Debugging Unsatisfiable Classes in OWL Ontologies

⋆

Aditya Kalyanpur[1], Bijan Parsia[2], Evren Sirin[1], James Hendler[1]

University of Maryland, MIND Lab, 8400 Baltimore Ave,
College Park MD 20742, USA

{aditya, evren, hendler}[1]@cs.umd.edu, bparsia@isr.umd.edu[2]

Abstract

As an increasingly large number of OWL ontologies become available on the Semantic Web and the descriptions in the ontologies become more complicated, finding the cause of errors becomes an extremely hard task even for experts. Existing ontology development environments provide some limited support, in conjunction with a reasoner, for reporting errors in OWL ontologies. Typically these are restricted to the mere detection of, for example, unsatisfiable concepts. However, the diagnosis and resolution of the bug is not supported at all. For example, no explanation is given as to *why* the error occurs (e.g., by pinpointing the root clash, or axioms in the ontology responsible for the clash) or *how* dependencies between classes cause the error to propagate (i.e., by distinguishing *root* from *derived* unsatisfiable classes). In the former case, information from the *internals* of a description logic tableaux reasoner can be extracted and presented to the user (glass box approach); while in the latter case, the reasoner can be used as an oracle for a certain set of questions and the asserted structure of the ontology can be used to help isolate the source of the problems (black box approach). Based on the two approaches, we have integrated a number of debugging cues generated from our reasoner, Pellet, in our hypertextual ontology development environment, Swoop. A conducted usability evaluation demonstrates that these debugging cues significantly improve the OWL debugging experience, and point the way to more general improvements in the presentation of an ontology to users.

Key words: OWL, Ontology Debugging, Explanation, Semantic Web

⋆ This paper is an extension of the WWW'05 Conference paper on 'Debugging OWL Ontologies' [1] and the DL'05 Workshop paper on 'Black Box Debugging of Unsatisfiable Concepts' [2].

1 Introduction

Now that OWL [3] is a W3C Recommendation, one can expect that a much wider community of users and developers will be exposed to the expressive description logic SHIF(D) and SHOIN(D) which are the basis of OWL-DL. These users and developers are likely not to have a lot of experience with knowledge representation (KR), much less logic-based KR, much less description logic based KR. For such people, having excellent documentation, familiar techniques, and helpful tools is a fundamental requirement.

A ubiquitous activity in programming is debugging, that is, finding and fixing defects in a program. Ontologies too have defects, and a common activity is to find and repair these defects. Unfortunately, the tool and training support for debugging ontologies is fairly weak. [1] We have chosen to focus on debugging unsatisfiable concepts (and contradictory ABoxes) because contradictions, in general, seem analogous to fatal errors in programs. Debugging fatal errors in programs can be relatively straightforward: the program crashes, there is a stack trace or similar information, and (one measure of) success is a running program. For ontologies, current tools (reasoners) do support indicating the dramatic failure of a unsatisfiable class, and success is similarly clear, however, the diagnosis and resolution of the bug is not supported at all. Consider the case of the Tambis OWL ontology [2] shown in Figure 1 in which more than a third of the classes are unsatisfiable:

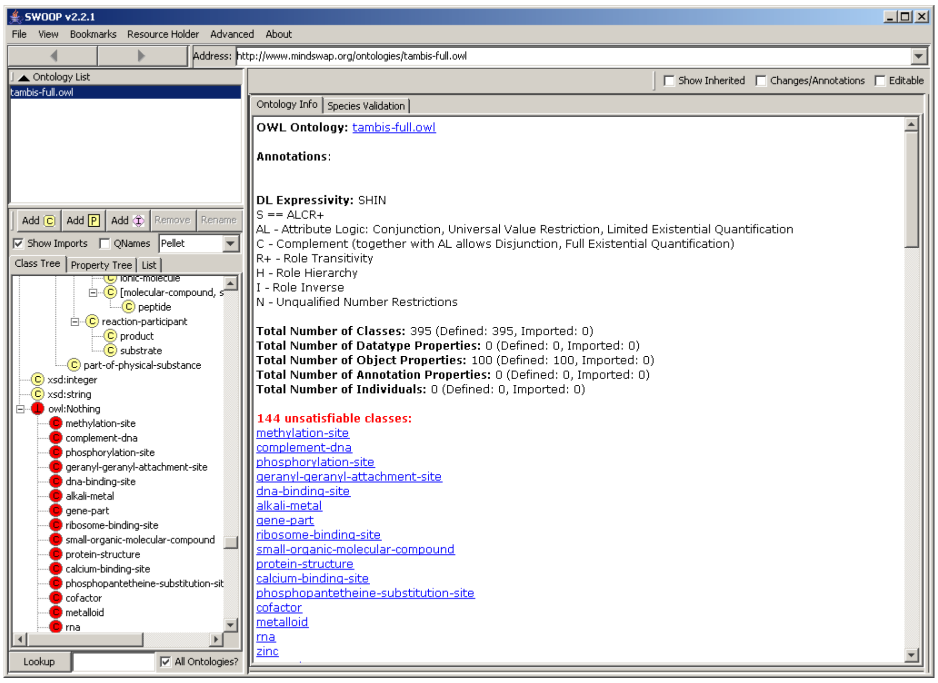
Here the tool has detected unsatisfiable classes in the ontology but no reason is given as to *why* a specific class is unsatisfiable or what measures can be taken to rectify it. Also, the fact that there are so many unsatisfiable classes makes the debugging task seem all the more overwhelming.

When new modelers encounter cases of unsatisfiability such as this, they are often a loss at what to do. This has two negative general consequences inhibiting adoption and effective use of OWL ontologies: either developers tend to under specify their concepts to "avoid" errors or they give up on ontologies altogether.

The above case illustrates that OWL ontology tools have to go much further in organizing and presenting the information supplied by the reasoner and existing in the ontology in order to aid debugging. For example, tools could pinpoint the problematic axioms in the ontology responsible for an unsat-

[1] While, historically, good KR modeling practices have been developed and described, often with an emphasis on description logics[4], tool support for "good" modeling remains elusive, especially given the lack of consensus on practice and the strong dependence of goodness on application and domain specifics.

[2] http://www.cs.man.ac.uk/˜horrocks/OWL/Ontologies/tambis-full.owl


SWOOP v2.2.1
File View Bookmarks Resource Holder Advanced About
Address: http://www.mindswap.org/ontologies/tambis-full.owl
Ontology List
tambis-full.owl
Add C Add P Add I Remove Rename
Show Imports QNames Pellet
Class Tree Property Tree List
ionic-molecule
[molecular-compound, s
peptide
reaction-participant
product
substrate
part-of-physical-substance
xsd:integer
xsd:string
owl:Nothing
methylation-site
complement-dna
phosphorylation-site
geranyl-geranyl-attachment-site
dna-binding-site
alkali-metal
gene-part
ribosome-binding-site
small-organic-molecular-compound
protein-structure
calcium-binding-site
phosphopantetheine-substitution-sit
cofactor
metalloid
rna
Lookup All Ontologies?
Show Inherited Changes/Annotations Editable
Ontology Info Species Validation

```
OWL Ontology: tambis-full.owl

Annotations:


DL Expressivity: SHIN
S == ALCR+
AL - Attribute Logic: Conjunction, Universal Value Restriction, Limited Existential Quantification
C - Complement (together with AL allows Disjunction, Full Existential Quantification)
R+ - Role Transitivity
H - Role Hierarchy
I - Role Inverse
N - Unqualified Number Restrictions

Total Number of Classes: 395 (Defined: 395, Imported: 0)
Total Number of Datatype Properties: 0 (Defined: 0, Imported: 0)
Total Number of Object Properties: 100 (Defined: 100, Imported: 0)
Total Number of Annotation Properties: 0 (Defined: 0, Imported: 0)
Total Number of Individuals: 0 (Defined: 0, Imported: 0)

144 unsatisfiable classes:
methylation-site
complement-dna
phosphorylation-site
geranyl-geranyl-attachment-site
dna-binding-site
alkali-metal
gene-part
ribosome-binding-site
small-organic-molecular-compound
protein-structure
calcium-binding-site
phosphopantetheine-substitution-site
cofactor
metalloid
rna
zinc
```

Fig. 1. OWL version of the Tambis ontology as viewed in an Ontology Browser and tested using a DL Reasoner

isfiable class; or detect and highlight interdependencies between unsatisfiable classes to help differentiate the root bugs from the less critical ones (especially useful in cases such as Tambis where there are a large number of unsatisfiable classes). Also, readable explanations (proofs) could be provided to help the modeler understand the cause of the problem.

The goal of our research is to develop techniques of the form described above to support the debugging of OWL Ontologies and to demonstrate its significance and use in practice. In this paper, we describe some first steps in providing debugging support for unsatisfiable concepts in the Swoop OWL ontology browser and editor [5]. We have explored both black box and glass box generation of debugging clues by a description logic tableau reasoner, in this case, our OWL reasoner Pellet [6].

A broader goal of this research is to demonstrate that good ontology debugging support not only gives users a *sense* of control over their modeling, but also encourages them to experiment more freely with expressions, helping them come to understand their ontologies through the debugging process.

2 Defects in OWL

Any description logic knowledge base can have defects (or inconsistencies), which need to be detected and resolved in order to obtain any useful accurate information from it. OWL ontologies are no different and we consider some factors which make them susceptible to errors.

(1) *Difficulty in understanding modeling*: As noted earlier, OWL users and developers are not likely to have a lot of experience with description logic based KR, and hence, without adequate tool support for training and explanation, engineering ontologies can be a hard task for such users. As ontologies become larger and more complex, highly non-local interactions in the KB combined with other intrinsic aspects of OWL such as open-world semantics, non-unique name assumption etc. make modeling, and analyzing the effects of modeling non-trivial even for domain experts.

(2) *Misalignment of OWL Ontologies*: When related domain ontologies created by separate parties are merged using `owl:imports`, the combination can result in modeling errors [7]. This could be due to ontology authors either having different views of the world, following alternate design paradigms, or simply, using a conflicting choice of modeling constructs. An example is when the two upper-level ontologies, CYC and SUMO are merged leading to a large number of unsatisfiable concepts due to disjointness statements present in CYC.

(3) *Migration to OWL*: Whenever OWL ontologies are extracted or derived from schema/ontologies in other languages such as XML, DAML etc, a faulty migration process can lead to an incorrect specification of concepts or individuals in the resultant OWL version. For example, the OWL version of the Tambis ontology contains 144 unsatisfiable classes (out of 395) due to an error in the transformation script used in the conversion process.

Defects in OWL can be due to various reasons, however, they broadly fall into three main categories: *syntactic, semantic and modeling defects.* We now discuss each of these defects separately in terms of the standard techniques for detecting these errors and for presenting them to a user.

2.1 Syntactic Defects

Syntactic issues loom large in OWL for a number of reasons including the baroque exchange syntax, RDF/XML and the use of URIs (and their abbreviations), but most of these are straightforward to detect and rectify using XML parsers and RDF validators (http://www.w3.org/RDF/Validator/). However, for OWL DL, there is yet another layer of syntactic structure on top of the corresponding RDF graph, i.e., a number of restrictions are imposed on the form of the graph in order for it to count as an instance of the OWL DL "species". These restrictions are quite onerous for authors and easy to violate as, in general, importing is not species safe: importing an OWL Lite document into another may result in an OWL Full document, and an OWL DL document importing either an OWL Lite or OWL DL document may become OWL Full. Even OWL Full, the *superset* of the rest, may become OWL DL or Lite upon an import. The WebOnt working group defined a category of OWL processor for so-called species validation, and though there were serious fears of the complexity and implementation of such validation, several implementations have emerged and appear to be reliable.

2.2 Semantic Defects

Given a syntactically correct OWL ontology, semantic defects are those which can be detected by an OWL reasoner. These include unsatisfiable classes and inconsistent ontologies. Unsatisfiable classes are those which cannot be true of any possible individual, that is, they are equivalent to the empty set (or, in description logic terms, to the bottom concept, or, in OWL lingo, to owl:Nothing). For example, class A is unsatisfiable if it is a subclass of both, class C and $\neg C$, since it implies a direct contradiction. Unsatisfiable concepts are usually a fundamental modeling error, as they cannot be used to characterize any individual. Unsatisfiable concepts are also quite easy for a reasoner to detect and for a tool to display. However, determining *why* a concept in an ontology is unsatisfiable can be a considerable challenge even for experts in the formalism and in the domain, even for modestly sized ontologies. The problem worsens significantly as the number and complexity of axioms of the ontology grows.

Inconsistent ontologies are those which have a contradiction in the instance data, e.g., an instance of an unsatisfiable class. They are also fairly easy for a reasoner to detect, if it can process the ontology at all. In fact, in tableau reasoners, unsatisfiability testing is reduced to a consistency test by positing that there is a member of the to be tested class and doing a consistency check on the resultant knowledge base (KB). However, unlike with mere unsatisfiable classes, an inconsistent ontology is, on the face of it, very difficult for a reasoner to do further work with. Since anything at all follows from a contradiction, no other results from the reasoner (e.g., with regard to the subsumption hierarchy) are useful.

2.3 Modeling/Style Defects

These are defects that are not necessarily invalid, syntactically or semantically, yet are discrepancies in the KB or unanticipated results of modeling, which require the modelers' attention before use in a specific domain or application scenario. Consider the following cases:

- There may be unintended inferences (subsumption, realization relationships etc.) discovered by the reasoner. For example, it can be inferred that "parents of at least three children" is a subclass of "parents with at least two children", even if there is no explicit assertion of that relationship. Though the reasoner can detect and report subsumptions such as this, it cannot distinguish between desirable (non)inferences and undesired ones.
- Missing type declarations can occur in a KB, such as if a resource is used in a particular manner that entails it to be of a particular type, but is not explicitly declared to be so, e.g., given the triple `<John hasParent Mary>`, where `hasParent` is known to be an OWLObjectProperty, one can infer that `John` and `Mary` both have to be of type OWLIndividual. In such cases, the reasoner will infer the corresponding entailment, but the absence of this explicit information could be considered as a defect.
- In some cases, redundancies may exist in the KB, such as when an asserted axiom is entailed by another set of axioms from the KB. Here, depending on whether the redundancy is desired or not, the case could be considered as a defect.
- There may be cases of unused atomic classes or properties with no references anywhere in the KB (i.e., the term is not explicitly used in any axiom in the KB), which can be considered as extraneous data.

We will not deal with the detection and debugging of these subtler, domain and modeler dependent defects, focusing, in this paper, on debugging unsatisfiable concepts. Since modeling defectiveness is very dependent on the modeler's intent, we believe that effective debugging requires the expression of that intent to the system. In other words, we suspect *testing* and test cases are the right modality for dealing with some of these defects.

3 Current State of the Art in OWL Ontology Debugging

There has long been significant interest in explaining inferences to the non-sophisticated user when implementing reasoning services for Description Logic (DL) systems. In [8] the author provides explanations as *proof fragments* based on standard structural subsumption algorithms for the CLASSIC KR system. The method has been extended for ALC reasoning in [9] wherein the authors use a *modified sequent calculus* to explain a tableaux based proof. An implementation of this idea can be seen in the tool - OntoTrack [10], which provides "instant reasoning feedback". Additionally, the *Inference Web* Infrastructure [11] comprised using a web-based registry for information sources, reasoners, etc., and a portable proof specification language for sharing explanations.

However, while the emphasis so far has been on explaining subsumption in relatively inexpressive terminologies (ALC), there has not been much work done in explaining and fixing errors in expressive DLs such as SHION(D). Moreover, with OWL reaching recommendation status only recently (February 2004), the area of debugging OWL ontologies, in particular, is a largely unexplored field.

In [12], the authors present a tool, Chimaera, which apart from supporting ontology merging, allows users to run a diagnostic suite of tests across an ontology. The tests include incompleteness tests, syntactic checks, taxonomic analysis, and semantic checks, and the results are displayed as an interactive log, which the users can study and explore. The focus here is clearly on detecting modeling defects, whereas explanation support for semantic defects is fairly weak.

Work has been done on a 'Symptom Ontology' [13] for representing errors and warnings resulting from defects in OWL ontologies, and an implementation is provided in the consistency-checking tool, ConVisor. The authors here do a good job of categorizing commonly occurring symptoms and motivate the significance of creating and exchanging standardized bug reports using a symptom ontology. However, just as in the previous case, their work does not deal with pinpointing the cause of inconsistency.

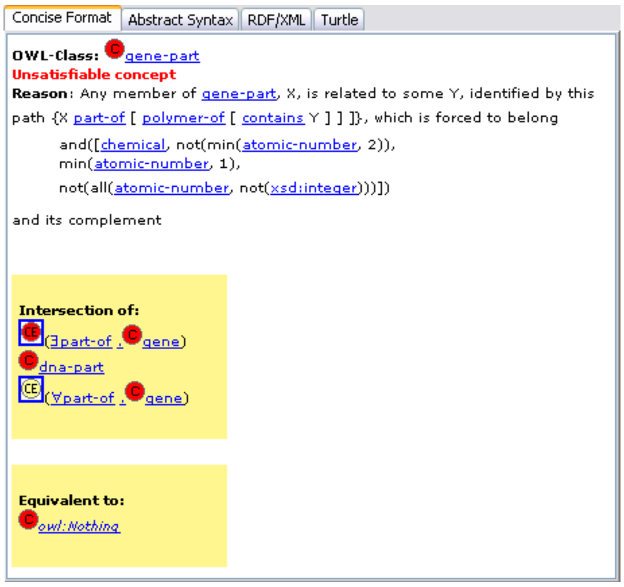
For explaining inconsistencies, two interesting efforts have been recently published. The first is a glass-box method described in [14], where non-standard reasoning algorithms based on minimization of axioms using Boolean methods are used to debug the DICE terminology. The authors focus on *axiom and concept pinpointing* and introduce useful relevant terms for this purpose such as MUPS (Minimal Unsatisfiability Preserving Sub-TBoxes). However, a drawback of their approach is that its restricted to unfoldable ALC TBoxes.

The second piece of relevant work is a black-box heuristic approach described in [15], which is used to detect and explain inconsistencies in OWL ontologies. The idea here is to use a pre-defined set of rules / conditions to detect commonly occurring error patterns in ontologies based on extensive use-case data. However, such a rule-based heuristic is clearly incomplete.

4 Diagnosing Unsatisfiability

When faced with a detected unsatisfiable concept, one must perform a diagnosis, that is, come to understand the underlying causes of the unsatisfiability and determine which are problematic. Once the diagnosis is completed, various remedies can be considered and their costs and benefits evaluated. We distinguish two families of reasoner-based techniques for supporting diagnosis: glass box and black box techniques. In glass box techniques, information from the *internals* of the reasoner is extracted and presented to the user (sometimes the implementation is altered in order to improve the information returned). In black box techniques, the reasoner is used as an oracle for a certain set of questions e.g., the standard description logic inferences (subsumption, satisfiability, etc.).

However, before we elaborate on the glass and black box ontology debugging techniques, it is important to discuss basic browsing and rendering functionality that an ontology engineering tool (coupled with a reasoner) can provide in order to aid understanding, analysis and debugging of the ontology. In this case, we discuss functionality as provided in Swoop (editor), which uses Pellet (reasoner), both of which are core components used for debugging purposes.



OWL-Class: gene-part
Unsatisfiable concept
Reason: Any member of gene-part, X, is related to some Y, identified by this path {X part-of [polymer-of [contains Y]]]}, which is forced to belong

```
        and([chemical, not(min(atomic-number, 2)),
        min(atomic-number, 1),
        not(all(atomic-number, not(xsd:integer)))])
```

and its complement

Intersection of:
CE (∃part-of . C gene)
C dna-part
CE (∀part-of . C gene)

Equivalent to:
C *owl:Nothing*

Fig. 2. The class `gene-part` is unsatisfiable on two counts: its defined as an intersection of an unsatisfiable class (`dna-part`) and an unsatisfiable class expression (∃`partof.gene`), both highlighted using red tinted icons.

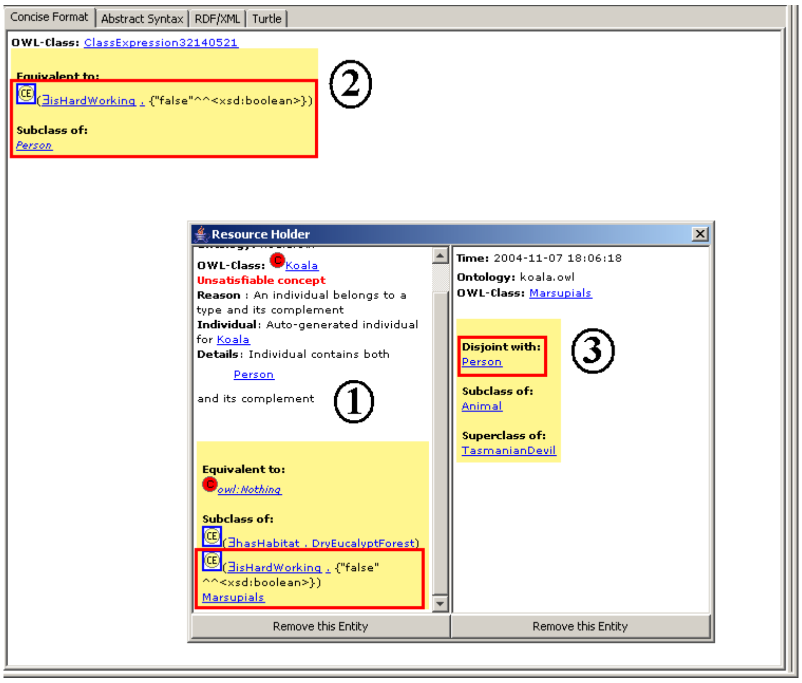
Swoop has a *debug* mode wherein the basic rendering of entities is augmented with information obtained from a reasoner. Different rendering styles, formats, and icons are used to highlight key entities and relationships that that are likely to be helpful to debugging process. For example, all *inferred* relationships (axioms) in a specific entity definition are italicized and are obviously not editable directly. In the future, we plan to extend this feature by displaying the reasoning chain for the simple, yet non-trivial inferences by pointing to related definitions and axioms (e.g., `C` is an intersection of (`D`,..) implies `C` is a subclass of `D`), but for now, simply highlighting them separately is useful to the ontology modeler as they can (potentially) point to unintended assertions. On a similar note, in the case of multiple ontologies, i.e., when one ontology imports another, all *imported* axioms in a particular entity definition are italicized as well. Highlighting them helps the modeler differentiate between explicit assertions in a single context and the net assertions (explicit plus implied) in a larger context (using imports), and can also reveal unintended semantics.

All unsatisfiable named classes, and even class expressions, are marked with red icons whenever rendered — a useful pointer for identifying dependencies between inconsistencies. In Figure 2 (the Tambis ontology), note how simply looking at the class definition of `gene-part` makes the reason for the inconsistency apparent: it is a subclass of the inconsistent class `dna-part` and the inconsistent class expression $\exists$`partof.gene`. The hypertextual navigation feature of Swoop allows the user to follow these dependencies easily, and reach the root cause of the inconsistency, e.g., the class which is independently inconsistent in its definition (i.e., no red icons in its definition). In this manner, the UI guides the user in locating and understanding bugs in the ontology by narrowing them down to their exact source.

Also note that the class expressions themselves can be rendered as regular classes, displaying information such as sub/super classes of a particular expression (by clicking on the associated CE icon, see Figure 3). This sort of ad hoc "on-demand" querying (e.g., find all subclasses of a specific query expression) helps reveal otherwise hidden dependencies. Consider the case of the inconsistent class `Koala` depicted in Figure 3, which contains three labeled regions (the figure makes use of the Comparator feature in Swoop, discussed in Section 6). Region 1 shows the definition of the `Koala` class in terms of its subclass-of axioms: note the presence of the class expression $\exists$`isHardWorking.false` and the named class `Marsupials` mentioned here. Now, clicking on the class expression reveals that its an inferred subclass of `Person` (Region 2)[3], and clicking on `Marsupials` shows that its defined as **disjoint-with** class `Person` (Region 3). Thus, the contradiction is found – an instance of `Koala` is forced to be an instance of `Person` and ¬`Person` at the same time, and the bug can be fixed accordingly.

Finally, Swoop has an interesting non-standard search feature which can be

[3] A simple heuristic to manually debug an unsatisfiable class is to inspect its asserted and inferred subclass relationships that could potentially cause a contradiction, as is what motivates clicking the class expression link here.



OWL-Class: ClassExpression32140521

Equivalent to:
(∃isHardWorking . {"false"^^<xsd:boolean>})

Subclass of:
Person

OWL-Class: Koala
Unsatisfiable concept
Reason : An individual belongs to a type and its complement
Individual: Auto-generated individual for Koala
Details: Individual contains both
Person
and its complement

Equivalent to:
owl:Nothing

Subclass of:
(∃hasHabitat . DryEucalyptForest)
(∃isHardWorking . {"false"^^<xsd:boolean>})
Marsupials

Time: 2004-11-07 18:06:18
Ontology: koala.owl
OWL-Class: Marsupials

Disjoint with:
Person

Subclass of:
Animal

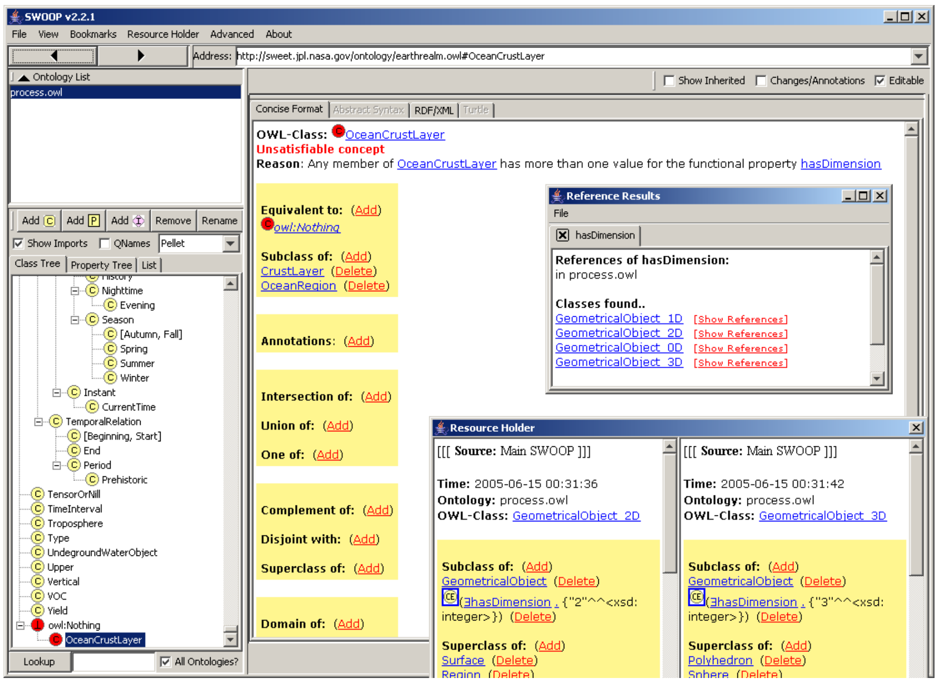
Superclass of:
TasmanianDevil

Fig. 3. The class `Koala` is unsatisfiable because (1) `Koala` is a subclass of `∃isHardWorking.false` and `Marsupials`; (2) `∃isHardWorking.false` is a subclass of `Person`; and (3) `Marsupials` is a subclass of ¬`Person` (disjoint). Note that the regions outlined in red are not automatically generated by the tool but are presented here for clarity.

useful during ontology debugging. This feature known as *Show References* highlights the usage of an OWL entity (concept/property/individual) by listing all references of that entity in local or external ontological definitions. The *Sweet-JPL* ontology set [4] presents an excellent use case for debugging using this feature. The class `OceanCrustLayer` is found to be unsatisfiable and a reason displayed for the clash is *'Any member of OceanCrustLayer has more than one value for the functional property hasDimension'* (Note: Clash detection is explained later). Now, running a *Show References* search on the property `hasDimension`, returns four classes `GeometricObject(0..3)D`, each of which has a different value restriction on the functional property `hasDimension`. This suggests that the unsatisfiable class is somehow related to more than one of these four classes causing the cardinality violation. This is indeed the case since by looking at the class hierarchy, one can note that `OceanCrustLayer` is a subclass of both the classes, `GeometricObject2D` and `GeometricObject3D`, and thus the reason for the contradiction becomes apparent.

While browsing, rendering and search functionality of the forms explained

[4] Sweet-JPL Ontologies are located at http://sweet.jpl.nasa.gov/ontology/. The bug in the ontology was fixed on May 24, 2005 after we e-mailed the ontology authors at NASA informing them about it. The previous faulty version can be found at http://www.mindswap.org/ontologies/debugging/buggy-sweet-jpl.owl



OWL-Class: OceanCrustLayer
Unsatisfiable concept
Reason: Any member of OceanCrustLayer has more than one value for the functional property hasDimension

Equivalent to: (Add)
owl:Nothing

Subclass of: (Add)
CrustLayer (Delete)
OceanRegion (Delete)

Annotations: (Add)

Intersection of: (Add)

Union of: (Add)

One of: (Add)

Complement of: (Add)

Disjoint with: (Add)

Superclass of: (Add)

Domain of: (Add)

Reference Results
File
hasDimension

References of hasDimension:
in process.owl

Classes found..
GeometricalObject_1D [Show References]
GeometricalObject_2D [Show References]
GeometricalObject_0D [Show References]
GeometricalObject_3D [Show References]

Resource Holder

[[[**Source:** Main SWOOP]]]

Time: 2005-06-15 00:31:36
Ontology: process.owl
OWL-Class: GeometricalObject_2D

Subclass of: (Add)
GeometricalObject (Delete)
(∃hasDimension . {"2"^^<xsd:integer>}) (Delete)

Superclass of: (Add)
Surface (Delete)
Region (Delete)

[[[**Source:** Main SWOOP]]]

Time: 2005-06-15 00:31:42
Ontology: process.owl
OWL-Class: GeometricalObject_3D

Subclass of: (Add)
GeometricalObject (Delete)
(∃hasDimension . {"3"^^<xsd:integer>}) (Delete)

Superclass of: (Add)
Polyhedron (Delete)
Sphere (Delete)

Fig. 4. The *Show References* feature (used along with the clash information and the resource holder) is used to hint at the source of the highly non-local problem for the unsatisfiable class `OceanCrustLayer`.

above can help the modeler understand and debug the ontology, it is still primarily a manual task and can be considerably challenging in most cases. Thus, automated techniques are needed to support ontology debugging and repair.

4.1 Glass box techniques

In glass box techniques, the internals of a description logic tableaux reasoner are modified to extract and reveal the cause for inconsistency of a concept definition. An advantage of the approach is that by tightly integrating the debugging with the reasoning procedure, precise results can be obtained. On the other hand, the reasoner needs to maintain extra data structures to track the source and its dependencies and this introduces additional memory and computation consumption.

Currently, glass box techniques are used to support two forms of debugging of unsatisfiable concepts:

(1) Present the *Clash* information: root cause of the contradiction
(2) Determine the minimal *Sets of Support*: the relevant axioms in the ontology that are responsible for the clash

4.1.1 Clashes

There are many different ways for the axioms in an ontology to cause an inconsistency. But these different combinations boil down to some basic contradictions in the description of an individual. Tableaux algorithms apply transformation rules to individuals in the ontology until no more rules are applicable or an individual has a clash. The basic set of clashes in a tableaux algorithm are:

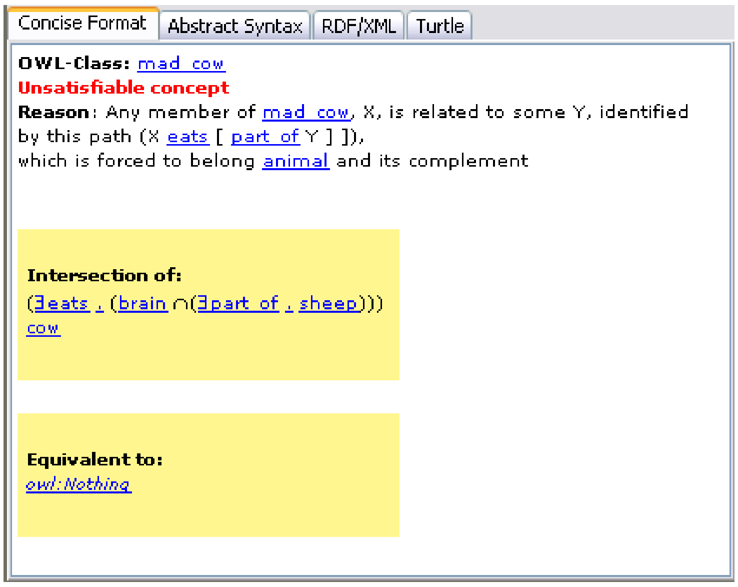
- *Atomic* An individual belongs to a class and its complement.
- *Cardinality* An individual has a max cardinality restriction but is related to more distinct individuals.
- *Datatype* A literal value violates the (global or local) range restrictions on a datatype property.

As a minimum requirement for the clash information to be useful for diagnosis, the reasoner should explain some details about the clash, e.g. which class and its complement is causing the clash. However, it is not easy to usefully present this clash information to the user. For example, the normalization and decomposition of expressions required in reasoning can obscure the error by getting away from the concepts actually used by the modeler, or the clash may involve some individuals that were not explicitly present in the ontology, but generated by the reasoner in order to try to adhere to some constraint. Those generated individuals may not even exist (or be relevant) in all models. For example, if an individual has a owl:someValuesFrom restriction on a property, the reasoner would generate a new anonymous individual that is the value of that property. Since these individuals (as with bnodes that exist in the original ontology) do not have a name (URI) associated with them, we can only use paths of properties to identify these individuals. This adds the extra burden to the user to make the connections between the identification path and the restrictions in the concept's definition but this is not always a big problem as illustrated in Figure 5 that shows an example from the Mad Cow ontology [5].

Depending on the reasoner's capabilities it is possible to increase the granularity of the clash explanations. For example, if an individual has two conflicting cardinality restrictions on a property, (e.g. $\geq$ `2 child` and $\leq$ `1 child`), then it is possible for the reasoner to detect this clash without generating individuals by just checking such obvious contradictions in cardinality restrictions. Generating explanations specific to these cases makes it easier for the user to see the relation between the clash and the existing axioms in the ontology.

Clash Detection Procedure:

[5] The Mad Cow ontology is used in OilEd tutorials



OWL-Class: mad cow
Unsatisfiable concept
Reason: Any member of mad cow, X, is related to some Y, identified by this path (X eats [part of Y]]), which is forced to belong animal and its complement

Intersection of:
(∃eats . (brain ∩(∃part of . sheep)))
cow

Equivalent to:
owl:Nothing

Fig. 5. The explanation of unsatisfiability for class `Mad Cow` includes the description of an anonymous individual created by the reasoner. It is easy to see the connection between the path that identifies the individual and the existential restrictions in the `Mad Cow` definition.

It is important to note that tableau expansion rules may find many different clashes during a satisfiability test. Due to the non-determinism caused by the OWL constructors such as owl:unionOf and owl:maxCardinality [6], some of the clashes do not reflect an error in the ontology but simply guide the tableau rules to the correct model. Therefore, the question is how to identify the *inconsistency-revealing* clashes from intermediary clashes. It turns out that dependency directed backjumping technique can be utilized to make this distinction.

Dependency directed backjumping is an optimization technique that adds an extra label to the type and property assertions so that the branch numbers that caused the tableau algorithm to add those assertions are tracked. Obviously, assertions that exist in the original ontology and the assertions that were added as a result of only deterministic rule applications will not depend on any branch. This means these assertions are direct consequence of the axioms in the ontology and affect every interpretation. If a clash found during tableau expansion does not depend on any non-deterministic branch, the reasoner will stop applying the rule as it is obvious that there is no way to fix the problem by trying different branches.

[6] When there is a maxCardinality restriction on a property and more values are provided for that property, reasoner is forced to assign equivalence between some of these values in order to satisfy the cardinality restriction. There might be multiple different combinations to select the individuals, thus the choice is non-deterministic, i.e reasoner tries every different possibility.

When the reasoner is known to use dependency directed backjumping (all existing DL reasoners —Racer, Fact, Pellet — use this technique), then looking at the last clash to explain an unsatisfiability is generally enough (though, one should verify this by examining the dependency set information of the clash).

Issues in Clash Detection:

(1) The clash information may be incomplete. Consider the case in which the inconsistency is due to all the different non-deterministic branches failing for different reasons. A simple concept description that illustrates this problem is $\mathtt{A} \sqsubseteq \mathtt{B} \sqcap \mathtt{C} \sqcap (\neg\mathtt{B} \sqcup \neg\mathtt{C})$. Concept `A` is unsatisfiable because it is either a subclass of $\neg\mathtt{B}$ or $\neg\mathtt{C}$ (due to the disjunction). However, neither is possible since they both cause a clash with other concepts in the conjunction. In this setting, it is not enough to present the last clash as it will not be accurate.

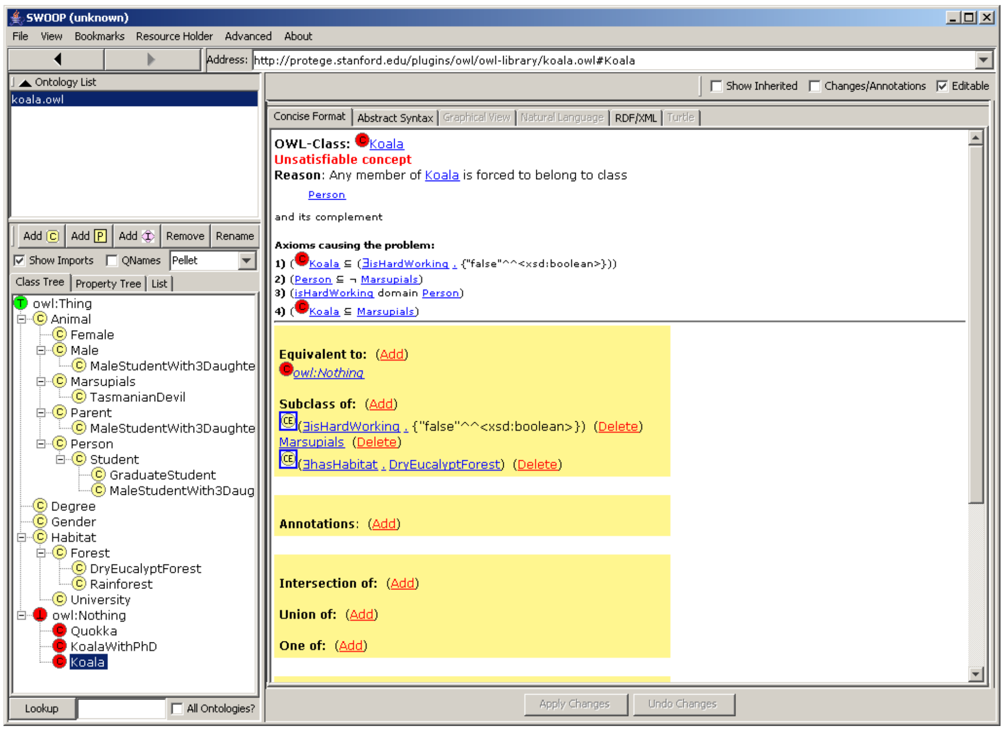
This problem can be overcome when all the clashes encountered are recorded and the dependency set of the last clash is examined to find the relevant set of clashes for the inconsistency. Unfortunately, in this case it is harder to understand the problem as the user is expected to look at all the different clash reasons.

(2) The clash presented does not *explain* the cause of the problem, since it fails to reveal the asserted definitions directly responsible for the clash. For example, while the property path (as seen in Figure 5), helps establish a connection between the unsatisfiable class in question and the root clash, it does not indicate reasons for the property successors i.e., whether the successor comes from an asserted `owl:minCardinality` or `owl:someValuesFrom` restriction, or a combination of an existential with an `owl:allValuesFrom` restriction. However, computing the clash information is relatively easy and does not introduce much overhead in terms of speed or memory consumption for the reasoner, as shown in the evaluation later.

This problem is resolved in the next subsection by computing the sets of support axioms.

4.1.2 Sets of support

As noted above, the clash information does not specify which set of axioms are causing this inconsistency — essential information for the user trying to fix the problem. Finding the source of the problem manually may still take some reasonable effort, especially when the descriptions in the ontology are complex. It is possible to extend the reasoner to keep track of the source axioms for assertions in a way similar to the dependency sets discussed earlier (see Figure 6), and the procedure is described below.


SWOOP (unknown)
File View Bookmarks Resource Holder Advanced About
Address: http://protege.stanford.edu/plugins/owl/owl-library/koala.owl#Koala
Ontology List
koala.owl
Add C Add P Add I Remove Rename
Show Imports QNames Pellet
Class Tree Property Tree List
owl:Thing
Animal
Female
Male
MaleStudentWith3Daughte
Marsupials
TasmanianDevil
Parent
MaleStudentWith3Daughte
Person
Student
GraduateStudent
MaleStudentWith3Daug
Degree
Gender
Habitat
Forest
DryEucalyptForest
Rainforest
University
owl:Nothing
Quokka
KoalaWithPhD
Koala
Lookup All Ontologies?
Show Inherited Changes/Annotations Editable
Concise Format Abstract Syntax Graphical View Natural Language RDF/XML Turtle
Apply Changes Undo Changes

OWL-Class: Koala
Unsatisfiable concept
Reason: Any member of Koala is forced to belong to class
Person
and its complement

Axioms causing the problem:
1) (Koala ⊆ (∃isHardWorking . {"false"^^<xsd:boolean>}))
2) (Person ⊆ ¬ Marsupials)
3) (isHardWorking domain Person)
4) (Koala ⊆ Marsupials)

Equivalent to: (Add)
owl:Nothing

Subclass of: (Add)
(∃isHardWorking . {"false"^^<xsd:boolean>}) (Delete)
Marsupials (Delete)
(∃hasHabitat . DryEucalyptForest) (Delete)

Annotations: (Add)

Intersection of: (Add)

Union of: (Add)

One of: (Add)

Fig. 6. The set of axioms that support the inconsistency of Koala concept is displayed in debug mode.

Tableaux Tracing Procedure: Before we explain the tracing procedure, we briefly revisit how tableaux reasoning works: Satisfiability checking of a class involves building a model of the class using a set of tableaux expansion (trigger) rules. The process starts with explicit assertions in the ontology and new axioms (inferences) are added by the reasoner depending on the triggering rule.

For example, suppose we have two assertions:

```
{x rdf:type C, C rdfs:subClassOf D}
```

The reasoner will add the additional assertion

```
{x rdf:type D}
```

which will depend on both of the above assertions. Also, as noted in the previous section, various internal modifications are done by the reasoner such as normalization and absorption that combine and transform axioms in different ways in order to optimize clash detection.

Our goal is to track and store the original source axioms from the ontology as they are modified and used throughout the tableaux expansion process. For this purpose, we extend the dependency sets used in the clash detection procedure to keep track of the axioms. As the reasoner continues applying the tableau rules, the axiom set for each assertion needs to be updated as well as the dependency set information. When an *inconsistency-revealing* clash is discovered, the axiom set is presented along with the clash information. This ensures that only the axioms directly relevant to the inconsistency are obtained. For example, suppose given class A, the clash detected is atomic, i.e., A is forced to be an instance of both: C and $\neg C$. In this case, the axiom set obtained via the dependency branches are only those responsible for making $A \sqsubseteq C$ and separately making $A \sqsubseteq \neg C$, and thus relevant to the contradiction.

The key elements of the current implementation are:

(1) While transforming axioms from the ontology into the data structures used internally by the reasoner, we precompute and store references between normalized versions of the data structures and the original axioms in an *ExplanationTable* (HashMap). During tableaux expansion in the reasoner i.e. type addition to an individual (unfolding), edge addition between individual (property restriction) etc., we refer to the *ExplanationTable* to obtain the corresponding axioms responsible for the addition. The *ExplanationTable* contains several tweaks to make sure that references are maintained and retrieved properly:
   - If the data structure corresponds to a list (say an intersection list in a subclass axiom), we store references for each component of the list
   - Whenever equivalent-class axioms (or domain/range/disjoint axioms) are converted to internal representations, we store references between the original axiom and the mapped counterparts.
   - During retrieval, if an axiom is not found in the *ExplanationTable*, we check if the negated version of the axiom is present.

(2) For absorption, we use the algorithm described in [16] which is a 7 step algorithm that absorbs GCI (or parts of GCIs) into primitive definitions in the KB. Tracking here entails storing axioms responsible for introducing elements in the absorption set (steps 1,3,4), and then references between the new absorbed primitive definition and these axioms (step 2).

(3) For internalization, we track the source of the unabsorbed components (after the absorption phase), and then store references between each component (converted to a disjunct) in the Universal Concept (UC) and the original axiom in *ExplanationTable*. When applying the UC to each node in the tableaux, we recover the axioms corresponding to each disjunct and add them to the dependency branch for each disjunction (Note: each disjunction introduces non-deterministic branches).

(4) On a more general note: in order to prevent axiom tracking from adversely affecting the normal operation of the reasoner, all tracking is done in a separate completion strategy. Also, if calls are to be made to the core methods of the reasoner, *mirror* methods are created which perform the dependency tracking, and *mirrors* are called instead of the original methods from this completion strategy.

The pseudo code for the tableaux tracing is given in Appendix A. For a detailed technical report on tableaux tracing, see [17].

Issues in Sets of Support:

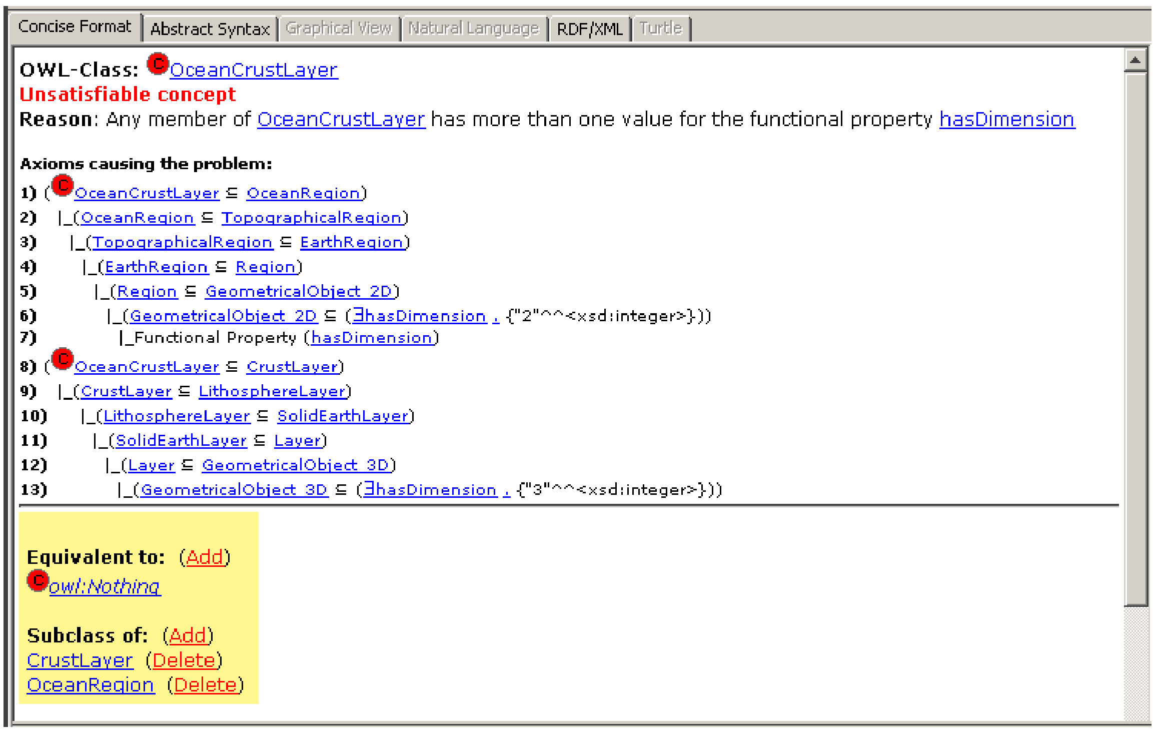
(1) The sets of support for a specific class (when determined minimally) coincide with the notion of Minimal Unsatisfiability Preserving Sub-TBox (MUPS) as defined in [14], i.e., they represent the smallest sub-TBox of the ontology which contains the contradiction for the class. Note that there may be more than one MUPS for a single class, since there may be different inconsistency-causing clashes responsible for its unsatisfiability. For example, consider a class defined by two subclass axioms: $\texttt{A} \sqsubseteq (\texttt{C} \sqcap \neg\texttt{C})$ and $\texttt{A} \sqsubseteq (\texttt{D} \sqcap \neg\texttt{D})$. In this case, each axiom is a distinct MUPS of class `A` and in order to resolve the bug in `A`, both axioms (each of the MUPS) need to be fixed separately. In our current solution, we determine only a single MUPS (sets of support) since typically in reasoners, tableaux expansion stops when the first inconsistency causing clash is found.

(2) Having found the sets of support for an unsatisfiable class, presenting the axioms in order to facilitate understanding of the problem is a separate challenge. For now, we have worked on a recursive ordering strategy which arranges axioms such that atleast one component in the RHS of the current axiom aligns with the LHS of the next. An example of this ordering is shown in Figure 7 for the unsatisfiable class `OceanCrustLayer` in the Sweet-JPL ontology set (discussed earlier).

    We are also working on an alternate technique to explain the contradiction by computing the subsumption trace [18]. We intend to study both approaches in more detail based on a thorough evaluation.

(3) The sets of support highlight the axioms responsible for the inconsistency of a single class. However, in an ontology with numerous unsatisfiable classes, it is critical to find *common* sets of support i.e. axioms that cause inconsistency for a large number of related classes since they essentially point to the root of the problem. This is difficult to achieve in practice using the glass box, since it requires finding sets of support for each unsatisfiable class in the ontology, which can be extremely time consuming for a large set of unsatisfiable classes.

    In order to resolve this problem, we explore black box alternatives to detect structural dependencies between unsatisfiable classes and focus on differentiating between *root* and *derived* unsatisfiable classes (see Section 4.2).



OWL-Class: OceanCrustLayer
Unsatisfiable concept
Reason: Any member of OceanCrustLayer has more than one value for the functional property hasDimension

Axioms causing the problem:

```
1) (OceanCrustLayer ⊆ OceanRegion)
2)   |_(OceanRegion ⊆ TopographicalRegion)
3)     |_(TopographicalRegion ⊆ EarthRegion)
4)       |_(EarthRegion ⊆ Region)
5)         |_(Region ⊆ GeometricalObject_2D)
6)           |_(GeometricalObject_2D ⊆ (∃hasDimension . {"2"^^<xsd:integer>}))
7)             |_Functional Property (hasDimension)
8) (OceanCrustLayer ⊆ CrustLayer)
9)   |_(CrustLayer ⊆ LithosphereLayer)
10)    |_(LithosphereLayer ⊆ SolidEarthLayer)
11)      |_(SolidEarthLayer ⊆ Layer)
12)        |_(Layer ⊆ GeometricalObject_3D)
13)          |_(GeometricalObject_3D ⊆ (∃hasDimension . {"3"^^<xsd:integer>}))
```

Equivalent to: (Add)
owl:Nothing

Subclass of: (Add)
CrustLayer (Delete)
OceanRegion (Delete)

Fig. 7. The set of axioms that contain the inconsistency of `OceanCrustLayer` are ordered to facilitate understanding of the problem by systematically leading the user to the source of the contradiction.

4.1.3 Related Problem: Explaining Subsumption

As an interesting sidenote, the problem of explaining subsumption is closely related to explaining contradiction for an unsatisfiable class. This is not surprising since in tableaux reasoners subsumption-check between a pair of atomic classes — $C \sqsubseteq D$ is reduced to a consistency-check of the complex class description $C \sqcap \neg D$. Thus, the minimal sets of support that entail the contradiction of $C \sqcap \neg D$ necessarily entail the subsumption of C by D, and can be used to explain non-trivial subsumptions to users.

4.2 Black Box Techniques

In black box techniques, the reasoner is used as an oracle for a certain set of questions (standard inferences such as satisfiability, subsumption etc.) and the asserted structure of the ontology is used to help isolate the source of the problems. Thus it has two advantages over the glass box approach: reasoner independence, i.e., you do not need a specialized, explanation generating reasoner, and avoiding the performance penalty (memory consumption) of glass box techniques.

Currently, the black box methods focus on detecting dependencies between unsatisfiable classes, i.e., identifying *root* and *derived* unsatisfiable classes.

4.2.1 Root/Derived Unsatisfiable Classes

For the purpose of debugging, we categorize unsatisfiable classes in an ontology into two types:

(1) *Root Class* - this is an unsatisfiable class in which a clash or contradiction found in the class definition (axioms) does not *depend on the unsatisfiability* of another class in the ontology. More specifically, the unsatisfiability bug for a root class cannot be fixed by *simply* correcting the unsatisfiability bug in some other class, instead, it requires fixing some contradiction stemming from its own definition. Example of a root class is: $Student \sqsubseteq (\geq 2) hasAdvisor \sqcap (\leq 1) hasAdvisor$.

(2) *Derived Class* - this is an unsatisfiable class in which a clash or contradiction found in a class definition either directly (via explicit assertions) or indirectly (via inferences) *depends on the unsatisfiability* of another class (we refer to it as the *parent* unsatisfiable class). Hence, this is a less critical bug in the sense that (in most cases) it can be resolved by fixing the unsatisfiability of this parent dependency. Note that there may be cases in which fixing the dependency bug reveals yet another unsatisfiability bug in the class, which needs to be resolved separately (making the derived class necessarily, but not sufficiently, dependent on the parent). Example of a derived class is: Class $GraduateStudent \sqsubseteq Student$, where *Student* is an unsatisfiable class itself, in this case, its parent.

We give formal definitions for the different types of unsatisfiable classes in terms of MUPS of an unsatisfiable class:

Definition 1 *(Root and Derived) C is a* ***derived*** *unsatisfiable class iff it satisfies the following condition:* $\exists i, j$ *s.t.* $MUPS_i(C) \supseteq MUPS_j(D)$ *where D is another unsatisfiable class in the ontology, in this case its parent. If C does not satisfy this condition, it is a* ***root*** *unsatisfiable class.*

Intuitively, a derived unsatisfiable class C depends on parent D if D being unsatisfiable makes C unsatisfiable. The converse is not guaranteed. [7]

[7] We note that in most cases that we have observed, unsatisfiable classes have only one MUPS making the converse hold as well i.e. for such cases, fixing the bug in the parent makes the derived class satisfiable as well.

4.2.2 Automating Dependency Detection

In this section, we present a dependency-detection algorithm that separates the root from the derived unsatisfiable classes in an ontology as provided by a reasoner. The algorithm consists of two parts: *asserted* dependency detection and *inferred* dependency detection and we describe each in detail in the following subsections.

For each unsatisfiable class in the ontology, the algorithm returns all its parent dependency classes along with the corresponding axioms that link this class to the parent. More specifically, the data structure returned is a set of tuples, where each tuple τ is recursively defined with fixed point:

$$\tau = (dep,\ axiom)$$

where, *dep* is a set of dependency sets dep', such that each dep' is a set of unsatisfiable classes that together cause the bug (could be a singleton set), and
axiom = associated axiom linking current class to dependency set

For example,

$$\tau(A) = (\{\{D\}, =\}, \{\{C, E\}, \sqsubseteq\})$$

implies the following:

- A has an equivalent class axiom relating it to its parent dependency D . This means that D being unsatisfiable is sufficient to cause A to be unsatisfiable, e.g. $A = D \sqcap (\leq 1p)$
- A has a subclass axiom relating it to parents C and E. This means *both* C and E should be unsatisfiable to make A unsatisfiable, e.g. $A \sqsubseteq (C \sqcup E)$

Asserted Dependency Detection: Structural Tracing

This phase is used to detect dependencies between unsatisfiable classes by analyzing the asserted structure of the ontology. It is divided into three stages:

- *Stage 1: Pre-processing* - given a class definition (considering its equivalence and subclass axioms), we obtain a set of all property-value chains inherent in these axioms, which terminate in a universal value restriction ($\forall$) on an unsatisfiable class. Here, we use the reasoner as the black box to report when the class is unsatisfiable. For example, consider the class definition A:
$$A = \exists p.\forall q.(B \sqcap (\exists r.C \sqcup \forall s.D))$$
If classes B, D are unsatisfiable, the following property-value chains (*allPC*) are obtained for A:
$$allPC = (p.q., B), (p.q.s., D_o).$$

Note that:

· The class D has a subscript 'o' to denote that a `union` (or non-deterministic branch) exists in its property chain ($p.q.s$), making this value restriction an 'optional' requirement (to be handled later in the post-processing stage).

· We also consider role hierarchy in constructing property-value chains, i.e., in the example above, if the property p has ancestor properties p_1, p_2, we expand the chains in our *allPC* set to:

$$((p/p_1/p_2).q., B), ((p/p_1/p_2).q.s., D_o)$$

Universal value restrictions on a property must be satisfied iff the property exists i.e. the class definition entails a ≥ 1 cardinality restriction on the property. In Stage 2 (dependency-tracing of a particular class), each time we discover the existence of a property, we check the *allPC* chains to ensure that the associated universal restriction is satisfied. However, note that we need to determine the set of *allPC* chains beforehand, since the non-localization of the class definition makes it difficult to verify all universal restrictions during tracing directly.

- *Stage 2: Dependency-tracing* - this is the core stage in which a recursive set of methods are used to extract all parent dependency unsatisfiable classes and the adjoining axioms given the original class definition (for a pseudo code of the tracing see **Appendix B**). The output contains a mixture of definite and optional dependency cases. **Note: In all the cases below, we use the reasoner as the black box to check if a class is unsatisfiable.**

We present the basic cases of the tracing approach.

Unsatisfiable class A is derived if:

(1) A (set of) equivalent/subClass axioms relate class A directly to unsatisfiable class B (B becomes its parent)

(2) A (set of) equivalent/subClass axioms relate class A to an intersection set, *any* of whose elements are unsatisfiable, e.g. $A = (B \sqcap C \sqcap \ldots \sqcap D)$, and one of $B, C, \ldots, D$ is unsatisfiable (any such unsatisfiable class becomes its parent)

(3) A (set of) equivalent/subClass axioms relate class A to a union set, *all* of whose elements are unsatisfiable, e.g. $A = (B \sqcup C \sqcup \ldots \sqcup D)$, and all $B, C, \ldots D$ are unsatisfiable (all such unsatisfiable classes become its parent)

(4) A (set of) equivalent/subClass axioms entail that class A has an existential ($\exists$) property restriction on an unsatisfiable class, e.g. $A = \exists(p, B)$ and B is unsatisfiable (B becomes its parent)

(5) A (set of) equivalent/subClass axioms entail that class A has a ($\geq$ 1) cardinality restriction on a property-chain, and the universal ($\forall$) value restriction on that chain is not satisfied (object/value of property chain becomes its parent)

(6) A (set of) equivalent/subClass axioms entail that class A has a ($\geq$ 1) cardinality restriction on a property, and the domain of the property is

unsatisfiable, e.g. $A \sqsubseteq (\geq 1p), domain(p) = B$, and B is unsatisfiable, making it the parent of A (similar *domain* check has to be made for every ancestor property of p)

(7) A (set of) equivalent/subClass axioms entail that class A has a ($\geq$ 1) cardinality restriction on an object property, and the range of its inverse is unsatisfiable, e.q. $A \sqsubseteq (\geq 1p), range(p^{-}) = B$, and B is unsatisfiable, making it the parent of A (similar *range* check has to be made for every ancestor property of p^{-})

An interesting case of the algorithm occurs when we find a pair of derived unsatisfiable classes such that one is marked as the parent of the other. Here, its unclear which, if any, of the classes is the *actual* parent. For example, if we have the equivalent class axiom $A \equiv B$ in the ontology and both classes, A and B, are unsatisfiable, its unclear whether A depends on B for its unsatisfiability, or vice versa, or if both classes depend on the axiom for their unsatisfiability. In such cases, simple ontology modifications may be used to reveal the dependency. In the above example, we remove the equivalence axiom from the ontology and test the satisfiability of A and B again using the reasoner. This gives us three possible outcomes:

(1) Both classes become satisfiable: This implies that there is a strong inter-dependence between the two classes which makes each class unsatisfiable. Hence we mark them both as non-derived tentatively (to be pruned in the later stage).
(2) One class becomes satisfiable while the other remains unsatisfiable: This implies that the former is derived from the latter (parent) class.
(3) Both classes stay unsatisfiable: We run the structural tracing algorithm on each class again to find alternate distinct dependencies.

We now list a straightforward, yet important lemma related to structural tracing:

Lemma 1 *Derived dependencies detected in this stage satisfy Definition 1.*
Proof 1 *For each unsat. class A above, we find an asserted axiom or a sets of axioms (say S_{AB}) which make it dependent (either by a subsumption relationship or a property-link) on another unsatisfiable class B. Thus we can form atleast one $MUPS(A)$ which contains any $MUPS(B)$ plus the set S_{AB}. Thus A is derived as per Definition 1.*

Note that the tracing approach described above does not consider nominals or transitive property relations. Hence, it can be considered as sound (i.e. it finds accurate dependencies between unsatisfiable classes), but not complete (does not find all dependencies).

- *Stage 3: Post-processing* - if the final dependency set contains an *optional* unsatisfiable class C_o, we check if *adjoining* (siblings within the same nested set) dependencies are definite i.e. without an 'o' tag. If they are, we simply remove the optional dependency C_o; else (if C_o is the sole dependency) we transform it to a C getting rid of the optional tag. We do this recursively, until all the optional unsatisfiable classes are either pruned out or transformed in the final dependency set.

For example, if the final dependency set for a class is:

$$dep = (\{\{C_o, D\}, =\}, \{\{E_o\} \sqsubseteq\})$$

we reduce it to:

$$dep = (\{\{D\}, =\}, \{\{E\}, \sqsubseteq\})$$

Optional classes are unsatisfiable class dependencies occurring in a disjunction (union set). Thus, they are guaranteed to cause unsatisfiability if and only if they are the sole reason for it, else they need to be pruned out of the final dependency set.

For detailed examples of the structural tracing algorithm, see [18].

Inferred Dependency Detection

The problem with detecting hidden dependencies in a KB with unsatisfiable classes is the masking of useful subsumption relationships in the TBox (since all unsatisfiable classes are subsumed by everything). Hence, given a TBox with unsatisfiable concepts, we consider a *subsumption safe* transformation as one which modifies the TBox by trying to get rid of all inconsistencies while attempting to preserve the *intended* subsumption hierarchy as much as possible. Here, we present only a heuristic approach that seems to work well in practice. [8]

The heuristic consists of two steps:

(1) For every axiom in the TBox that refers to a class of the form $\neg C$, where C could be atomic or complex, we replace $\neg C$ by a new atomic class C'.
(2) Similarly, for every axiom in the TBox that refers to a class of the form $\leq n.p$, where p is an OWL property, we replace $\leq n.p$ by a new atomic class P'.

There is an important aspect of the algorithm above which attempts to preserve subsumption relations in the underspecified KB, that is: every substitution is stored in a cache, and each time a new one is made, we check for subsumption (using a reasoner) between satisfiable terms in the original KB, and if found, we add corresponding relations between concepts in the transformed KB. For example, consider a TBox with the following 3 axioms:

(ax_1) $A = \neg C \sqcap \leq 1.p$

[8] An alternate approach to detect inferred dependencies is described in [2], wherein we remove a class and test the satisfiability of the related classes.

(ax_2) $B = \neg D \sqcap \leq 2.p$
(ax_3) $D \sqsubseteq C$

Here, we substitute $\neg C$ by C' and $\neg D$ by D'. Now, because of the subsumption of D by C in the original KB, we add the axiom $C' \sqsubseteq D'$ to the transformed KB. Similarly, after substituting $\leq 1.p$ by P_1 and $\leq 2.p$ by P_2, due to the subsumption relation $\leq 1.p \sqsubseteq \leq 2.p$, we add the axiom $P_1 \sqsubseteq P_2$ to the transformed KB. This way the subsumption relation $A \sqsubseteq B$ is preserved in the modified KB.

The motivation for the above approach is to remove well-known causes for concept unsatisfiability, i.e. class complements and max cardinality violations on a property as discussed in [1]. In fact, since both steps are independent, we perform any one step first and test subsumption before moving to the other.

Note that the above procedure is always sound since the monotonic nature of the logic (OWL-DL) implies that removing an axiom from the KB (or underspecifying it in the manner in which we have by reducing expressivity) cannot add a new subsumption relation. Though the procedure is incomplete because *safety* cannot be guaranteed and the original subsumption relation may get destroyed.

After applying the above transformation, we can use the reasoner as the black box to classify the (relevant part of) KB to detect hidden dependencies between concepts not caught in structural tracing[9]. Moreover, if the concepts turn out to be satisfiable, and hidden dependencies are revealed, we can use the transformations performed to pinpoint axioms which cause incoherence in the core (parent) unsatisfiable classes.

To elaborate, consider a simple TBox with two unsatisfiable classes A, B, and the following axioms:

(ax_1) $A = D \sqcap \exists p.D$
(ax_2) $A \sqsubseteq \neg D$
(ax_3) $B = C \sqcap \exists p.C$
(ax_4) $C \sqsubseteq D$

In this TBox, the unsatisfiability masks the subsumption of B by A (i.e. $B \sqsubseteq A$). If, however, we apply step 1 of the transformation above and reduce the TBox to a consistent form by modifying axiom ax_2 to $A \sqsubseteq D'$, the new TBox $T' \models B \sqsubseteq A$, and the hidden dependency is revealed. Moreover, we also know that axiom ax_2 is an integral part of the problem.

[9] Note that in general, if C, D are unsatisfiable classes in an ontology, and a fragment of the ontology (set of axioms) entails $C \sqsubseteq D$, then C is derived from D based on Definition 1, see Lemma 1.

5 Performance Evaluation

5.1 *Preliminary Glass Box Evaluation*

For evaluating the runtime and memory performance of the glass box debugging techniques, we applied them to a set of real-world and hand-crafted ontologies that covered a wide range of unsatisfiable class cases, such as due to *inverse* or *transitive* properties, GCIs etc. The ontologies can be found at `http://www.mindswap.org/ontologies/debugging`. We processed each ontology using Swoop/Pellet and compared the performance of the normal reasoning mode vs. the debugging mode which computed the clash and sets of support. In all cases, the algorithm returned sound and complete results, i.e., the exact sets of support axioms containing the contradiction for the unsatisfiable class were obtained.

Key results of our evaluation are summarized in the table below:

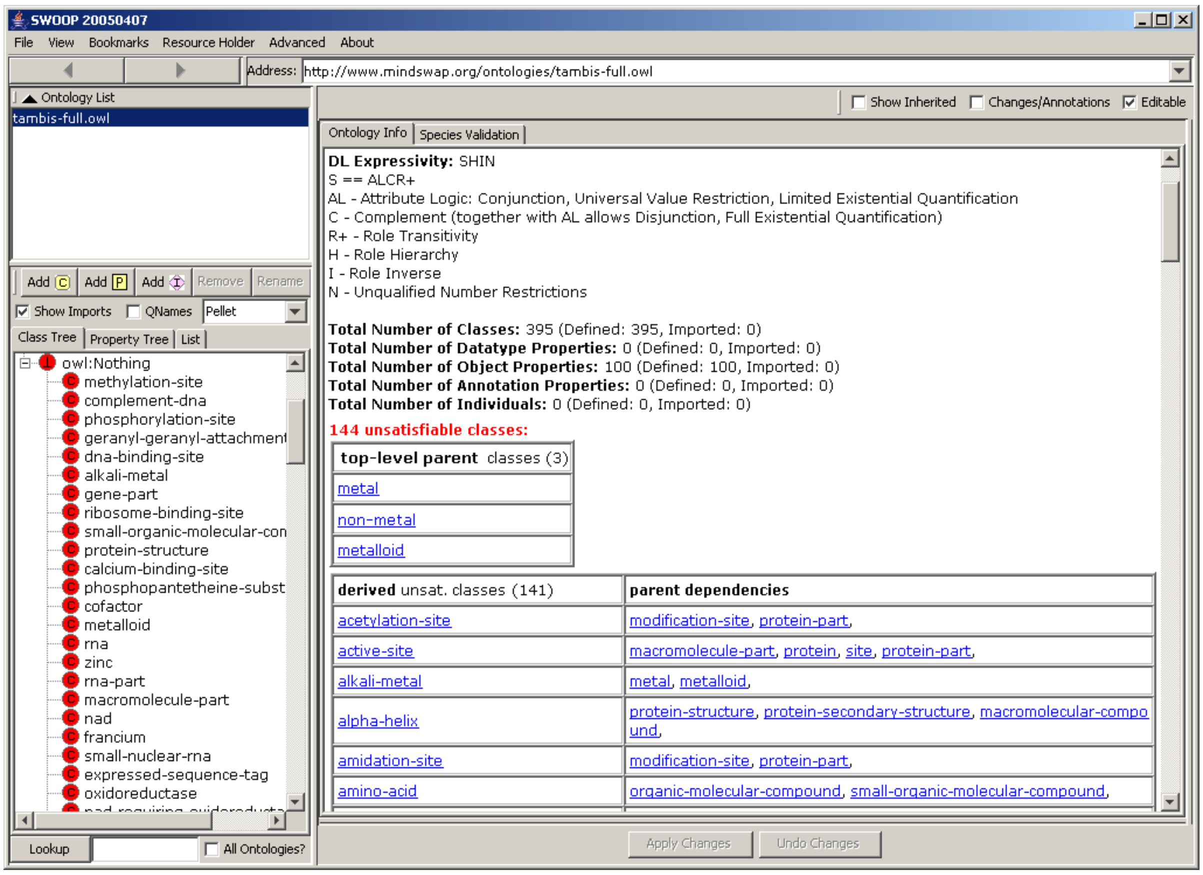
Ontology / Expressivity	**No. of Classes / Unsat. Classes**	T_{Norm} / T_{Debug} /	$M_{ExpTable}$
`koala.owl` / ALCON(D)	21 / 3	40 / 40 ms	780 KB
`mad-cows.owl` / ALCHION(D)	54 / 1	301 / 330 ms	1.029 MB
`sweet-jpl.owl` / ALCHIO(D)	1391 / 1	16.453 / 16.895 s	8.84 MB
`tambis.owl` / SHIN	395 / 144	4366 / 4416 ms	3.65 MB
`university.owl` / SION(D)	28 / 8	70 / 72 ms	786 KB

In the table, T_{Norm}/T_{Debug} corresponds to the time taken for Pellet to process the entire ontology (i.e., check satisfiability of each class and compute the subsumption hierarchy) in the normal/debug modes respectively, and $M_{ExpTable}$ corresponds to the approximate memory consumption of the class *ExplanationTable* that is used to track and store the source axioms. As these preliminary results show, dependency tracking in Pellet based on our algorithm introduces some memory overhead but only marginally increases the running time of the normal reasoning procedures.

5.2 *Preliminary Black Box Evaluation*

For black box debugging evaluation, we decided to use the Tambis OWL ontology since its expressive (SHIN), moderately-sized (395 classes) and contains a large number of unsatisfiable classes (144). Many of the unsatisfiable classes depend in simple ways on other unsatisfiable classes, so that a brute force going down the list correcting each class in turn is unlikely to produce correct results, or, at best, will be pointlessly exhausting. In one case, three changes repaired over seventy other unsatisfiable classes. Given the highly non-local effects of assertions in a logic like OWL, it is not sufficient to take on defects in isolation.

We implemented the black-box techniques in the Swoop, and carried out the analysis and debugging of Tambis. Running the structural tracing algorithm on Tambis identified 111 *derived* classes with at least one parent dependency, and 33 classes with no parent dependencies (potential *root* classes) in approx. 20 ms. This was a significant result, the problem space was pruned by more than 75% enabling us to direct our attention on a narrow set of unsatisfiabile classes, and moreover, for each unsatisfiabile class, we obtained the dependency relation (via axioms) leading to its corresponding parents, which were presented in the Swoop UI.



owl:Nothing
- methylation-site
- complement-dna
- phosphorylation-site
- geranyl-geranyl-attachment
- dna-binding-site
- alkali-metal
- gene-part
- ribosome-binding-site
- small-organic-molecular-con
- protein-structure
- calcium-binding-site
- phosphopantetheine-subst
- cofactor
- metalloid
- rna
- zinc
- rna-part
- macromolecule-part
- nad
- francium
- small-nuclear-rna
- expressed-sequence-tag
- oxidoreductase

DL Expressivity: SHIN
S == ALCR+
AL - Attribute Logic: Conjunction, Universal Value Restriction, Limited Existential Quantification
C - Complement (together with AL allows Disjunction, Full Existential Quantification)
R+ - Role Transitivity
H - Role Hierarchy
I - Role Inverse
N - Unqualified Number Restrictions

Total Number of Classes: 395 (Defined: 395, Imported: 0)
Total Number of Datatype Properties: 0 (Defined: 0, Imported: 0)
Total Number of Object Properties: 100 (Defined: 100, Imported: 0)
Total Number of Annotation Properties: 0 (Defined: 0, Imported: 0)
Total Number of Individuals: 0 (Defined: 0, Imported: 0)
144 unsatisfiable classes:

top-level parent classes (3)
metal
non-metal
metalloid

derived unsat. classes (141)	**parent dependencies**
acetylation-site	modification-site, protein-part,
active-site	macromolecule-part, protein, site, protein-part,
alkali-metal	metal, metalloid,
alpha-helix	protein-structure, protein-secondary-structure, macromolecular-compound,
amidation-site	modification-site, protein-part,
amino-acid	organic-molecular-compound, small-organic-molecular-compound,

Fig. 8. **Debugging Tambis**: Identifying Root/Parent and Derived Unsatisfiable Classes. Note that the time taken for the structural tracing was less than a second.

Out of the remaining 33 potential *root* classes, we applied the inferred dependency detection algorithm and uncovered equivalence between all of them.

This was both, a surprising and interesting result, and due to the fact that all 33 classes shared the same structure (defined equivalent to the same intersection set) out of which 3 were asserted as mutually disjoint in the ontology thus causing the contradiction; while the remaining 30 classes were inferred to be equivalent to the above 3 classes making them unsatisfiable as well. In this case, not only were hidden dependencies detected, but the disjoint axioms causing the incoherence were obtained as well, exposing the classes whose definition contained the axioms (in this case, the classes `metal, non-metal, metalloid`).

As a result, we now have an efficient iterative procedure to remove all the unsatisfiability bugs in the ontology: at each point, we focus *solely* on fixing all the root classes (if any), or top-level *parent* classes identified by the pinpointed axioms, which effectively fixes a large set of directly derived class bugs. However, doing so might reveal additional contradictions and a new set of unsatisfiable classes. We then use the dependency detecting algorithm again to obtain new potential roots (with problematic axioms) and repeat the fixing process iteratively till no unsatisfiable classes are left in the ontology.

6 Usability Evaluation

In order to determine the practical use and efficiency of the debugging features implemented in Swoop/Pellet, we conducted a small usability-study as follows:

(1) We selected twelve subjects in all having at least 9 months of experience with OWL and with an understanding of description-logic reasoning that varied greatly (novices to experts)

(2) Each subject received a 20-30 minute orientation that covered:
- an overview of the semantic errors found in OWL ontologies (using examples of unsatisfiable classes)
- a brief tutorial of Swoop, demonstrating its key browsing, editing and search features
- a detailed walkthrough of the glass and black box debugging support in Swoop using a set of toy ontologies

(3) The twelve subjects were *randomly* divided into 4 groups of three subjects each:
   (a) **Group 1:** Subjects in this group received *no debugging support* at all, i.e., only a list of unsatisfiable classes in the ontology was displayed by the reasoner
   (b) **Group 2:** Subjects in this group received only glass-box debugging support, i.e., the *clash information and sets of support axioms* for an unsatisfiable class were displayed
   (c) **Group 3:** Subjects in this group received only black-box debugging

support, i.e., the *root/derived unsatisfiable classes* were displayed along with the facility for class-expression highlighting and reasoning[10]

(d) **Group 4:** Subjects in this group received *both clash/SOS and root/derived* debugging support

(4) Having formed the groups, each subject was given three erroneous ontologies – `University.owl`, `Sweet-JPL.owl` and `mini-Tambis.owl` (in random order), any of which the subject had not seen before. The subject was asked to debug the ontologies in Swoop (*independently*) using only the features assigned to the group the subject belonged to. The following guidelines were observed during the debugging process:
   - The subject was given 20 minutes (maximum) to debug an ontology. He/she was free to stop the debugging process at any time, or to extend it beyond the specified time-period as desired
   - While debugging any unsatisfiable class, the subject was asked to write down a brief explanation of the contradiction for that class (in his/her own words) based on the understanding of the problem. In addition, the subject was asked to suggest a likely fix for the problem where possible
   - The tool automatically counted the number of entity definitions viewed, and the changes made to the ontology during the entire debugging process, both of which we considered as key sub-tasks

(5) Having obtained the times taken by a subject for debugging each of the three ontologies, we took the average of the times (for the group) in order to nullify the expertise and skill factor of the subject (note that the subjects were randomly assigned to the groups as mentioned earlier).

(6) Finally, after working on all three ontologies, the subject was handed a questionnaire to elicit feedback on the entire debugging experience using Swoop

Key properties of the ontologies used in the study were:

Ontology	**Total Classes**	**Unsat. Classes**	**Root/Derived**
`1. University.owl`	28	8	5/3
`2. Sweet-JPL.owl`	1537	1	1/-
`3. mini-Tambis.owl`	183	30	5/25

Our hypothesis was as follows:

(1) *The clash information and sets-of-support (SOS) axioms provided is better than no support for all the erroneous ontologies, i.e., the subject will take less time to understand and fix the errors correctly using the*

[10] The latter can be considered as a black-box debugging service since the reasoner is used as an oracle to extract useful inferences about specific class-expressions.

clash/SOS (in this case computed using the glass box technique) than without any debugging support. The reason for this is that the information would help pinpoint and illustrate the source of the contradiction for the unsatisfiable class.

(2) *For a relatively small no. of unsatisfiable classes (i.e., ontologies 1 and 2), the clash/SOS information will outperform both, the root/derived information (in this case computed using the black-box technique) and the no support, and perform not too worse than the full-debug support.* The reason for this is that the subject could potentially distinguish the root from the derived classes by manually inspecting the sets of support for each class, thus reducing the impact of automatically identifying them.

(3) *For a large no. of unsatisfiable classes with different roots (i.e., ontology 3) the root/derived debugging support will match the performance of the clash/SOS information, and additionally, the full-debug support will be clearly better than either of the two.* The reason for this is that manually discovering the root/derived classes would often be hard and time-consuming in such cases, and the black-box automated detection technique would help narrow down the problem space tremendously.

The results of the usability study are summarized in the graph in Figure 9. The graph displays the average time taken (in mins) per group to debug all the errors for each of the three ontologies given (Note: 'F' represents a Failure to debug the error).

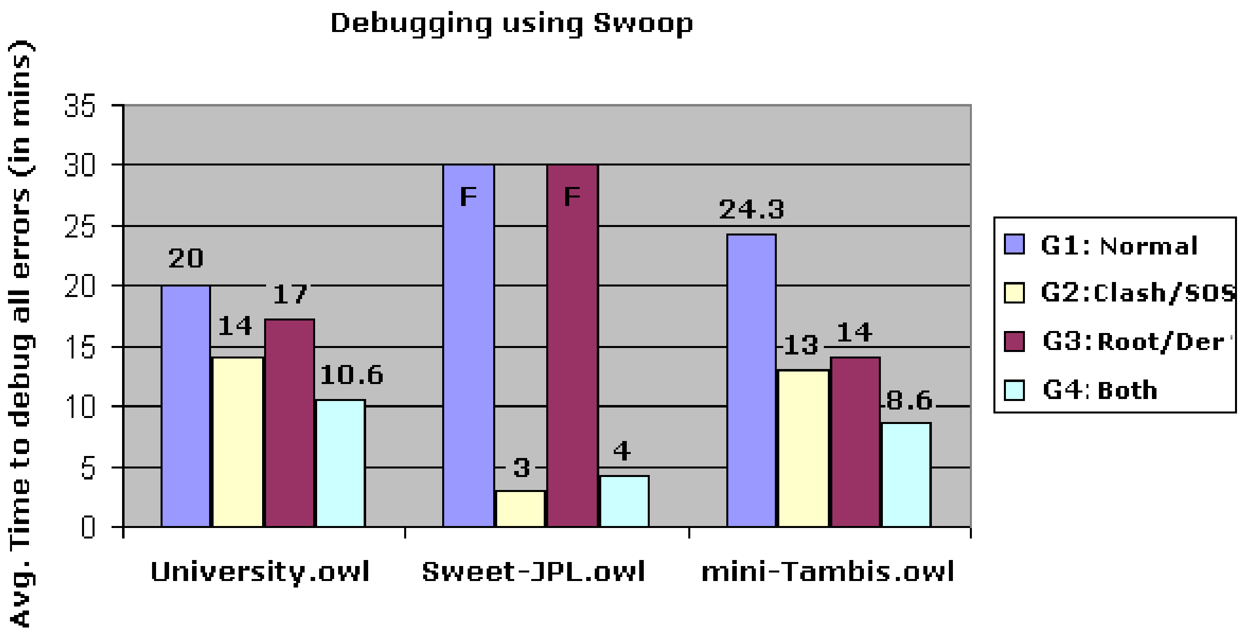


Fig. 9. Results of the Debugging Usability Study:

As seen from the graph, the statistical results obtained are in agreement with hypothesis (1), i.e., a 2-tailed T-test shows that debugging with clash/SOS is significantly better than debugging without it for $p \approx 0.01$. While the timings for the ontologies are in agreement with hypothesis (2) and (3), given the small size of the study, a measure of the statistical results was not significant for verifying those hypothesis. We plan to conduct a more extensive evaluation to fully justify them.

For `University.owl`, all subjects were able to identify the erroneous axiom(s) for each of the unsatisfiable classes within the time period given, however, only 1 subject in normal/root-derived (black box) mode was able to understand the cause of the problem, whereas, 2/3 using the clash/SOS (glass box) and 3/3 using the full-debug mode were able to understand and explain the problem correctly. Also, the time taken to fix all the errors using the full-debug mode was approx. half of that taken using the normal-mode.

In the case of `Sweet-JPL.owl`, without support axioms no subject was able to understand the cause of the error due to the highly non-local interactions in the large ontology, whereas, using the axioms, each subject took under 5 minutes to understand and fix the problem correctly.

Interestingly, the black-box support performed nearly similar to the glass-box in the case of `mini-Tambis.owl` since subjects found it easier to debug the roots identified by the black-box even without the sets of support, than to discover the roots without the black-box and with the sets of support, given the large number of unsatisfiable classes. Also, for this ontology, the subjects in the normal mode fixed only 2/3 roots in the time period given, i.e, they could not fully complete the debugging.

We learnt some useful lessons based on our observations of the debugging process and the feedback given by the subjects:

- **For Group 1 – no-debug mode:**
  - · 3/3 subjects rated the hypertextual navigation (with back/next history) as the most useful feature for understanding relationships and causes of errors in the ontology
  - · 2/3 subjects found ontology changes immensely useful to identify erroneous axioms by using a trial-and-error strategy
  - · The *Show References* search feature was never used by any of the subjects. Based on their comments, it seemed that they were unclear about its use and significance. Interestingly, a subject in Group 3 found this feature very helpful, implying that the feature either supports a different debugging style (to that of the authors in this mode) or requires better presentation.
- **For Group 2 – clash/SOS mode (only glass box):**
  - · 3/3 subjects rated the sets of support axioms as the most useful feature

  · 2/3 subjects felt that a proof-style layout of the support axioms with intermediate inferences shown as well would help explain the problem better.
  · Overall, 6 subjects were exposed to the glass-box (3 from this group and 3 using the full-debug mode), and they were divided on the significance of the clash information. While half the subjects used the clash information to pinpoint relevant components of the support axioms, the other half found the information poorly presented and redundant given the support axioms, pointing us to a definite area of improvement.
- **For Group 3 – root/derived mode (only black box):**
  · 1/3 subject used the *Show-References* feature extensively to aid debugging, especially for mini-Tambis.owl, where discovering a commonly-used property restriction helped understand the source of the contradiction for a set of unsatisfiable classes
  · 1/3 subject felt that the Class-Expression (CE) support needed to be made more effective by allowing arbitrary queries on CEs
  · 2/3 subjects suggested displaying the number of derived dependencies that arise from each root to highlight the more significant roots
- **For Group 4 – full-debug mode:**
  · 3/3 subjects felt that it was the *combination* of the clash/SOS presentation and the root/derived identification and not one specific feature that was useful to debug all errors in the ontology

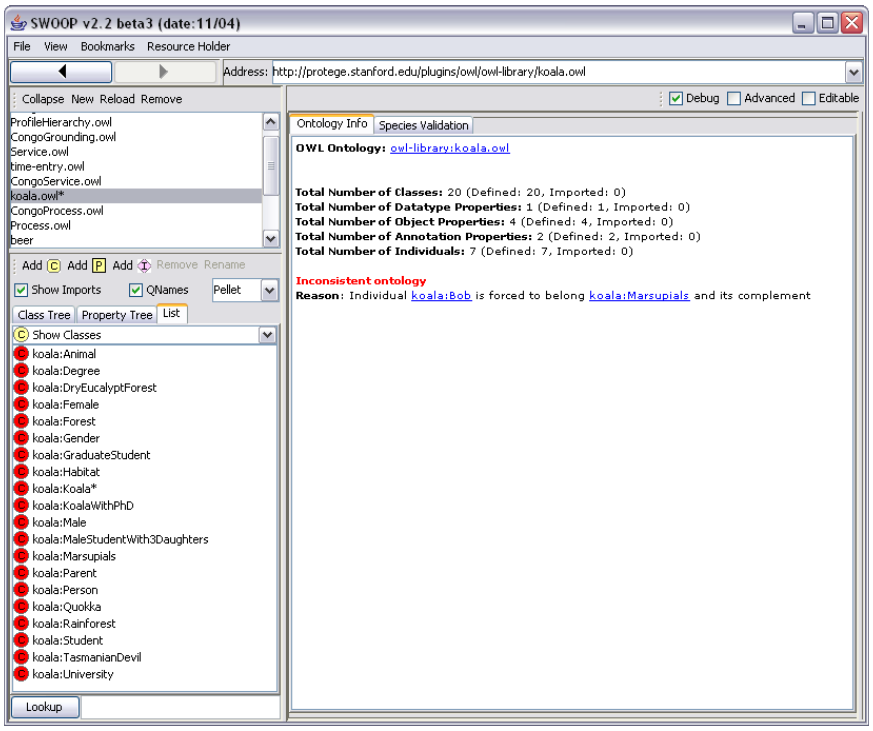
Overall, the response of the subjects in the study was very encouraging. Many relative newcomers to OWL and description-logic were impressed by the fact that they were able to correctly fix all the errors in ontologies which they had not seen before within the specified time period. Experts in the field who had real-world experience in OWL ontology modeling and manual debugging were surprised at how easy the task of debugging was made for them.

7 Future Work

7.1 Diagnosing Inconsistent KBs

As noted in Section 2, an inconsistent ontology is, on the face of it, very difficult for a reasoner to do further work with. Since anything at all follows from a contradiction, reasoner results breakdown on an inconsistent KB. In Figure 10, we can see how the naive application of the reasoner to an inconsistent ontology marks *all* the classes as unsatisfiable, even though, in this example, no class is “in itself” unsatisfiable.

Many of the techniques discussed in the prior section are, in fact, applicable to the diagnosis of inconsistent KBs, with a few slight twists. This should be no surprise as unsatisfiability detection is performed by attempting to generate an inconsistent KB. Thus, the glass box techniques for diagnosing unsatisfiable concepts in principle work to help diagnose inconsistent KBs.



http://protege.stanford.edu/plugins/owl/owl-library/koala.owl

ProfileHierarchy.owl
CongoGrounding.owl
Service.owl
time-entry.owl
CongoService.owl
koala.owl*
CongoProcess.owl
Process.owl
beer

koala:Animal
koala:Degree
koala:DryEucalyptForest
koala:Female
koala:Forest
koala:Gender
koala:GraduateStudent
koala:Habitat
koala:Koala*
koala:KoalaWithPhD
koala:Male
koala:MaleStudentWith3Daughters
koala:Marsupials
koala:Parent
koala:Person
koala:Quokka
koala:Rainforest
koala:Student
koala:TasmanianDevil
koala:University

OWL Ontology: owl-library:koala.owl

Total Number of Classes: 20 (Defined: 20, Imported: 0)
Total Number of Datatype Properties: 1 (Defined: 1, Imported: 0)
Total Number of Object Properties: 4 (Defined: 4, Imported: 0)
Total Number of Annotation Properties: 2 (Defined: 2, Imported: 0)
Total Number of Individuals: 7 (Defined: 7, Imported: 0)

Inconsistent ontology
Reason: Individual koala:Bob is forced to belong koala:Marsupials and its complement

Fig. 10. When the ontology is inconsistent due to an assertion about an individual, all the classes end up inconsistent because reasoner can infer anything from an inconsistent ontology.

First, we provide a categorization of the different kind of reasons for inconsistent KBs:

(1) *Inconsistency of Assertions about Individuals* There are no unsatisfiable classes in the ontology but there are conflicting assertions about one individual, e.g., an individual is asserted to belong to two disjoint classes or an individual has a cardinality restriction but related to more individuals.
(2) *Individuals Related to Unsatisfiable Classes* There is an unsatisfiable class description and one individual is asserted to belong to that class or has an existential restriction to the unsatisfiable.
(3) *Defects in Class Axioms Involving Nominals* It might be the case that inconsistency is not directly caused by type or property assertions, i.e., ABox assertions, but caused by class axioms that involve nominals, i.e., TBox axioms. Nominals are simply individuals mentioned in owl:oneOf and owl:hasValue constructs. As an example consider the following set of axioms:
```
MyFavoriteColor = { Blue }
PrimaryColors = { Red, Blue, Yellow }
```

MyFavoriteColor $\sqsubseteq \neg$PrimaryColors

These axioms obviously cause an inconsistency because the enumerated MyFavoriteColor and PrimaryColors share one element, i.e., individual named Blue, but they are still defined to be disjoint. The final effect is similar as defining an individual to belong to an unsatisfiable concept but there is no direct type assertion in the ontology.

Depending on the type of inconsistency there are different options to be taken. For the first type of inconsistency, clash information would point to the individual that contains the problem (as already seen in Figure 10) and the details pane for the individual would flag the inconsistent concept expression. For the second type of inconsistency, removing the assertions about individuals would immediately reveal the unsatisfiable concept because all the other classes would now be satisfiable. Of course, there may be more than one defect in the ontology and each of these inconsistencies need to be solved separately in order to fix the overall problem.

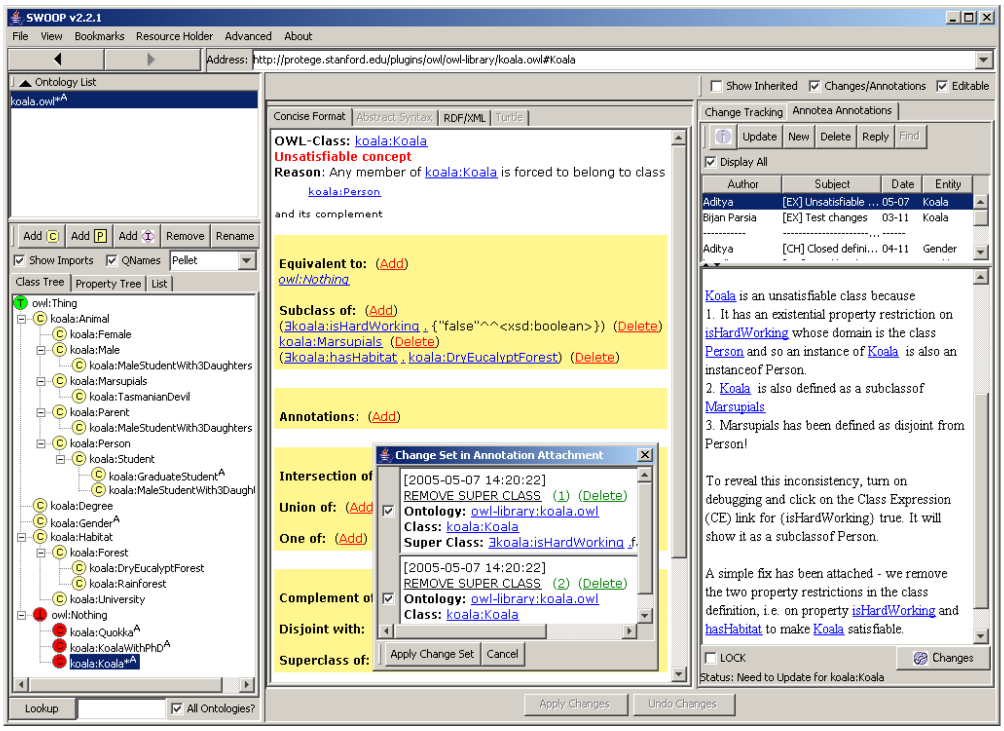
The third type of inconsistency is very different in nature because even removing all the assertions about individuals from the reasoner would not solve the problem. It is required to get rid of the problematic class axioms in order to make the ontology consistent. In this case, we need to make use of the glass box techniques to find the set of supporting axioms for the problem and try to fix the problem by examining this information along with the asserted facts.

7.2 Exploring remedies

Thus far we have focused on bug detection and diagnosis, that is, the initial information gathering phase of the debugging process. That phase is focused on helping the modeler *understand* the problem. Once there is understanding, then the modeler needs to take action. However, often there are a number of possible alternative actions (or sets of actions) that would correct the bug, or, in spite of all the debugging information supplied by the system, the source of the problem is unclear. At this point, a programmer will tend to start experimenting with possible changes. Part of good debugging support for OWL ontologies is making such experimentation safe, easy, and effective.

Swoop has an ontology versioning feature that supports ad hoc undo/redo of changes (with logging) coupled with the ability to checkpoint and archive different ontology versions. Such a feature can play a vital role in ontology debugging. Consider the scenario in which a user starts with an inconsistent ontology version, performs a set of changes in succession (undoing and redoing as necessary), in order to reach a final consistent version. Here the change logs give a direct pointer to the source of inconsistency. The checkpointing allows the user to switch between versions easily exploring different modeling alternatives.

Alternately, if the user has two different ontology versions, one consistent and the other inconsistent, a *diff* between the versions can be performed using Swoop's Concise Format Renderer in order to determine possible change paths between the versions. By examining these change paths, and noting the common bug-producing changes, users can find and eliminate erroneous entity-definitions and axioms in the ontology.



SWOOP v2.2.1
File View Bookmarks Resource Holder Advanced About
Address: http://protege.stanford.edu/plugins/owl/owl-library/koala.owl#Koala
Ontology List
koala.owl*A
Add C Add P Add I Remove Rename
Show Imports QNames Pellet
Class Tree Property Tree List
owl:Thing
koala:Animal
koala:Female
koala:Male
koala:MaleStudentWith3Daughters
koala:Marsupials
koala:TasmanianDevil
koala:Parent
koala:MaleStudentWith3Daughters
koala:Person
koala:Student
koala:GraduateStudent
koala:MaleStudentWith3Daughters
koala:Degree
koala:Gender
koala:Habitat
koala:Forest
koala:DryEucalyptForest
koala:Rainforest
koala:University
owl:Nothing
koala:Quokka
koala:KoalaWithPhD
koala:Koala*A
Lookup All Ontologies?
Concise Format Abstract Syntax RDF/XML Turtle
Show Inherited Changes/Annotations Editable
Change Tracking Annotea Annotations
Update New Delete Reply Find
Display All
Apply Change Set Cancel
Apply Changes Undo Changes
LOCK Changes

OWL-Class: koala:Koala
Unsatisfiable concept
Reason: Any member of koala:Koala is forced to belong to class koala:Person and its complement

Equivalent to: (Add)
owl:Nothing

Subclass of: (Add)
(∃koala:isHardWorking . {"false"^^<xsd:boolean>}) (Delete)
koala:Marsupials (Delete)
(∃koala:hasHabitat . koala:DryEucalyptForest) (Delete)

Annotations: (Add)

Intersection of
Union of: (Add
One of: (Add)
Complement of
Disjoint with:
Superclass of:

Change Set in Annotation Attachment

[2005-05-07 14:20:22]
REMOVE SUPER CLASS (1) (Delete)
Ontology: owl-library:koala.owl
Class: koala:Koala
Super Class: ∃koala:isHardWorking .f

[2005-05-07 14:20:22]
REMOVE SUPER CLASS (2) (Delete)
Ontology: owl-library:koala.owl
Class: koala:Koala

Author	Subject	Date	Entity
Aditya	[EX] Unsatisfiable ...	05-07	Koala
Bijan Parsia	[EX] Test changes	03-11	Koala
----------	----------------------...	------	
Aditya	[CH] Closed defini...	04-11	Gender

Koala is an unsatisfiable class because
1. It has an existential property restriction on isHardWorking whose domain is the class Person and so an instance of Koala is also an instanceof Person.
2. Koala is also defined as a subclassof Marsupials
3. Marsupials has been defined as disjoint from Person!

To reveal this inconsistency, turn on debugging and click on the Class Expression (CE) link for {isHardWorking} true. It will show it as a subclassof Person.

A simple fix has been attached - we remove the two property restrictions in the class definition, i.e. on property isHardWorking and hasHabitat to make Koala satisfiable.

Status: Need to Update for koala:Koala

Fig. 11. Collaborative Debugging and Repair using Annotea

Once a series of changes has proven effective in removing the defect and seems sensible, the modeler can use Swoop's integrated Annotea [19] client to publish the set of changes plus a commentary (see Figure 11). Other subscribers to the Annotea store can see these changes and commentary in context they were made, apply the changes to see their effect, and publish responses. These exchanges persist, providing a repository of real cases for subsequent modelers to study.

Finally, in Swoop we have a provision to store and compare OWL entities via the Comparator panel. Snapshots of Items can be added to this placeholder at any time and that view will persist there until the user decides to remove or replace them at a later stage. Upon adding an entity, a time-stamped snapshot of it is saved (with hyperlinks and all), thus providing a reference point for future tasks. The significance of the Comparator was amply demonstrated in

Section 3 (see Figure 3) where we studied the unsatisfiability bug in the class `Koala` by saving snapshots of related classes and expressions in the Comparator, and browsing them to identify the exact cause for the bug.

8 Conclusion

In this paper, we have presented a suite of features integrated into the OWL ontology browser and editor — Swoop, and the description logic tableaux reasoner — Pellet, designed to aid modelers in debugging inconsistency related errors in their ontologies. Key features include the detection of clash/sets of support axioms responsible for an unsatisfiable class, and the identification of root/derived unsatisfiable classes. Two orthogonal debugging techniques are used for these features - *glass box* and *black box*, the former optimized to compute the clash/SOS directly, while the latter better suitted to identify dependencies in a large no. of unsatisfiable classes.

We are, in general, focused on the whole user experience, and the reactions from the user base in the conducted usability-study has been very encouraging. Moving around the ontology should be trivial and the hypertextual navigation supports this – a feature unanimously welcomed by all users in the study. Also, instead of shifting into a completely separate debugging pane, augmenting the existing display with additional cues allowed users to follow the problem using familiar techniques. We made alterations both easy and safe, allowing users to experiment more freely with expressions and axioms in the ontology. Finally, users felt that the debugging features not only helped them understand problems in the tested ontologies, but also gave them an insight into more generic and commonly misunderstood outcomes of ontology modeling (such as proper usage of equivalence vs. subsumption, intersection vs. union etc).

We intend to continue extending and improving the glass and black box techniques to debug all types of semantic defects in OWL-DL ontologies, and where possible, exploring strategies to handle modeling/style defects. The results will be validated by conducting an extensive evaluation on real-world ontologies, coupled with more in-depth usability studies.

Our experience has been that the process of debugging can assist the understanding of an ontology, both by providing motivation and by providing guidance. For example, the Tambis ontology is large, complex, and describes an unfamiliar domain. But each author quickly learned quite a bit about the ontology (and even of the domain) by trying to understand the various unsatisfiabilities. This suggests that using similar techniques for identifying and displaying dependencies would be effective in helping users explore and come to understand new ontologies.

9 Acknowledgments

We would like to thank all the participants of the usability study and in particular, Jennifer Golbeck, who helped us with the setup, design and analysis of the experiment.

This work was completed with funding from Fujitsu Laboratories of America – College Park, Lockheed Martin Advanced Technology Laboratory, NTT Corp., Kevric Corp., SAIC, National Science Foundation, National Geospatial-Intelligence Agency, DARPA, US Army Research Laboratory, NIST, and other DoD sources.

References

[1] B. Parsia, E. Sirin, A. Kalyanpur, Debugging owl ontologies, in: Proceedings of the 14th International World Wide Web Conference (WWW2005), Chiba, Japan, 2005.

[2] A. Kalyanpur, B. Parsia, E. Sirin, Black box debugging of unsatisfiable concepts, in: Proceedings of the Eighteenth International Workshop on Description Logics DL 2005 (To Appear), 2005.

[3] M. Dean, G. Schreiber, OWL Web Ontology Language Reference W3C Recommendation. http://www.w3.org/tr/owl-ref/.

[4] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, Owl pizzas: Common errors & common patterns from practical experience of teaching owl-dl, in: European Knowledge Acquisition Workshop (EKAW), 2004.

[5] A. Kalyanpur, B. Parsia, J. Hendler, A tool for working with web ontologies, in: International Journal on Semantic Web and Information Systems, 2004.

[6] E. Sirin, B. Parsia, Pellet: An owl-dl reasoner, in: International Semantic Web Conference (ISWC2004), Japan (Poster), 2004, http://www.mindswap.org/2003/pellet.

[7] R. Cornet, A. Abu-Hanna, Evaluation of a frame-based ontology. a formalization-oriented approach, in: Proceedings of MIE2002, 2002.

[8] D. McGuinness, Explaining reasoning in description logics, Ph.D. thesis, New Brunswick, New Jersey (1996).
URL `citeseer.ist.psu.edu/mcguinness96explaining.html`

[9] A. Borgida, E. Franconi, I. Horrocks, D. McGuinness, P. Patel-Schneider, Explaining alc subsumption, in: Proceedings of the International Workshop on Description Logics - DL-99, 1999.

[10] T. Liebig, O. Noppens, Ontotrack: Combining browsing and editing with reasoning and explaining for owl lite ontologies, in: Proceedings of the 3rd International Semantic Web Conference (ISWC) 2004, Japan, 2004.

[11] D. McGuinness, P. Patel-Schneider, Infrastructure for web explanations, in: Proceedings of 2nd International Semantic Web Conference (ISWC2003), 2003.

[12] D. McGuinness, R. Fikes, J. Rice, S. Wilder, The chimaera ontology environment., in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)., 2000.

[13] K. Baclawski, C. Matheus, M. Kokar, J. Letkowski, P. Kogut, Towards a symptom ontology for semantic web applications., in: International Semantic Web Conference, 2004, pp. 650–667.

[14] S. Schlobach, R. Cornet, Non-standard reasoning services for the debugging of description logic terminologies, in: Proceedings of IJCAI, 2003, 2003.

[15] H. Wang, M. Horridge, A. Rector, N. Drummond, J. Seidenberg, A heuristic approach to explain the inconsistency in owl ontologies, in: 8th Intl. Protg Conference (Accepted Abstract), 2005.

[16] I. Horrocks, Implementation and optimisation techniques, in: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003, pp. 306–346.

[17] A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca-Grau, Tableaux tracing(Technical Report).
URL `http://www.mindswap.org/papers/tableauxTracing.pdf`

[18] A. Kalyanpur, B. Parsia, E. Sirin, Black box debugging of unsatisfiable classes. http://www.mindswap.org/papers/iswc05-debugging.pdf.

[19] J. Kahan, M.-R. Koivunen, E. Prud'Hommeaux, R. Swick, Annotea: An open RDF infrastructure for shared web annotations, in: Proc. of the WWW10 International Conference, 2001.

10 Appendix A

Tableaux Reasoning Procedure with tracing integrated [11]

(1) Pre-processing:

(a) Normalization:

[11] This is a simplified version of the algorithm without tracing support for inverse/transitive roles, role hierarchy etc. However, it is possible to directly extend the principle idea behind the algorithm to cover those cases.

```
        (i) for each axiom (Ax) in the OWL Ontology,
              • normalize Ax and store reference between Ax and norm(Ax) in
                the Explanation-Table
    (b) Absorption:
        (i) for each GCI $C \sqsubseteq D$ in the OWL Ontology,
              • create GCI set G = $\{\neg D, C\}$
              • obtain axiom-set $S_{ax}$ corresponding to G from Explanation-Table
              • if G can absorb into primitive definition $A \sqsubseteq B$,
                then create new partial definition $A \sqsubseteq G'$ (G' is GCI corre-
                sponding to current G) and store reference between this defin-
                ition and $S_{ax}$ in Explanation-Table
              • if $C, \neg C \in G$ is replaced by $E$,
                then $S_{ax} = S_{ax}+$ axioms responsible for $C/\neg C \equiv E$
              • if G cannot be absorbed,
                then store reference between G and $S_{ax}$ in Explanation-Table
    (c) Internalization:
        (i) for each *unabsorbed* GCI $C \sqsubseteq D$,
              • get axiom-set $S_{ax}$ corresponding to GCI from Explanation-Table
              • create *conjunct* corresponding to GCI: $(D \sqcup \neg C)$
              • build Universal Concept (UC) = UC $\sqcap (D \sqcup \neg C)$
              • store reference between *conjunct* and $S_{ax}$ in Explanation-Table
    (d) Dependency Maintenance Initialization:
        (i) create map *depends* for each node $x$ in the tableaux (step 2 below),
            with (key, value) pairs:
            key := label added to $x$
            value := dependency set DS, {branch-no || source-axioms}
            (below we refer to the axioms component directly as $DS(x, label)_{axioms}$
       (ii) also create $DS$ for each edge $R(x, y)$ added to the tableaux


(2) Tableaux Expansion to check Satisfiability of class $C$:

    (a) create individual node $x$ corresponding to $C$, i.e., $L(x) = \{C\}$
    (b) while either clash found in deterministic branch or tableaux is complete,
        (i) apply *unfolding rule* to $x$, i.e. add concept label D,
              • $DS(x, D)_{axioms}$ += axioms responsible for adding $D$ to $L(x)$
                (obtained from Explanation-Table)
       (ii) apply *someValues rule* to $x$, i.e. add edge label $R$ from node $x$ to
            node $y$,
              • $DS(R(x, y))_{axioms}$ += axioms responsible for adding edge $x \rightarrow$
                $y$ (obtained from Explanation-Table)
      (iii) (..similar tracing while applying other completion rules..)
    (c) if atomic clash is detected, i.e. both $A, \neg A \in L(x)$
        then return *sets-of-support* = $node_x{:}DS(A)_{axioms} \cup node_x{:}DS(\neg A)_{axioms}$
```

11 Appendix B

Structural Tracing Pseudo-code (Stage 2):

---

function *traceClass (C)* → return **parent** dependency dep:

```
(1) if (C is satisfiable) return $\phi$;

(2) dep = $\phi$;

(3) if (C is atomic) dep += {C};

(4) else (C is complex),
    (a) if C is of form $(C_0 \sqcap C_1 \sqcap ..C_n)$,
            for each unsatisfiable element $C_i$ of the intersection,
                if ($C_i$ is atomic), dep += {$C_i$};
                else dep += traceClass(C);
    (b) if C is of form $(C_0 \sqcup C_1 \sqcup ..C_n)$,
            dep' = $\phi$,
            for each element $C_i$ of the union,
                if ($C_i$ is satisfiable), dep' = $\phi$, break;
                if ($C_i$ is atomic), dep' += {$C_i$};
                else dep' += traceClass(C);
            dep += dep';
    (c) if C is of form $\exists R.D$,
            propChain += {R};
            if (D is unsatisfiable),
                if (D is atomic) dep += {D};
                else dep += traceClass(D);
    (d) if ($\exists D$ s.t. $D \sqsubseteq \exists R^{-}.C$) and (D is unsatisfiable),
            if (D is atomic) dep += {D};
            else dep += traceClass(D);
    (e) if C is of form $\geq 1.R$, or of form $\exists R.\{I\}$
        (i) propChain += {R};
        (ii) if $\forall PCchain$ (pre-computed) starts with propChain,
                dep += terminal class (value) of $\forall PCchain$;
        (iii) if (R has domain E) and (E is unsatisfiable),
                if (E is atomic), dep += {E};
                else dep += traceClass(E);
            (repeat steps (i-iii) for equivalent and super-properties of R)
        (iv) if R has inverse Q,
                if (Q has domain F) and (F is unsatisfiable),
                    if (F is atomic), dep += {F};
                    else dep += traceClass(F);
            (repeat step for equivalent and super-properties of Q)
```

```
(5) **for each** equivalent class D of C,
        dep += *traceClass*(D);
(6) **for each** superclass D of C,
        dep += *traceClass*(D);
(7) return dep;
```