging Super Conditions (DSC).

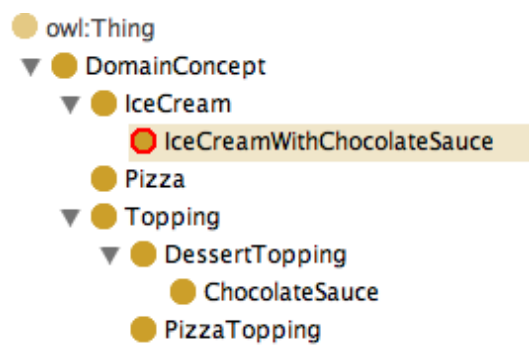


Fig. 3. ExampleHierarchy



Fig. 4. DebuggingSteps