Repairing Unsatisfiable Concepts in OWL Ontologies

Aditya Kalyanpur[1], Bijan Parsia[1], Evren Sirin[1], Bernardo Cuenca-Grau[2]*

[1] MINDLAB, University of Maryland, College Park, USA**
aditya@cs.umd.edu, bparsia@isr.umd.edu, evren@cs.umd.edu
[2] School of Computer Science, University of Manchester, UK
bcg@cs.man.ac.uk

Abstract. In this paper, we investigate the problem of repairing unsatisfiable concepts in an OWL ontology in detail, keeping in mind the user perspective as much as possible. We focus on various aspects of the repair process – improving the explanation support to help the user understand the cause of error better, exploring various strategies to rank erroneous axioms (with motivating use cases for each strategy), automatically generating repair plans that can be customized easily, and suggesting appropriate axiom edits where possible to the user. Based on the techniques described, we present a preliminary version of an interactive ontology repair tool and demonstrate its applicability in practice.

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 $\mathcal{SHOIN}(\mathcal{D})$ which is the basis of OWL-DL. As semantic descriptions in OWL ontologies become more complicated, ontology debugging becomes an extremely hard task for users, especially for those with little or no experience in description-logic-based knowledge representation. In such cases, ontology debugging tools are needed to explain and pinpoint defects in ontological definitions.

A common defect found in OWL Ontologies is unsatisfiable concepts, i.e., concepts which cannot have any individuals. Unsatisfiable concepts are usually a fundamental modeling error, and 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.

* This author is is supported by the EU Project TONES (Thinking ONtologiES) ref: IST-007603

** 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.

In our previous work, we have developed a suite of techniques for debugging unsatisfiable concepts in OWL Ontologies [6]. Our work focused on two key aspects: given a large number of unsatisfiable concepts in an ontology, identifying the *root* and *derived* unsatisfiable concepts from among them; and given a particular unsatisfiable concept in an ontology, extracting and presenting to the user the minimal set of axioms from the ontology responsible for making it unsatisfiable. We have provided an optimized implementation in the reasoner Pellet [14], a UI in the ontology editor Swoop [5], and proposed various enhancements in the display to improve the explanation of the cause of error. We have shown through a user study that these techniques are effective for debugging inconsistency errors in OWL ontologies.

However, while the emphasis was on pinpointing and explaining the errors in OWL ontologies, there was a lack of support for (semi-)automatically repairing or fixing them. Though in most cases, repairing errors is left to the ontology modeler's (/author's) discretion, and understanding the cause of the error certainly helps make resolving it much easier, bug resolution can still be a non-trivial task, requiring an exploration of remedies with a cost/benefit analysis, and tool support here can be quite useful.

In this paper, we present the following main contributions:

- We enhance the information that drives the repair process by modifying our algorithm to capture the part(s) of axiom(s) responsible for an error (section 3.2)
- We propose a technique to generate repair solutions automatically based on strategies used to rank erroneous axioms and a modified Reiter's Hitting Set algorithm (sections 3.3, 3.4). In addition, we consider strategies for rewriting axioms (section 3.5).
- We describe a preliminary implementation of an interactive ontology repair tool and discuss results of a conducted pilot study (section 3.6).

Note that while we focus on repairing unsatisfiable concepts in a consistent OWL Ontology, the underlying problem involves dealing with and rectifying a set of erroneous axioms, and thus the same principles for generating repair solutions are applicable when debugging an inconsistent OWL ontology.

2 Related Work

To our knowledge, the most relevant work is described in [13], where the authors identify minimal conflicting axiom sets responsible for unsatisfiable concepts in an $\mathcal{ALC}$ knowledge base, and then in [11], [12] use Reiter's Hitting Set algorithm to compute repair solutions from the conflict sets.

However, we differ from the work above in two key respects: our axiom-based solution works for a much more expressive description logic $\mathcal{SHOIN}$, and hence OWL-DL, with a finer granularity (identifying erroneous parts of axioms); and we consider ranking axioms for our repair solution and accordingly modify the Reiter's HS algorithm to generate repair plans.

3 Ontology Repair

3.1 Preliminaries

Before proceeding further, we revisit the notion of a MUPS (*Minimal Unsatisfiability Preserving Sub-TBoxes*), which was formally introduced in [13]. Roughly, a MUPS for an atomic concept A is a minimal fragment of the KB in which A is unsatisfiable. Obviously, a concept may have several different MUPS within an ontology. Finding *all* the MUPS of an unsatisfiable concept is a critical task from a debugging point of view, since one needs to remove at least one axiom from each set in its MUPS in order to make the concept satisfiable.

3.2 Continuing Where We Left Off

Improving Axiom-based Explanation A key component of our earlier debugging solution was the Axiom Pinpointing service which was used to extract the MUPS of a concept that is unsatisfiable w.r.t. a $\mathcal{SHOIN}$ knowledge base [4]. We have since extended this service to identify *specific parts* of axioms that are responsible for the inconsistency, and we refer to the resultant axiom set as the *precise* MUPS.

We briefly describe the idea behind this extension here (for details of our implementation, see [4]). Since we aim at identifying relevant parts of axioms, we define a function that splits the axioms in a KB K into "smaller" axioms to obtain an equivalent KB K_s that contains as many axioms as possible. This function rewrites the axioms in K in a convenient normal form and split across conjunctions in the normalized version, e.g., rewriting $A \sqsubseteq C \sqcap D$ as $A \sqsubseteq C, A \sqsubseteq D$. In some cases, we are forced to introduce new concept names, only for the purpose of splitting axioms into smaller sizes (which prevents any arbitrary introduction of new concepts); for example, since the axiom $A \sqsubseteq \exists R.(C \sqcap D)$ is not equivalent to $A \sqsubseteq \exists R.C$, $A \sqsubseteq \exists R.D$, we introduce a new concept name, say E, and transform the original axiom into the following set of "smaller" axioms: $A \sqsubseteq \exists R.E$, $E \sqsubseteq C$, $E \sqsubseteq D$, $C \sqcap D \sqsubseteq E$. Finally, the problem of finding the *precise* MUPS of an unsatisfiable concept in K reduces to the problem of finding its MUPS in the split version of the KB K_s. Note that to prevent an exponential blowup, we do not split the entire KB beforehand, instead perform a *lazy splitting* of certain specific axioms on the fly (as described in [4]).

Given the precise MUPS, any tool used to display it for the purpose of debugging the error can choose a suitable presentation format to highlight the relevant parts. For our debugging tool UI, we chose to strike out irrelevant parts of axioms that do not contribute to the contradiction (see Figure 1). So far, our tests have shown that this form of highlighting makes a significant difference in the presentation and greatly aids repair, since it makes it explicit to the user, the part of the axiom(s) that needs to be altered in order to resolve the bug.

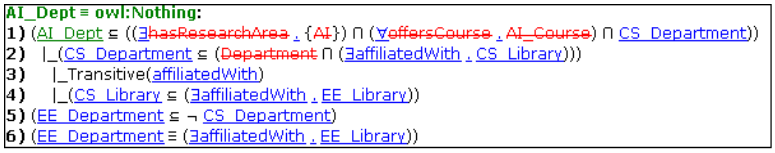

Fig. 1. Displaying the minimal set of axioms from the ontology (with key entities highlighted and irrelevant parts struck out) responsible for making the concept `AI_Dept` in the `University` ontology unsatisfiable.

3.3 Strategies for Ranking Axioms

We now discuss a key piece of the repair process: selecting which erroneous axiom(s) to remove from the MUPS in order to fix the unsatisfiable concepts.

For this purpose, an interesting factor to consider is whether the axioms in the MUPS can be *ranked* in order of importance. Repair is then reduced to an optimization problem whose primary goal is to get rid all of the inconsistency errors in the ontology, while ensuring that the highest rank axioms are preserved and the lowest rank axioms removed from the ontology.

A simple criterion to rank axioms is to count the number of times it appears in the MUPS of the various unsatisfiable concepts in an ontology. This idea is similar to the notion of *arity* of the axiom as discussed in [13]. If an axiom appears in n different MUPS (in each set of the MUPS), removing the axiom from the ontology ensures that n concepts turn satisfiable. Thus, higher the frequency, lower the rank assigned to the axiom.

Besides the axiom frequency in the MUPS, we consider the following strategies to rank ontology axioms:

- Impact on ontology when the axiom is removed or altered (need to identify *minimal impact* causing changes)
- Test cases specified manually by the user to rank axioms
- Provenance information about the axiom (author, source reliability, timestamp etc.)
- Relevance to the ontology in terms of its usage

Impact Analysis The basic notion of revising a knowledge base while preserving as much information as possible has been discussed extensively in belief revision literature [1]. We now apply the same principle to repairing unsatisfiable concepts in an OWL ontology, i.e., we determine the impact of the changes made to the ontology in order to get rid of unsatisfiable concepts, and identify minimal-impact causing changes. Since repairing an unsatisfiable concept involves removing axioms in its MUPS, we consider the impact of axiom removal on the OWL ontology.

A fundamental property of axiom removal based on the monotonicity of OWL-DL is the following: *removing an axiom from the ontology cannot add a new entailment.* Hence, we only need to consider entailments (subsumption,

instantiation etc.) that are lost upon axiom removal, and need not consider whether other concepts in the ontology turn unsatisfiable.

For now, we shall only consider subsumption/disjointness (between atomic concepts) and instantiation (of atomic concepts) as the only interesting entailments to check for when an axiom is removed. In the next subsection, we discuss how the user can provide a set of test cases as additional interesting entailments to check for.

As mentioned earlier, our Axiom Pinpointing service computes the minimal set of axioms (justification) responsible for any arbitrary entailment of an OWL-DL ontology . Thus, we can use this service to compute the justification sets for the significant subsumption and instantiation relationships in the ontology. When removing an axiom, we can check if it falls into a particular justification set, and accordingly determine which subsumption and/or instantiation relation(s) would break directly. Axioms to be removed can then be ranked based on the number of entailments they break (higher the rank, lesser the entailments broken).

An important distinction is the entailments resulting from the unsatisfiable concepts in the ontology. Note that when a concept is unsatisfiable, it is equivalent to the bottom concept (or in OWL lingo, `owl:Nothing`), and hence is trivially equivalent to all other unsatisfiable concepts, and is a subclass of all satisfiable concepts in the ontology. In this case, we need to differentiate between the stated or explicit entailments related to unsatisfiable concepts and the trivial ones. Thus, we apply the following strategy: if a given entailment related to an unsatisfiable concept holds in a fragment of the ontology in which the concept is satisfiable, we consider the entailment to be explicit.

There are two techniques to obtain such explicit entailments: the first is a brute-force approach that involves considering all possible (minimal) solutions to fix the unsatisfiable concept in the ontology, and verifying if the entailment still holds in the modified ontology. In order to obtain minimal repair solutions, we can use Reiter's algorithm as seen in the next section. On the other hand, the second approach is much faster (though incomplete) and is based on using the *structural analysis* techniques seen in [6] to detect the explicit relationships involving unsatisfiable concepts without performing large scale ontology changes. For example, we can use the *Ontology Approximation* heuristic to get rid of the contradictions in the ontology while revealing the hidden subsumption entailments.

Having obtained the explicit entailments related to unsatisfiable concepts, we can present them to the user to learn which, if any, of the relationships are (un)desired. This information would then be used in the plan generation phase.

We consider a few examples that highlight the significance of this strategy.

Example 1: In the Tambis OWL ontology[3], the three critical unsatisfiable concepts are: `metal, non-metal, metalloid`. The unsatisfiability arises because each concept is defined to be equivalent to the same complex concept:

[3] Note: All ontologies mentioned in this paper are available online at http://www.mindswap.org/ontologies/debugging/

chemical ⊓ (=1)atomic-number ⊓ ∃atomic-number.integer, and also defined to be *disjoint* from each other.

In this case, though the disjoint axioms appear in each of the three unsatisfiable concepts MUPS, removing them is not the correct solution, since eliminating the disjointness makes all three concepts metal, non-metal, metalloid equivalent which is probably undesired.

In fact, a better solution is to weaken the equivalence to a subclass relationship in each concept definition, thereby getting rid of the subclasses: chemical ⊓ (=1)atomic-number ⊓ ∃atomic-number.integer ⊑ metal / non-metal / metalloid; and we find that removing these relationships has no impact on other entailments in the ontology.

Example 2: Consider the following $MUPS$ of an unsatisfiable concept OceanCrustLayer w.r.t. the Sweet-JPL ontology $\mathcal{O}$: { (1) OceanCrustLayer ⊑ CrustLayer, (2) CrustLayer ⊑ Layer, (3) Layer ⊑ Geometric_3D_Object, (4) Geometric_3D_Object ⊑ ∃ hasDimension. {3D}, (5) OceanCrustLayer ⊑ OceanRegion, (6) OceanRegion ⊑ Region, (7) Region ⊑ Geometric_2D_Object, (8) Geometric_2D_Object ⊑ ∃ hasDimension. {2D}, (9) hasDimension is Functional }.

Note that in $\mathcal{O}$, each of the concepts CrustLayer, OceanRegion, Layer, Region, Geometric_3D_Object, Geometric_2D_Object, has numerous individual subclasses.

In this case, removing the functional property assertion on hasDimension from $\mathcal{O}$ eliminates the disjoint relation between concepts Geometric_2D_Object and Geometric_3D_Object, and between all its respective subclasses. Also, removing any of the following axioms $2, 3, 4, 6, 7, 8$ eliminates numerous subsumptions from the original ontology. Thus, using the minimal impact strategy, the only option for repair is removing either 1 or 5, which turns out to be the correct solution, based on the feedback given by the original ontology authors.

User Test Cases In addition to the standard entailments considered in the previous subsection, the user can specify a set of test cases describing desired entailments. Axioms to be removed can be directly ranked based on the desired entailments they break.

Also, in some cases, the user can specify *undesired* entailments to aid the repair process. For example, a common modeling mistake is when an atomic concept C inadvertently becomes equivalent to the top concept, owl:Thing. Now, any atomic concept disjoint from C becomes unsatisfiable. This phenomenon occurred in the CHEM-A ontology, where the following two axioms caused concept A (anonymized) to become equivalent to owl:Thing: $\{A \equiv \forall R.C$, $domain(R, A)$ }. Here, specifying the undesired entailment prevented our ontology-effect strategy from considering the impact of removal of the erroneous axiom (in this case, the equivalence, which needed to be changed to a subclass) on this entailment.

Provenance Information regarding Change Provenance information about an axiom can act as a useful pointer for determining its importance/rank, i.e., based on factors such as:

- reliability of the source (author, document etc.)
- context/reason for which the axiom was added (specified as an annotation or otherwise)
- time the axiom was specified

OWL has support for adding human-readable annotations to entities in an ontology using `owl:AnnotationProperties` such as *rdfs:label*, *rdfs:comment*. However, there is no direct provision to annotate assertions or axioms in the ontology, unless one resorts to reification. In general, manually providing provenance information about axioms can be a tedious task, and thus tool support is critical. To address this issue, ontology editors such as Protege [7], KAON [8] and Swoop have the option to maintain an elaborate change log to record provenance information.

In Swoop, we automatically keep track of all changes made to an OWL ontology, storing information such as authorship, date etc of each change. Additionally, we use a *change-ontology* that represents various atomic and complex change operations to serialize the change-log to RDF/XML, which can then be shared among users.

Such information is extremely useful for ranking axioms in a collaborative ontology building context, i.e., if a group of authors are collectively building an ontology, and there exists a precedence level among the authors, i.e., ontology changes made by the supervisor are given higher priority than those made by a subordinate. In this case, for each change made, one can derive the corresponding axioms added to the ontology, and automatically determine the rank of each axiom based on the person making the change.

Syntactic Relevance There has been research done in the area of ontology ranking [2], where for example, terms in ontologies are ranked based on their structural connectedness in the graph model of the ontology, or their popularity in other ontologies, and the total rank for the ontology is assigned in terms of the individual entity ranks. Since an ontology is a collection of axioms, we can, in theory, explore similar techniques to rank individual axioms. The main difference, of course, lies in the fact that ontologies as a whole can be seen as documents which link to (or import) other ontology documents, whereas the notion of linkage is less strong for individual axioms.

Here, we present a simple strategy that ranks an axiom based on the *usage* of elements in its signature, i.e., for each OWL entity (atomic class, property or individual) in the signature of the axiom, we determine how often the entity has been referenced in other axioms in the ontology, and sum the reference counts for all the entities in the axiom signature to obtain a measure of its syntactic (or structural) relevance.

The significance of this strategy is based on the following intuition: if the entities in the axiom are used (or are referred to) often in the remaining axioms

or assertions of the ontology, then the entities are in some sense, core or central to the overall theme of the ontology, and hence changing or removing axioms related to these entities may be undesired. For example, if a certain concept is heavily instantiated, or if a certain property is heavily used in the instance data, then altering the axiom definitions of that concept or property is a change that the user needs to be aware of. Similarly, in large ontologies where certain entities are accidentally underspecified or unused, axioms related to these entities may be given less importance.

The simple strategy presented above can be altered in various ways such as by restricting usage counts to certain axiom types, and/or weighing certain kinds of axioms differently than others (e.g., weighing property attribute assertions such as `InverseFunctional` higher). This would be motivated by user preferences depending on the ontology modeling philosophy and purpose (e.g., see OntoClean [3]).

3.4 Generating Repair Solutions

So far, we have devised a procedure to find tagged MUPS for an unsatisfiable concept in an OWL-DL ontology and proposed various strategies to rank axioms in the MUPS. The next step is to generate a repair plan (i.e., a set of ontology changes) to resolve the errors in a given set of unsatisfiable concepts, taking into account their respective MUPS and axiom ranks.

Modifying Reiter's Algorithm For this purpose, we use the Reiter's Hitting Set algorithm [10], which given a diagnosis problem and a collection of conflict sets for that problem, generates minimal hitting sets from the conflict sets. A hitting set for a collection of sets C is a set that touches (or intersects) each set in C. A hitting set is minimal for C, if no proper subset of it is a hitting set for C. This approach was suggested in [12], which generates hitting sets from the MUPS – the idea here is that removing all the axioms in the minimal hitting set removes one axiom from each of the MUPS and thus renders all concepts satisfiable. The same principle applies to our repair solution except that we need to modify the HS algorithm to take into account the axiom ranks.

Given a collection C of conflict sets, Reiter's algorithm introduces the notion of a hitting set tree (HST), which is the smallest edge-labeled and node-labeled tree such that a node n in HST is labeled by a tickmark if C is empty, otherwise its labeled with any set $s \in C$. For each node n, let $H(n)$ be the set of edge labels on the path in HST from the root to n; then the label for n is any set $s \in C$, that satisfies the property $s \cap H(n) \leftarrow \emptyset$, if such a set exists. If n is labeled by a set s, then for each $\sigma \in s$, n has a successor n_σ joined to n by an edge labeled by σ. For any node labeled by a tickmark, the labels of its path from the root ($H(n)$) is a hitting set for C. Also, while generating the HST, if the search along a path exceeds the current optimal solution, the search is terminated earlier, marked by a cross in the label of a node.

Now for our problem, the MUPS of the unsatisfiable concepts correspond to the conflict sets. However, while the normal HST algorithm has the optimality

criteria as the minimal path length, we set it as the minimal *path rank* instead, i.e., the sum of the ranks of the axioms in the path $H(n)$ should be minimal. Also, in the standard algorithm, there is no basis for selecting an axiom over another while building the edges of the HST, whereas we can use the ranks of the axioms when making a selection to prune down the search space, i.e., at each stage, we select the lowest ranked axiom while creating a new edge.

Figure 2 shows a HST for a collection $C = \{\{2,5\},\{3,4,7\},\{1,6\},\{4,5,7\},\{1,2,3\}\}$ with the axioms $1-7$ ranked as follows: $r(1) = 0.1$, $r(2) = 0.2$, $r(3) = 0.3$, $r(4) = 0.4$, $r(5) = 0.3$, $r(6) = 0.3$, $r(7) = 0.5$, where $r(x)$ is the rank of axiom x. The ranks are computed based on the factors mentioned earlier, such as arity, impact analysis etc. each weighed separately if needed using appropriate weight constants. The superscript for each axiom-number denotes the rank of the axiom, and P_r is the path rank computed as the sum of the ranks of axioms in the path from the root to the node. For example, for the leftmost path shown: $P_r = 0.2 + 0.3 + 0.1 + 0.3 = 0.9$.

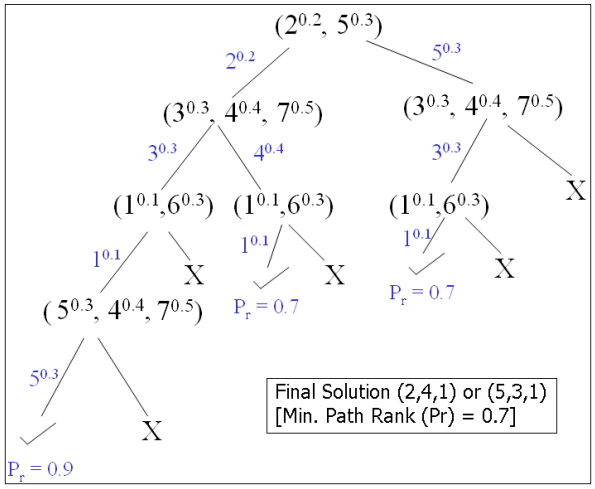

Fig. 2. Modified Reiter's Hitting Set Algorithm: Generating a repair plan based on ranks of axioms in the MUPS of unsatisfiable concepts.

As shown in the figure, by choosing the lowest rank axiom in each set while constructing the edges of the HST, the algorithm only generates 3 hitting sets, two of which are minimal, while avoiding numerous path checks (indicated by the crosses). The repair solution found with the minimal path rank is either {2,4,1} or {5,3,1}.

However, there is a drawback of using the above procedure to generate repair plans, i.e., impact analysis is only done at a single axiom level, whereas the cumulative impact of the axioms in the repair solution is not considered. This can lead to non-optimal solutions. For example, in the Tambis ontology, where

the three root classes are asserted to be mutually disjoint, removing any one of the disjoint axioms does not cause as large an impact as removing all the disjoints together.

In order to resolve this issue, we propose another modification to the algorithm above: each time a hitting-set HS is found, we compute a new path-rank for HS based on the cumulative impact of the axioms in the hitting-set. The algorithm now finds repair plans that minimize these new path-ranks. Note that the early termination condition for paths remains the same since the path rank represents a lower bound, as cumulative impact is always greater than or equal to the sum of individual unique impacts.

Improving and Customizing Repair The algorithm described above can be used in general to fix any arbitrary set of unsatisfiable concepts, once the MUPS of the concepts and the ranks for axioms in the MUPS is known. Thus, a brute force solution for resolving *all* the errors in an ontology involves determining the MUPS (and ranking axioms in the MUPS) for *each* of the unsatisfiable concepts. This is computationally expensive and moreover, unnecessary, given that strong dependencies between unsatisfiable concepts may exist. Thus, we need to focus on the MUPS of the critical or root contradictions in the ontology.

To achieve this, we make use of a debugging service we have devised in [6] that identifies the *root* unsatisfiable concepts in an ontology, which propagate and cause errors elsewhere in the ontology, leading to *derived* unsatisfiable concepts. Intuitively, a root unsatisfiable concept is one in which a clash or contradiction found in the concept definition does not *depend on the unsatisfiability* of another concept in the ontology; whereas, a derived unsatisfiable concept acquires a contradiction due to its dependence on another unsatisfiable concept. For example, if A is an unsatisfiable concept, then a concept B ($B \sqsubseteq A$) or C ($C \sqsubseteq \exists R.A$) also becomes unsatisfiable due to its dependence on A, and is thus considered as derived.

We have experimented with the root/derived debugging service on numerous OWL ontologies that have a large number of unsatisfiable concepts and found it to be useful in narrowing down the error space quickly, e.g, for the Tambis OWL Ontology, only 3 out of 144 unsatisfiable concepts were discovered as roots in under 5 seconds. From a repair point of view, the key advantage here is that one needs to focus on the MUPS of the root unsatisfiable concepts alone since fixing the roots effectively fixes a large set of directly derived concept bugs.

Also, the service guides the repair process which can be carried out by the user at three different granularity levels:

– *Level 1: Reparing a single unsatisfiable concept at a time*: In this case, it makes sense to deal with the root unsatisfiable concepts first, before resolving errors in any of the derived concepts. This technique allows the user to monitor the entire debugging process closely, exploring different repair alternatives for each concept before fully fixing the ontology. However, since at every step in the repair process, the user is working in a localized context (looking at a single concept only), the debugging of the entire ontology

could be prolonged due to new bugs introduced later based on changes made earlier. Thus, the repair process may not be optimal.
- *Level 2: Repairing all root unsatisfiable concepts together*: The user could batch repair all the root unsatisfiable concepts in a single debugging iteration before proceeding to uncover a new set of root/derived unsatisfiable concepts. This technique provides a cross between the tool-automation (done in level 3) and finer manual inspection (allowed in level 1) with respect to bug correction.
- *Level 3: Repairing all unsatisfiable concepts*: The user could directly focus on removing all the unsatisfiable concepts in the ontology in one go. This technique imposes an overhead on the debugging tool which needs to present a plan that accounts for the removal of all the bugs in an optimal manner. The strategy works in a global context, considering bugs and bug-dependencies in the ontology as a whole, and thus may take time for the tool to compute, especially if there are a large number of unsatisfiable concepts in the ontology (e.g. Tambis). However, the repair process is likely to be more efficient compared to level 1 repair.

The number of steps in the repair process depends on the granularity level chosen by the user: for example, using Level 1 above, the no. of steps is atleast the no. of unsatisfiable concepts the user begins with; whereas using Level 3 granularity, the repair reduces to a single big step. To make the process more flexible, the user should be allowed to change the granularity level, as and when desired, during a particular repair session.

3.5 Suggesting Axiom Rewrites

Now, to make our repair solution more flexible, we consider strategies to rewrite erroneous axioms instead of strictly removing them from the ontology[4].

Using Erroneous Axiom Parts As shown in section 3.2 (see Figure 1), our Axiom Pinpointing service has been extended to identify parts of axioms in the MUPS responsible for making a concept unsatisfiable. Having determined the erroneous part(s) of axioms, we can suggest a suitable rewrite of the axiom that preserves as much as information as possible while eliminating unsatisfiability.

Identifying Common Pitfalls Common pitfalls in OWL ontology modeling have been enumerated in literature [9]. We have summarized some commonly occurring errors that we have observed (in addition to those mentioned in [9]), highlighting the *meant* axiom and the reason for the mistake in each case.

[4] Note that rewriting an axiom involves an axiom removal followed by an addition. Thus, similar to the impact analysis performed for axiom removal, we also need to consider entailments that are introduced when an axiom is added. Currently, we only check if unsatisfiable concepts arise upon axiom addition, and we are working on iterative reasoning techniques (see http://www.mindswap.org/papers/TR-incclass.pdf) to optimally compute other entailments added.

Asserted	Meant	Reason for Misunderstanding
$A \equiv C$	$A \sqsubseteq C$	Difference between Defined and Primitive concepts
$A \sqsubseteq C$ $A \sqsubseteq D$	$A \sqsubseteq C \sqcup D$	Multiple subclass has intersection semantics
`domain(P,A)` `range(P,B)`	$A \sqsubseteq \forall P.B$	Global vs. Local property restrictions
`domain(P,A)` `domain(P,B)`	domain(P, $A \sqcup B$)	Unclear about multiple domain semantics

A library of error patterns can be easily maintained, extended and shared between ontology authors using appropriate tool support. Once we have identified the axioms in the ontology responsible for an unsatisfiable concept, we can check if any of the axioms has a pattern corresponding to one in the library, and if so, suggest the *meant* axiom to the user as a replacement. We note that in a lot of cases that we have observed, the most common reason for unsatisfiability is the accidental use of equivalence instead of subsumption.

In some cases, an additional heuristic to consider is the label (or ID) of the concept or role, which acts as a pointer to its intended meaning and can be used to detect mismatches in modeling. For example, the unsatisfiable concept `OceanCrustLayer` seen earlier in the Sweet-JPL OWL ontology was accidentally defined to be a subclass of `CrustRegion`, instead of `CrustLayer`.

A combination of the heuristics was used to debug an error in the University ontology. The concept `ProfessorInHCIorAI` was responsible for the unsatisfiable concepts `AI_Student` and `HCI_Student` because there were two separate subclass axioms for `ProfessorInHCIorAI`, associating it with the student concepts separately, whereas the 'or' in the concept name implied that a disjunction was intended.

3.6 Interactive Repair Tool (Preliminary Evaluation)

We are currently working on an ontology repair plug-in for Swoop. The key design goal is to provide a flexible, interactive framework for repairing unsatisfiable concepts in an ontology by allowing the user to analyze erroneous axioms, weigh axiom ranks as desired, explore different repair solutions by generating plans on the fly, preview change effects before executing the plan and compare different repair alternatives. Moreover, the tool also suggests axiom edits where possible.

Figure 3 is a screenshot of the Swoop repair plugin when used to debug the University OWL ontology. As can be seen, the top segment of the repair frame displays a list of unsatisfiable concepts in the ontology, with the *root* classes marked. The adjacent pane renders the axioms responsible for making the concepts selected in the list unsatisfiable. There are two view modes for this pane – the one shown in Figure 3 displays the erroneous axioms for each unsatisfiable class in separate tables with axioms indented (as described in [6]), and common axioms responsible for causing multiple errors highlighted as shown.

The other view (not shown) displays all erroneous axioms globally in a single list.

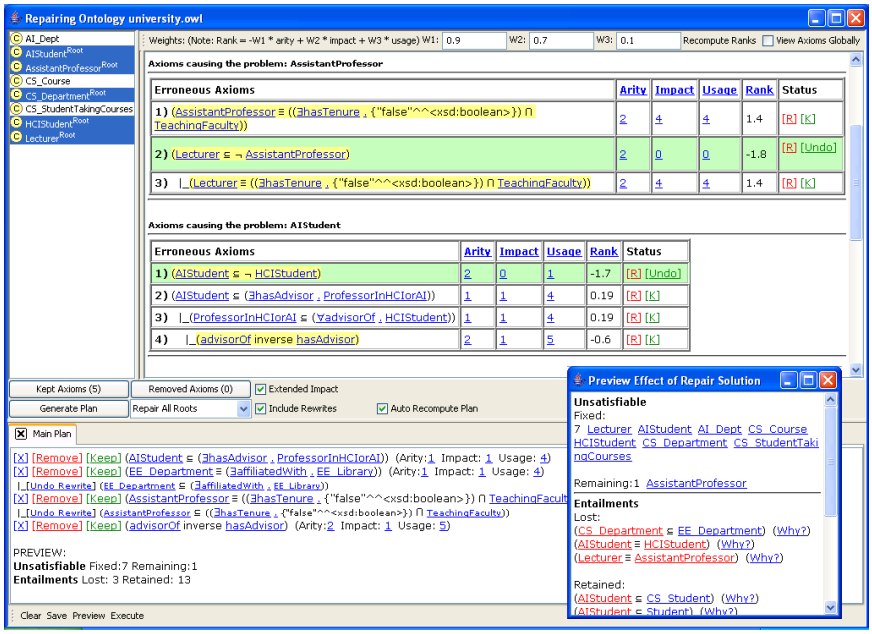

Fig. 3. Interactive Repair in Swoop: Generating a repair plan to remove all root unsatisfiable concepts in the University OWL Ontology. The popup in the lower right corner displays a *preview* of the current repair plan including unsatisfiable concepts that would get fixed and key entailments that would be lost or retained.

The tables display for each axiom, its arity, impact and usage, computed as described earlier. The values for these parameters are hyperlinked, clicking on which pops up a pane which displays more details about the parameter (not shown in the figure). Also, clicking on the table headers re-sorts the results based on the parameter selected. The total rank for each axiom, displayed in the last column of the table, is the weighted sum of the parameter values, with the weights (and thus ranks) being easily reconfigurable by the user. For example, users interested in generating minimal impact plans can assign a higher weight to the impact parameter, while users interested in smaller sized plans can weigh arity higher. The range of the weights is from -1.0 to 1.0.

As discussed earlier, we provide three different granularities for the repair process, i.e., the ability to fix a particular set of unsatisfiable concepts; all the *roots* only; or all the unsatisfiable classes directly in one go. For example, in Figure 3, the user has asked the tool to generate a plan to repair all the roots.

For a repair tool to be effective, it should support the easy customization of the plan to suit the user's needs. In the simple case, the user can either choose

to *keep* a particular axiom in the ontology, or forcibly *remove* a particular one. These user-enforced changes are automatically reflected in the plans. In Figure 3, the user has chosen to *keep* the disjoint axioms AIStudent $\sqsubseteq \neg$HCIStudent, and Lecturer $\sqsubseteq \neg$AssistantProfessor in the ontology (highlighted in green in the Table). In the advanced case, the user can choose to keep or remove a particular entailment of the ontology, e.g., a particular subclass relation. The tool then takes these desired and undesired entailments into account when generating a plan.

Finally, axiom rewrites suggested by the tool can be (optionally) included in the plan as well. In the figure, the tool has suggested weakening the two equivalence axioms to subclass relations, which removes the contradictions in the unsatisfiable classes, but preserves the semantics as much as possible. Obviously, the user can directly edit erroneous axioms if desired.

The repair plan can be saved, compared with other plans and executed, after which the ontology changes (which are part of the plan) are logged in Swoop. These changes can be serialized and shared among ontology users (as shown in [5]).

Pilot Study We conducted a small pilot study involving twelve subjects, who had at least one year's experience with OWL and an understanding of description-logic reasoning that varied greatly (novices to experts). Each subject received a 30 minute orientation including 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; and a detailed walkthrough of the debugging and repair support in Swoop using a set of toy ontologies.

We selected two OWL Ontologies – University.owl and miniTambis.owl and asked each subject to fix all the unsatisfiable classes in a particular ontology using the debugging techniques seen in [6] (case 1), and in the other ontology using the repair techniques described in this paper (case 2). The subjects were randomly assigned to the two cases, but the overall distribution was equally proportional in that given a particular ontology, an equal number of subjects (six) debugged it with and without using the repair facilities. At the end of the study, our goal was to compare the performance improvement, if any, of using the repair services over the previous debugging services, which were shown to be useful in an earlier study [6].

The results of the study were encouraging. We found that while the quality of the repair solutions in both cases were comparable, the time taken to arrive at a solution in the second case was between 2-8 times less than in first case. More importantly, the subjects felt that in the second case, they understood the different alternatives for repair, and decided on one knowing its overall impact. Three key features appreciated by the subjects were the impact analysis to see lost/retained entailments, the suggested axiom rewrites and the option to modify the plan on the fly by *keeping* or forcibly *removing* axioms.

4 Conclusion

In this paper, we have discussed the problem of repairing unsatisfiable concepts in OWL Ontologies, and provided solutions that tie in nicely with (and extend) our earlier work on explanation and debugging [6]. Thus, we are now in a position to construct an end-to-end framework for interactive debugging and repair of OWL Ontologies, though more extensive testing and evaluation is necessary. Given the nature of the problem, our focus right from the start, has been on the user-experience and in aiding the overall understanding and analysis of the ontology, and the results so far have been in correspondence with our goal.

References

1. Gardenfors P. Makinson D. Alchourron, C. On the logic of theory change: Partial meet contraction and revision functions. 1985. Journal of Symbolic Logic 50 (1985).
2. Li Ding, Rong Pan, Tim Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and Ranking Knowledge on the Semantic Web. In *Proceedings of the 4th International Semantic Web Conference*, LNCS 3729, pages 156–170. Springer, November 2005.
3. Nicola Guarino and Christopher Welty. Evaluating ontological decisions with ontoclean. *Commun. ACM*, 45(2):61–65, 2002.
4. A. Kalyanpur, B. Parsia, B. Cuenca-Grau, and E. Sirin. Axiom pinpointing: Finding (precise) justifications for arbitrary entailments in $\mathcal{SHOIN}$ (owl-dl). Technical report, UMIACS, 2005-66, 2006. Technical Report.
5. A. Kalyanpur, B. Parsia, E.Sirin, B. Cuenca-Grau, and J. Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 2005. To Appear.
6. Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics, Volume 3 Issue 4*, 2005. (To Appear).
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. Daniel Oberle, Raphael Volz, Boris Motik, and Steffen Staab. An extensible ontology software environment. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, chapter III, pages 311–333. Springer, 2004.
9. Alan Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns . In *EKAW*, pages 63–81, 2004.
10. R. Reiter. A theory of diagnosis from first principles. 1987. Artificial Intelligence 32:57-95.
11. S. Schlobach. Debugging and semantic clarification by pinpointing. 2005. European Semantic Web Conference ESWC.
12. S. Schlobach. Diagnosing terminologies. In *In Proceedings of AAAI'05*, 2005.
13. S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of IJCAI*, 2003.
14. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. Technical report, University of Maryland Institute for Advanced Computes Studies (UMIACS), 2005-68, 2005. Available online at http://www.mindswap.org/papers/PelletDemo.pdf.