Debugging OWL Ontologies

Bijan Parsia
University of Maryland
8400 Baltimore Ave., Suite 200
College Park, MD 20742
bparsia@isr.umd.edu

Evren Sirin
Dept. of Computer Science
University of Maryland
College Park, MD 20742
evren@cs.umd.edu

Aditya Kalyanpur
Dept. of Computer Science
University of Maryland
College Park, MD 20742
aditya@cs.umd.edu

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 detecting and diagnosing errors in OWL ontologies. Typically these are restricted to the mere detection of, for example, unsatisfiable concepts. We have integrated a number of simple debugging cues generated from our description logic reasoner, Pellet, in our hypertextual ontology development environment, Swoop. These cues, in conjunction with extensive undo/redo and Annotea based collaboration support in Swoop, significantly improve the OWL debugging experience, and point the way to more general improvements in the presentation of an ontology to new users.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Answer/reason extraction*

General Terms

Human Factors

Keywords

OWL, ontology engineering, explanation, Semantic Web

1. INTRODUCTION

Now that OWL 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, 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.

In this paper, we discuss the problem of *debugging* OWL-DL ontologies. In particular, we focus on the diagnosis and correction of *unsatisfiable concepts*. Unsatisfiable concepts 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).[1] 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.

When new modelers encounter realistic cases of unsatisfiability, they are often a loss at what to do. The tool has told them there is a problem, but given them no help in fixing it. This has two negative general consequences inhibiting adoption and effective use of OWL ontologies: either developers tend to under specify their concepts to "avoid" error (at least, to avoid "fatal" error) or they give up on ontologies altogether.

We believe that good debugging support will give users control and a *sense* of control over their modeling. This encourages them to experiment more freely with expressions, but also helps them come to understand their ontologies through the debugging process. As when debugging programs, mere identification of where an exception or error occurs is not sufficient for generally successful debugging as confidence that a fix is a successful fix depends on the programmers general understanding of the code.

In this paper, we describe some first steps in providing debugging support for unsatisfiable concepts in the Swoop OWL ontology browser and editor. 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.

2. DETECTING DEFECTS

Before diagnosing a problem with an ontology one has to *detect* the problem. Reliably detecting all semantic defects (at least, in principle) is a core motivation for limiting the expressiveness of knowledge representation formalisms to that with tractable — or "pratical" — decision procedures. It is also helpful and familiar to programmers to see

[1] In the file naming scheme in this paper, we bind the prefixes 'owl' to 'http://www.w3.org/2002/07/owl#', 'rdf' to 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', and 'rdfs' to 'http://www.w3.org/2000/01/rdf-schema#'.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

a list of errors and warnings as such lists propose an *agenda* for working with the ontology. Of course, the trap here is becoming so focused on fixing "compiler" errors that you fail to test whether you met the specification. In general, it is easier to fix a syntax error than correct a subtle bug in the program's logic or a race condition in a threaded application. In some cases, the difference is due to the difficulty of finding the defect that is the real cause of the problem.

In Section 2, we discuss several classes of errors for OWL ontologies in terms of the standard techniques for detecting these errors and for presenting them to a user.

Debugging programs can be done with nothing but a text editor and print statements, but compilers, lint tools, symbolic debuggers, tracers, inspectors, code browsers, and all the various convenience tools that modern programming environments supply make a difference in both the success of and the satisfaction with detecting and fixing bugs. For ontologies, the situation is similar, except that ontology languages and their semantics are, as a rule, alien to developers. Thus, the tools have to go much further in organizing and presenting the information supplied by the reasoner and existing in the ontology. We are developing our OWL ontology browser/editor, Swoop, toward being something akin to an integrated development environment (IDE) for ontologies. Instead of a programming environment model, we have taken the "Web" part of the Web Ontology Language to heart and used the familiar Web browser interface as our primary inspiration. As we show below, with the right cues, hypermedia traversal proves to be an effective debugging modality.

2.1 Unsatisfiable Concepts

Like many OWL editors, Swoop uses a description logic reasoner to determine which named concepts in the ontology are unsatisfiable. Typically, if the ontology can be processed at all, the reasoner can check each concept in the ontology for satisfiability. The process of satisfiability checking causes the class tree to rearrange itself, if necessary, and various inferred relationships to be associated with a class definition. In Figure 1, the unsatisfiable concepts show up with a red tinted icon in the class tree and are displayed as subclasses of the empty concept, owl:Nothing. In the definition display of an unsatisfiable class (the right pane in Figure 1), the class is shown to be equivalent to owl:Nothing.

In the toy koala ontology[2], this display is fairly effective. There are three unsatisfiable, rather simple concepts. Effective browsing support is probably sufficient for debugging these concepts with ease. Swoop's Web browser-like, hypermedia-driven interface makes it fairly easy to probe and detect the problematic classes — all of which contain exactly the same modeling error, in two cases, by sharing the exact same problematic expression. The situation is considerably different when faced with the 144 unsatisfiable concepts of an OWL version of the Tambis ontology. The OWL was generated by a conversion script and a number of errors crept in during that process. Many of the unsatisfiable concepts depend in simple ways on other unsatisfiable concepts, so that a brute force going down the list correcting each concept 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

[2]http://protege.stanford.edu/plugins/owl/owl-library/koala.owl ontology is used in the Protégé tutorial

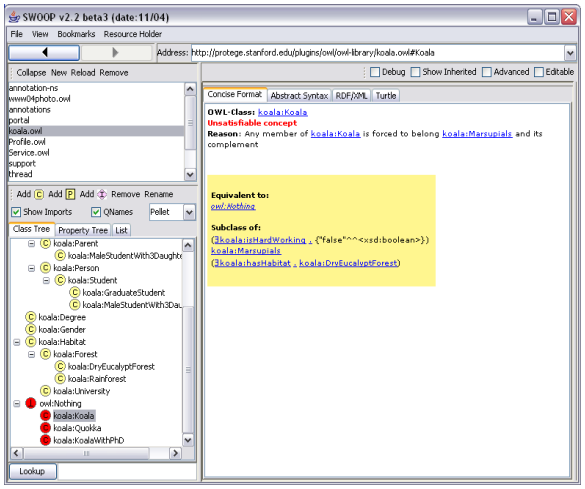

Figure 1: The class Koala has been found to be unsatisfiable, that is, a subclass of owl:Nothing.

classes. Given the highly non-local effects of assertions in a logic like OWL, it is not sufficient to take on defects in isolation.

By definition, if a program can detect whether a class is unsatisfiable, then it is a reasoner. Satisfiability checking just *is* an inference service. In our work, we tend to use our tableau based description logic reasoner, Pellet. Pellet has a number of advantages: It natively supports OWL, including a repairable subset of OWL Full; it has extensive support for XML Schema datatypes; it has ABox (a.k.a., instance) support; it covers the broadest range of OWL DL of any reasoner that we know, including both SHIN(D), SHON(D), SHIO(D), and various subsets of their union, SHOIN(D) (a.k.a., OWL DL); it is open source and in active, public development. The last is very important for certain debugging strategies which require access to the internals of the reasoner, as we will discuss in the next section. Pellet is not as mature or developed or, as a consequence, fast for many tasks as Racer, FaCT, or FaCT++. It is fast enough for a number of realistic and real ontologies so that lack of responsiveness is not a problem for our experimentation.

2.2 Inconsistent Ontologies

Inconsistent ontologies 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. 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.

2.3 Syntax, Species, and Complexity

Syntactic issues loom large in OWL for a number of reasons including the baroque exchange syntax, RDF/XML,

the use of URIs (and their abbreviations), and the fact that, for OWL DL, there is yet another layer of syntactic structure on top of the corresponding RDF graph. To make matters worse, OWL DL imposes a number of restrictions 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. However, mere species validation does not tell the whole story. The species distinctions were supposed to track expressivity well enough to give an indication of the type of reasoner that may be used. OWL Full is undecidable and there was no real production experience with building reasoners for such a logic. Most interesting subsets of OWL DL have proof theories, practical algorithms, and actual optimized implementations. Ordinarily, OWL DL reasoners cannot handle OWL Full ontologies, however, not all OWL Fullisms are pernicious to a DL system. For example, often a missing type assertion can be guessed from the context, given a background assumption that the modeler is trying to stay in OWL DL. Furthermore, a subset of OWL Full seems to be, even without serious "repair", amenable to standard description logic techniques, or can be made so. Determining those cases, and heuristically repair the other ones, has emerged as a surprising important tool in the current OWL debugging kit. There is pressure in many communities to stay inside OWL DL or even OWL Lite.

Finally, the OWL species are very coarse grained. The description logic community has long categorized various description logics by constructing mnemonic names that encode the precise expressivity of the particular logic, e.g., the logic SH extended by inverse roles is SHI and again with qualified cardinality restrictions is SHIQ. Understanding what the minimal logic one's KB falls into seems to be a help, not only in simply understanding one's KB, but also for diagnosing performance issues, or even the mere applicability of certain reasoners.

2.4 Subtler Defects

Some defects thus far discussed are not necessarily defects. There are occasions for defining an unsatisfiable concept, and clearly whether the species your ontology falls into is a *bug* or a *feature* is a matter for the modeler to decide. Similarly, there are inference services that a reasoner can provide that can help detect features of the ontology that may or may not be a defect, depending on the modeler's intent. 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. So, if for some reason the expected or desired (non)subsumption or class membership does not hold, the reasoner can detect and report this. But it cannot distinguish between desirable (non)subsumptions and undesired ones. Thus, the problem of organizing the subsumptions for analysis is more difficult than the problem of organizing the set of unsatisfiable concepts in the OWL version of the Tambis ontology, with the exception of the subsumption hierarchy itself. Even there, requiring modelers to simply inspect a large hierarchy is unhelpful.

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. 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 (which, as with many debugging techniques, can slow things down, if only by voiding optimizations). 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.)

3.1 Glass box techniques

3.1.1 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

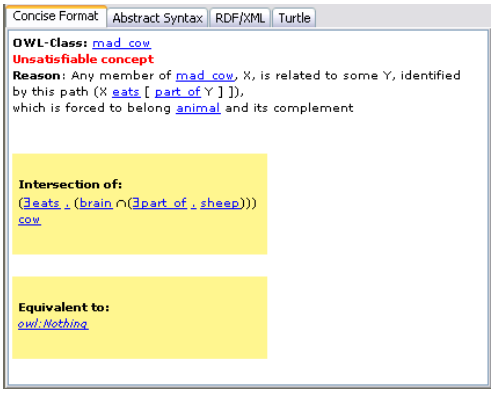

Figure 2: The explanation of unsatisfiability for class `Mad Cow` includes the description of an anonymous individual created by the reasoner. It is easy to see the connection between the path that identifies the individual and the existential restrictions in the `Mad Cow` definition.

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 2 that shows an example from the Mad Cow ontology[3].

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.

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[4], 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.

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). Of course, it is still possible that the inconsistency is due to the fact that all the different non-deterministic branches failed 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 $\neg$`B` or $\neg$`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. This is due, of course, to the harder nature of the problem itself.

3.1.2 *Sets of support*

The clash information provides hints about the cause of the inconsistency. However, it 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.

Naturally, the assertions in the ontology will depend on themselves only. New assertions added by the tableau rules will depend on a union of axioms which needs to be computed according to 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. As the reasoner continues applying the tableau rules the explanation sets for each assertions needs to be updated as well as the dependency set information. When an *inconsistency-revealing* clash is discovered (as explained above) the explanation set can also be presented along with the clash information. Figure 3 shows the set of axioms found by the reasoner as the source of the unsatisfiability for concept `koala:Koala`.

Presenting the set of support axioms provides more insight about the problem, but unfortunately it is much harder

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

[4]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.

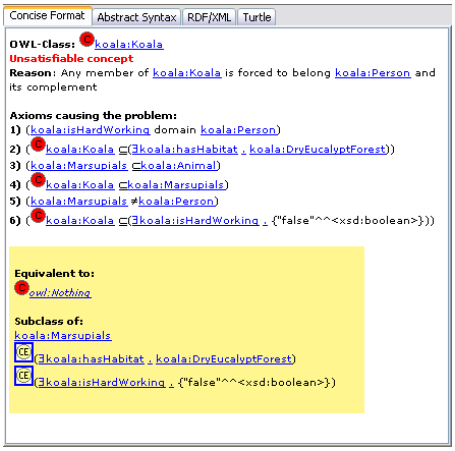

Figure 3: The set of axioms that support the inconsistency of Koala concept is displayed in debug mode.

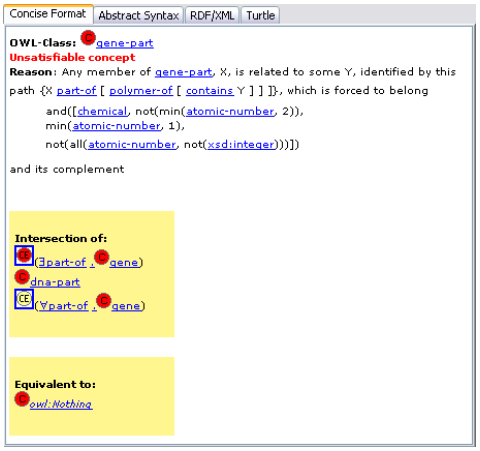

Figure 4: The class `gene-part` is unsatisfiable on two counts: its defined as an intersection of an unsatisfiable class (`dna-part`) and an unsatisfiable class expression (∃`partof.gene`), both highlighted using red tinted icons.

to compute compared to the clash information. The reasoner needs to maintain extra data structures to track the source and this introduces additional memory and computation consumption. But a bigger issue here, as noted in the previous section, is the internal modifications done by the reasoner such as normalization and absorption that combines and transforms the axioms from the ontology. Normalization induces a less critical problem because it is still possible to establish the relation between the normalized class expression and the class expression used in the ontology. However, the absorption method combines axioms from different parts of the ontology to create simpler subclass relations in order to eliminate General Concept Inclusion (GCI) axioms that substantially decrease the performance of reasoner. Disabling absorption optimization makes it relatively easier to compute the support sets. However, handling ontologies with complex structure, such as the Tambis ontology, is not feasible at all without absorption. In this case, it is more practical to use the black-box techniques as explained in the next section.

3.2 Black box techniques

3.2.1 *Browsing and rendering concepts*

As noted earlier, ontology engineering tools have to go a long way in presenting the information supplied by the reasoner in a form conducive to understanding and debugging the ontology. 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.

In addition, 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 4 (the Tambis ontology), note how simply looking at the class definition of `gene-part` makes the reason for the inconsistency apparent: it is an subclass of the inconsistent class `dna-part` and the inconsistent class expression ∃`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 5). 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 5, 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 ∃`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), 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.

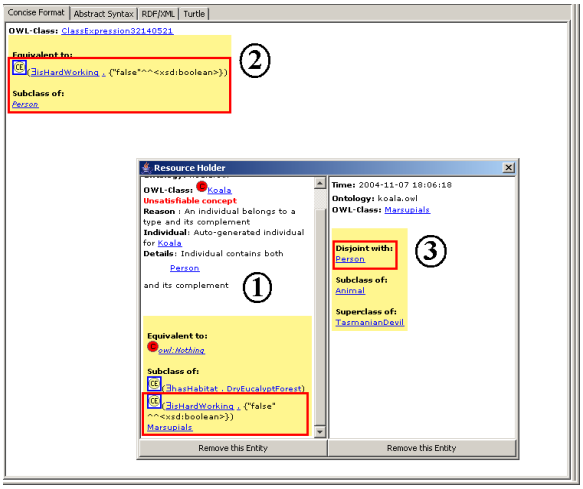

Figure 5: The class `Koala` is unsatisfiable because (1) `Koala` is a subclass of ∃`isHardWorking.false` and `Marsupials`; (2) ∃`isHardWorking.false` is a subclass of `Person`; and (3) `Marsupials` is a subclass of ¬`Person` (disjoint)

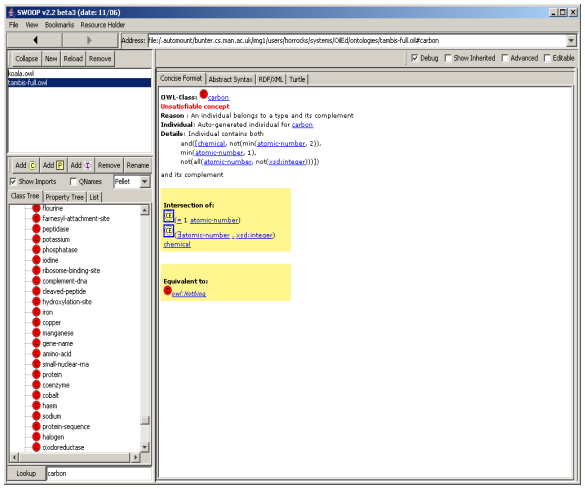

Figure 6: The class `Carbon` in the Tambis ontology is independently unsatisfiable, and is a root cause of numerous other (dependent) inconsistencies in the ontology.

While promising, there are severe limitations with this straightforward navigational approach (similar to the problems with naive structural analysis) due to the fact that constructs in expressive description logics can have highly non-local effects. Consider the case when the user has navigated (by following hyperlinks) to the apparent root cause (class) of a set of related inconsistencies, that is, one that, in our display, has no obvious dependencies on other unsatisfiable concepts. There are a few such cases in the Tambis ontology, for example, the class `Carbon` (Figure 6). Interestingly, not only is Carbon apparently a root, but it is a significant one. Removing the unsatisfiability bug in `Carbon` resolves the unsatisfiability of seventy-seven classes in the ontology. Here, determining the exact reason for the unsatisfiability is non-trivial, given the clash explanation provided by the reasoner — all we know is an instance of `Carbon` is forced to be an instance of two disjoint classes (in this case, arbitrary class expressions). At this point, a glass box approach would seek to determine the set of support of the clash, which, presumably, would gather all the relevant axioms even if they did not explicitly mention `Carbon`. However, this gathering does not necessarily *situate* the axioms in the classes definition. Some black box techniques can help highlight these dependencies, such as:

1. Based on the *type* of clash causing the unsatisfiability (See Section 3.1.1 for the explanation of these types), one could conduct a relevant search to find likely clash generating candidates. Here it means to search for all known (asserted+inferred) *equivalents and complements* of the class `Carbon`.

2. Instead of simply checking the unsatisfiability of a single class expression in the entity definition (which is the default Swoop modality), one could check the unsatisfiability of a *group of expressions* that when taken together are likely to generate a clash. Various heuristics can be used to select such expressions, for e.g., grouping multiple restrictions on the same property.

3. By examining the *details* of the clash provided by Pellet, one could identify key entities and expressions listed in the details, and search for possible **`paths`** (graphs) explicitly defined in the ontology that link the current unsatisfiable class to those entities and expressions. Identifying such paths is critical in understanding how clashes generate and propagate.

These techniques can also be used with reasoners that do not have glass box support and will not acquire it. This may be simply because the reasoner is propriety or just closed source, or it might be too difficult to alter. More important, some ontologies may be too large and complex to be processed without full optimization. In this case, the loss of trace information is dwarfed by the loss of any information at all.

4. DIAGNOSING INCONSISTENT KBS

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.

However, one fundamental problem with inconsistent ontologies is that any conclusion can be drawn from an inconsistent set of premises. As a result, the satisfiability of concepts and concept expressions cannot be computed. Figure 7 shows the result of adding an instance to an individual concept `Koala` where all the classes in the ontology become unsatisfiable due to this inconsistency. The ontology view shows the reason explaining which individual is causing the problem but looking at the assertions for this individual will not reveal anything useful simply because all the concepts and concept expressions would be shown inconsistent. The inconsistency of the KB needs to be solved in order for the reasoner to provide useful information that would help to fix the problem!

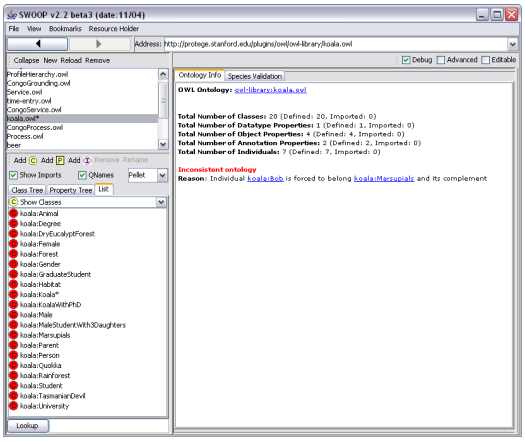

Figure 7: When the ontology is inconsistent due to an assertion about an individual, all the classes end up inconsistent because reasoner can infer anything from an inconsistent ontology.

Fortunately, the situation is not completely desperate as some improvements are possible depending on the type of problem in the KB. First, let us present 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:

   $\texttt{MyFavoriteColor} = \{ \texttt{Blue} \}$
   $\texttt{PrimaryColors} = \{ \texttt{Red, Blue, Yellow} \}$
   $\texttt{MyFavoriteColor} \sqsubseteq \neg\texttt{PrimaryColors}$

   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. In the first two cases, it is possible to remove all assertions about individuals while reasoning is being done. Since removing assertions about individuals would obviously fix the inconsistency, reasoner can process the ontology and present more reasonable results (note that the UI would still present the whole set of axioms and assertions from the ontology).

For the first type of inconsistency, clash information would point to the individual that contains the problem (as already seen in Figure 7) and the details pane for the individual would flag the inconsistent concept expression (similar to the inconsistent expressions in class descriptions as shown in Section 3.2.1. 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.

5. 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.

Once a series of changes has proven effective in removing the defect and seems sensible, the modeler can use Swoop's integrated Annotea client to publish the set of changes plus a commentary. 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 3.2.1 (see Figure 5) 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.

6. RELATED WORK

There has long been significant interest in explaining inferences to the non-sophisticated user when implementing reasoning services for Description Logic (DL) systems. In [4] 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 [2] wherein the authors use a *modified sequent calculus* to explain a tableaux based proof.

The *Inference Web* Infrastructure [6] comprised using a web-based registry for information sources, reasoners, etc., and a portable proof specification language for sharing explanations. So, reasoners (such as Pellet) which generate explanations using the above techniques could publish their explanation and explanation generating capability to interested clients (such as Swoop).

On the other hand, [8] propose non-standard reasoning algorithms (for ALC TBoxes) based on minimization of axioms using Boolean methods, and demonstrate promising results on the DICE terminology. Their approach which deals with *axiom and concept pinpointing* is directly related to our work, and we need to investigate it further for possible improvements to our solution.

[5] outline some critical usability issues in (deductive) DL systems based on real world empirical analysis. The paper covers a wide range of knowledge-access concerns such as explanations, error handling and pruning and identify key minimal usability requirements for each.

As for explanation support in Ontology editing toolkits, popular editors such as Protégé [7] and OilEd [1] provide very little or no explanation for reasoning and ontology debugging beyond what was discussed in Section 2. This is due, in part, to the lack of support for glass box techniques in available reasoners The recently released OntoTrack [3] goes a step further in this direction by giving "instant reasoning feedback" (using quasi natural language) for a limited fraction of OWL Lite, with a glass box approach is based on [2].

7. CONCLUSION & FUTURE WORK

In this paper we have presented a suite of features integrated into the OWL ontology browser and editor — Swoop, designed to aid modelers in debugging inconsistency related errors in their ontologies. We are, in general, focused on the whole user experience. Moving around the ontology should be trivial and the hypertextual navigation supports this. Also, instead of shifting into a completely separate debugging mode, we augment the existing display with additional cues that the user can follow using familiar techniques. We make alterations both easy and safe and encourage the sharing of experiences and successes. Thus far, reactions from our user base has been very encouraging.

We have focused on the obvious, and for obvious things the debugging cues work well. Tambis has proven to be a very useful challenge, but we need to gather a wider set of realistic cases, particularly those with non-local interactions. We plan to conduct formal user studies to test both the relative effectiveness of individual cues and of the holistic experience.

Our experience has been that the process of debugging can assist 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.

8. ACKNOWLEDGMENTS

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.

9. REFERENCES

[1] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a reason-able ontology editor for the Semantic Web. *Proceedings of KI2001, Joint German/Austrian Conference on Artificial Intelligence*, Sept. 2001.

[2] A. Borgida, E. Franconi, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Explaining alc subsumption. *In DL 99*, 1999.

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

[4] D. McGuinness. *Explaining Reasoning in Description Logics*. PhD thesis, New Brunswick, New Jersey, 1996.

[5] D. McGuinness and P. Patel-Schneider. Usability issues in knowledge representation systems. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, 1998.

[6] D. McGuinness and P. Patel-Schneider. Infrastructure for web explanations. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[7] N. Noy, M. Sintek, S. Decker, M. Crubezy, R. Fergerson, and M. Musen. Creating semantic web contents with Protégé-2000. *IEEE Intelligent Systems*, 2001.

[8] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. *Proceedings of IJCAI, 2003*, 2003.