ReTAX+: A Cooperative Taxonomy Revision Tool

Sik Chun (Joey) Lam, Derek Sleeman, and Wamberto Vasconcelos
Department of Computing Science
The University of Aberdeen, Aberdeen, AB24 3UE, UK
{slam, sleeman, wvasconc}@csd.abdn.ac.uk

Abstract

ReTAX is a system that assists domain experts to accommodate a new item in an established taxonomy; it suggests a number of ways in which that can be achieved namely; modifying the new entity, the taxonomy, or both. Further, a set of refinement operators are used to guarantee the consistency of the resulting taxonomy. ReTAX+ is a system which provides the user with additional functionalities as they build a new taxonomy or reuse an existing taxonomy. Specifically, it provides functions to enable a user to add, edit or delete the values associated with class attributes; additionally, it provides functions to add a class, delete a class, merge two classes, and split a class. Again consistent with the philosophy of ReTAX, ReTAX+ offers the user, when relevant, a number of options to achieve his objectives. For example, deleting a class is a fairly radical step and various decisions need to be made about the resulting "orphaned" instances and sub-classes.

1. Introduction

Our ultimate goal is to build a sophisticated Ontology Development and Maintenance environment. However, we made a conscious decision to first build a sophisticated taxonomy development tool; having achieved that objective we are now analysing the various enhancements which will be needed to ReTAX+'s functions to provide a comparable environment for handling ontologies. The basic approach of the original ReTAX system [1] is that it accepts as inputs a pre-established taxonomy and a new item to be classified. The system has a clear set of rules to determine whether a taxonomy is well formed, and an associated set of operators to achieve consistency. In fact, a new item can be accommodated onto a taxonomy by modifying the new entity, changes to the taxonomy or a combination of both. Whilst still retaining those features the enhanced system ReTAX+ now also provides an taxonomy maintenance environment in which attributes can be added, deleted and edited; similarly, a class can be added, deleted, merged with another class, or a class can be split. Again ReTAX+ is a cooperative system which suggests, at various stages, options; it is the user who makes the selections.

The paper is organised as follows: Section 2 presents an overview of the related work. Section 3 describes the system architecture. Section 4 provides some preliminary definitions on ontological analysis. In Section 5, the operations on attributes and classes and refinement strategies are presented. Finally, the implementation of system, the next steps, and a conclusion are given.

2. Related Work

Many tools for building ontologies have been developed in the last few years[1]. Four popular ontology editors are compared in this section: OntoEdit, WebODE, Protégé and OilEd, each of which covers a wide range of ontology development processes. We mainly focus on their developed evaluation services and the support for (semi-) automatic ontology improvement.

OntoEdit[2] employs Ontobroker[3] as its inference engine. It is used to process axioms in the refinement and evaluation phases [5]. However, it lacks transparency during refinement and evaluation, as the user does not obtain explanations why a particular change is necessary; the user only obtains information about the numbers of resulting changes, but not the details [6]. WebODE[4] uses the ODEClean [7] approach to evaluate the concept taxonomy by exposing inappropriate and inconsistent modeling choices according to the meta-properties. The errors in the taxonomy appear in a separate window and are high-lighted in the graph. However, there is no functionality provided for users to correct the reported errors; nor can the system make the changes easily. Protégé[5] has a powerful plug-in PROMPT[6]. This tool guides users through the merging process, presenting them with suggestions for which operations should be performed, and then performs certain operations automatically [8]. However, its focus lies on ontology merging and alignment, rather than ontology evaluation. OilEd[7] uses the reasoning support ontology system - FaCT, which checks class consistency and infers subsumption relationships [9]. Only these sources of errors are indicated but no explanation or suggestion for users to resolve inconsistencies are given.

All four tools have inference mechanisms, but only offer partial support for (semi-) automatic ontology refinement, even though they make the ontology easier to understand and modify [5]. Also there is no information provided for users to analyse the reasons for conflicts which arise. This means the feedback cannot directly help users revise the ontologies effectively.

---

[1] http://www.ontoweb.org/download/deliverables/D13_v1-0.zip
[2] http://www.ontoprise.de/home
[3] Ontobroker is an the inference engine produced by ONTOPRISE
[4] http://delicias.dia.fi.upm.es/webODE/
[5] http://protege.stanford.edu/
[6] http://protege.stanford.edu/plugins/prompt/prompt.html
[7] http://oiled.man.ac.uk/

It is generally agreed that the developers of ontologies need a great deal of support as this is seen as a complex process. Generally, ontology management tools provide facilities to add, edit and delete classes, subclasses and instances. We did a comparative review of the facilities provided by these four systems when the user attempts to delete a class, its subclasses and instances. OntoEdit pops up a window which indicates how many classes, relations, instances and values of instances will be deleted. WebODE also pops up a window which indicates the classes, attributes, relations or other references which will be deleted, and the subclasses which will be re-allocated to the root of the ontology. In Protégé, a class with instances cannot be deleted; a class without instances but with subclasses can be deleted (the subclasses are deleted as well). In OilEd, a class which is referenced by other classes or attributes or relations cannot be deleted.

However, in ReTAX+, a class with subclasses can be deleted, and the user is prompted, by a list of options, to choose how to handle the orphaned subclasses, attributes and instances (explained in Section 5.3.2). We concluded that proactive support for class and relationship identification is necessary during the refinement phase, and the editor should allow the user control of these processes [6].

3. System Architecture

ReTAX+ is designed as a 3-tier architecture: front end (user interface), middle (inconsistency detection and refinement strategies), and back end (output results).

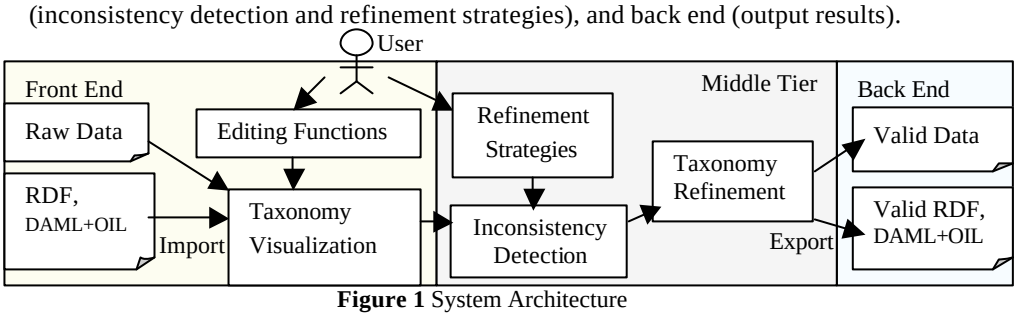

Figure 1 System Architecture

The ReTAX+ front end reads data from a text file, which contains the definitions of classes and attributes, and attribute values. An RDF file can also be imported to the system. The back end processes the refined taxonomy which can be stored in a text file or exported to an RDF file.

The middle tier is responsible for detecting inconsistencies in the taxonomy using consistency rules (explained in Section 5.3), and then implementing the refinements. Any invalid classes are displayed in different colours. In order to assist the user to modify the inconsistent structure, we implemented a process with the following characteristics:

- It explains and indicates the source and reasons for inconsistencies;
- The user is given a set of possible refinement strategies. For example, in the case of deleting a class, the user is given options for how to re-classify the orphaned subclasses, instances and attributes.

4. Definitions for Ontological Analysis

Previous efforts at organising taxonomies have focused on the semantics of the taxonomic relationship [10]. Our approach, however, concentrates on the properties (i.e. attributes) involved in the taxonomic relationship, rather than the semantics of the relationship itself. Therefore, only the properties of the class determine its consistency in the whole hierarchy structure; while their structural relationship (e.g. partition, generalization) and semantic meanings are not taken into account.

In this paper, we present and formalize the attributes and the taxonomy structure using set theory. We then demonstrate how they can be used for conceptual modelling. The following section gives formal definitions.

4.1 Taxonomy

A taxonomy is a finite set of frames, called *classes*, together with a finite set of arcs, called *relationships*, that join some or all of these classes. In a tree, classes lower down in the hierarchy can be seen as specialisations of the classes higher up in the hierarchy. The hierarchy reflects class-subclass relationship. There is a numeric threshold value t, which is a cut-off between dominant and subsidiary attributes. Its value is changeable by the user, however its default value is 0.5.

Definition 1 : A taxonomy T is a triple $T = \langle \boldsymbol{C}, t, ? \rangle$ where

- $\boldsymbol{C} = \{C_1, C_2, \ldots, C_n\}$ is a finite set of classes C_i, $1 = i = n$
- $t \in$ Â, is a real number representing a threshold value, $1 = t = 0$
- ?: $\boldsymbol{C} \times \boldsymbol{C}$ formally represents the class-subclass relationships among the classes of the taxonomy, such that if $(C, C¢) \in$? then $C¢$ is a subclass of C

4.2 Class & Attributes

All classes have the same attributes in a taxonomy, the values of attributes are what distinguish one class from other classes. The following three points explain why we decided all classes should have the same attributes:

1. Speed up design by reusing attributes
   When a class or sub-class is created, the user only needs to provide the attribute values, which are inherited from the root or parent, rather than creating new attributes to describe the class. The inheritance of attributes reduces the design effort.
2. Minimize the accidental design and structure errors
   This avoids the user introducing any duplicated attributes (e.g. an attribute which already exists in other classes) and the same semantical attributes (e.g. attributes which are essentially synonyms).
3. Extend the description of taxonomy to be more complete
   When an attribute is appended to a class, it is propagated to the whole taxonomy; the descriptions of other classes are extended with the extra attribute.

In order to increase the differentiation of an individual class from the whole hierarchy, each attribute has a discriminatory relevance index (which we abbreviate as *dri*) indicating the taxonomic significance or discriminatory power of an attribute. The attribute values could be either a set of integers, string, or classes, this attribute

is called the “ordered-set”; or a range of integers, called “integer-range”, which has the lowest and the highest limit of the range.

Definition 2: A class C is a pair $C = \langle c, \boldsymbol{A} \rangle$ where

- c is the name of the class
- $\boldsymbol{A}$ is the set of attributes of the class.

Definition 3: The attributes $\boldsymbol{A}$ of a class are represented as a set of triples $\langle a, \boldsymbol{V}, d \rangle$ where

- a is the attribute name
- $d \in$ Â, is a real number representing its *dri* value, $1 = d = 0$
- $\boldsymbol{V}$ is either a set of attribute values $\{v_1, v_2, \ldots, v_n\}$ or an integer pair $\langle r_L, r_H \rangle$

$$\boldsymbol{V} = \begin{cases} \langle r_L, r_H \rangle, \text{ then } \boldsymbol{A} \text{ is an integer-range type attribute,} \\ \qquad \text{where } r_L \text{ and } r_H \text{ is the lowest and highest limit of the range} \\ \{v_1, v_2, \ldots, v_n\}, \text{ then } \boldsymbol{A} \text{ is an ordered-set type attribute,} \\ \qquad \text{where } v \text{ can be an integer, string or class} \end{cases}$$

Definition 4: The subsidiary attributes of a class are the subset of attributes whose *dri* values are less than the given threshold value t, that is,

$$\text{sub}(\langle c, \boldsymbol{A} \rangle) \subseteq \boldsymbol{A},$$
$$\text{sub}(\langle c, \boldsymbol{A} \rangle) = \{ \langle a, \boldsymbol{V}, d \rangle \mid \langle a, \boldsymbol{V}, d \rangle \in A, d < t \}$$

Definition 5: The dominant attributes of a class are the subset of attributes whose *dri* values are greater than or equal to the given threshold value t, that is,

$$\text{dom}(\langle c, \boldsymbol{A} \rangle) \subseteq \boldsymbol{A},$$
$$\text{dom}(\langle c, \boldsymbol{A} \rangle) = \{ \langle a, \boldsymbol{V}, d \rangle \mid \langle a, \boldsymbol{V}, d \rangle \in A, d \geq t \}$$

5. Operations on Taxonomy

Taxonomy development is necessarily an interactive process [6]. There are a number of factors which require taxonomy refinement [6]:

1. The original taxonomy often includes errors.
2. The accepted domain knowledge or the users requirements change and hence it is necessary to revise the corresponding taxonomy.

Therefore, the system has to cope with taxonomy modifications, and has to ensure consistency of the taxonomy whenever it is changed. Our approach is to implement semi-automatic taxonomy refinements, that is the system requires the user to choose an appropriate refinement strategy from a number of options and the system then implements the chosen changes. The current set of refinements is:

- Maintenance of the τ and *dri* values
- Adding, deleting and editing attributes
- Adding, deleting, splitting and merging classes

5.1 Maintenance of t and *dri*

If a taxonomy keeps expanding, either its width or depth increases, it may be necessary to change the τ value. The adjustment of τ has an important effect on the taxonomy, the τ arbitrary value could result in an excessive number of dominant attributes or none. To avoid these adverse effects, the system presents change

information in an orderly way, and describes any potential problems to the user when he adjusts the τ value [6]. After comprehending the information, the user commits or cancels the change. Once the change is activated, the system generates a list of suggestions and propagates changes to alleviate the side effects.

We realise that it might be difficult for the user to assign numeric values to attributes, to differentiate them into dominant and subsidiary features. Furthermore, during the design of a taxonomy such values can experience constant changes with repercussions throughout the taxonomy. We are currently investigating means to provide automatic support for this task, whereby the user would simply alert ReTAX+ about the status of an attribute, i.e., if it should be a dominant or a subsidiary feature. The system should then automatically assign a *dri* value to the attribute which would make that attribute subsidiary or, indeed, dominant, reflecting the user's preference. When a class is created ReTAX+ assign a default value of zero to the *dri* of all attributes, making them all subsidiary.

5.2 Manipulation of Attributes

Whenever an attribute of a class is modified, the change needs to be propagated to its superclasses and subclasses. The following sections describe these mechanisms.

5.2.1 Editing an attribute

The attribute values of an ordered-set type attribute can be deleted or added to; that of an integer-range type attribute can be modified.

Algorithm 1a: Ordered-set type attribute $\langle a, \{v_1, v_2, \ldots, v_n\}, d \rangle$
When a new attribute value v is added to an ordered set type attribute of a class, this value is added to its super-classes and sub-classes as well. Its sibling classes are unaffected with the addition. When an attribute value v is deleted from the ordered set type attribute, this value is deleted from its sub-classes, but its super-classes are not affected.
Algorithm 1b: Integer-range type attribute $\langle a, \langle r_L, r_H \rangle, d \rangle$
The range of an integer range type attribute of a class is constrained by that of its parent. When the integer range of the attribute is narrowed, the range of its sub-class is narrowed as well; if the range of the class is widened, then the ranges of the superclasses need to be widened but the ranges of the subclasses do not need to be changed (see Figure 2).

5.2.2 Adding an attribute

This extends the description of a taxonomy by adding a new attribute. Only a dominant attribute can be introduced to a class, so that it can be used to distinguish among the siblings.

When a new attribute is added to a class, the user can either assign a *dri* value, or indicate the current dominant attribute; in this case the system sets the *dri* value of the new attribute to be slightly higher than the *dri* value for the current dominant attribute. The new attribute is propagated to the whole taxonomy. The *dri* value in both its superclasses and subclasses decreases, as its importance is gradually

decreased. The *dri* value for those classes which are not its superclasses, subclasses and siblings are zero by default.

Algorithm 2: The new attribute, $\boldsymbol{A}_{new}$, its attribute values and *dri* are given by the user, and are added into a set of sibling classes $\boldsymbol{C_s}$ respectively. The subclasses of $\boldsymbol{C_s}$ inherit the same attribute values as $\boldsymbol{C_s}$; the superclasses of $\boldsymbol{C_s}$ have the integrated attribute values of $\boldsymbol{C_s}$. The *dri* value both in the subclasses and superclasses are less than the threshold value t. The attribute value is null in other classes; their *dri* value is zero by default.

5.2.3 Deleting an attribute

This removes an attribute $\langle a, \boldsymbol{V}, d \rangle$ from a taxonomy T.

Algorithm 3: Only an attribute which is subsidiary in every class in a taxonomy can be deleted. If the user wants to delete a dominant attribute of a class, then he has firstly to change the *dri* value of the attribute so that it is designated as a subsidiary attribute. Indeed this has to be done for all the nodes in the taxonomy, before an attribute can be deleted.

5.3 Manipulation of Classes

In this section, we describe the "Add a new class", "Delete a class", "Merge two classes" and "Split a class" functions, they are performed to improve the structure according to the user requirements. The user is informed which classes, instances and attributes are affected after these operations are initiated.

5.3.1 Adding a new class

One of the functions of ReTAX+ is to assist the user in accommodating a new class in an established taxonomy. Following the addition of the new class, the taxonomy may no longer be valid, in which case the system has to refine the taxonomy and possibly the class to ensure the new class is appropriately located.

Algorithm 4: When a class is created, ReTAX+ assigns a default value of zero to the *dri* of all attributes, making them all subsidiary. The user is required to fill the attribute values which are constrained by the attribute values of the root. He indicates the exact values of *dri* if he knows the importance of each attribute. Perhaps, he can just indicate the dominant attribute whose *dri* values will then be assigned by the system automatically. The new class must have at least one dominant attribute, otherwise, it has no characteristics and cannot be added in the taxonomy. Hence, the system searches for the most likely parent and adds the new class as its child. Otherwise, find the class whose dominant attribute is the same as the new class's dominate attribute, and add the class as its siblings. If no sibling is found, add the class as a child of the root. As a final step the taxonomy needs to be re-validated.

5.3.2 Deleting a class

When a class in the hierarchy is deleted, there are several concerns to be resolved. One is what to do with the orphaned sub-classes; and what to do with the class's

instances; thirdly, what to do when other classes have cited the to-be-deleted class as one of their attribute values. For each concern, we provide a set of possible strategies for users to choose based on their own preferences. For the orphaned sub-classes and those attributes whose value is the deleted class, see Algorithm 5. For the class's instances, they may either be deleted or reconnected to its super-class [4].

Algorithm 5: A class is deleted from a taxonomy, its orphaned sub-classes can either be deleted or reconnected to its super-class or the root. Suppose there is an attribute in a class the values of which are references to classes within the taxonomy. If the user deletes a class which appears as one of the values of this attribute, then the values of the attribute must be either deleted or altered: the reference to the deleted class is changed to a reference to the parent class (see Figure 3).

```
Edit_Attribute ( T, v, ⟨a, V, d⟩ , C, T¢ ) {    // v: a new attribute value input by the user
   switch(attribute-type)
   case (ordered-set-attribute):
      action = user's action either add or delete
      switch(action)
            case(add):
               C = ⟨c, A¢⟩,  A¢ = A ∪ {⟨a, V∪v , d⟩ } – {⟨a, V, d⟩ }
               for each Ck ∈  C do
                 if ((Ck, C) ∈ ? or (C, Ck) ∈ ?)   // Ck  is a superclass or subclass
                        Ck = ⟨c, A¢⟩,  A¢ = A ∪ {⟨a, V∪v , d⟩ } – {⟨a, V, d⟩}
            case(delete):
               v = the attribute value will be deleted by the user
               C = ⟨c, A¢⟩,  A¢ = A ∪ {⟨a, V – v , d⟩} – {⟨a, V, d⟩}
               for each Ck ∈  C do
                 if ((C, Ck) ∈ ?)    // Ck  is a subclass
                        Ck = ⟨c, A¢⟩,  A¢ = A ∪ {⟨a, V – v , d⟩}– {⟨a, V, d⟩}
   case (integer-range-attribute):
            ⟨ r¢L , r¢H⟩ = v  // v  is a new integer range input by the user
            C = ⟨c, A¢⟩,  A¢ = A ∪ {⟨ r¢L, r¢H⟩} – {⟨ rL, rH⟩}  //updated with the new range
            for each Ck ∈  C do {
               ⟨ rkL, rkH⟩ ∈ Ak , ⟨c, Ak⟩ = Ck
               if ( (C, Ck) ∈ ? ) {                          // Ck  is a subclass
                     if ( r¢L > rkL  )  r¢kL = r¢L  else r¢kL = rkL
                     if ( r¢H < rkH ) r¢kH = r¢H  else r¢kH = rkH
                     Ck = ⟨c, A¢⟩,  A¢ = A ∪ {⟨ r¢kL, r¢kH⟩} – {⟨ rkL, rkH⟩} //updated  subclass
               } else if ( (Ck, C) ∈ ? ) {     // Ck  is a superclass
                     if ( r¢L < rkL  )  r¢kL = r¢L  else r¢kL = rkL
                     if ( r¢H > rkH ) r¢kH = r¢H  else r¢kH = rkH
                     Ck = ⟨c, A¢⟩,  A¢ = A ∪ {⟨ r¢kL, r¢kH⟩} – {⟨ rkL, rkH⟩} //updated  superclass
               }
            }
   T¢ = ⟨∪k Ck, t, ? ⟩
}
```

Figure 2: Algorithm 1: Editing an Attribute

5.3.3 Merging two classes

The operation of merging classes is to create a new class from two existing classes which in some ways overlap. In our approach, only the properties (i.e. attributes) of classes are used to measure the similarity.

Algorithm 6: Once the similarity between two classes is larger than the threshold value, (it is set to 0.5 by default, although a user can change it according to her needs.), the two classes can be merged to create a new class, which includes the attribute values of the two classes. Their instances and children are re-classified to the newly created class. Similar to the "Delete a Class" section, when the two classes are merged all references to the original classes which occur as attribute values in other classes, have to be replaced by the name of the merged class.

Cosine Similarity

We define an algorithm that assesses similarity by comparing the attribute values of the classes. Cosine similarity is used to calculate the similarity between two classes, its value is always between 0 and 1.

$$\text{Cosine Similarity} = \frac{\text{Number of terms in Common}}{\text{Normalized}}$$

The *cosine similarity function* [11] between these objects, CS_{XY}, treats the set of attributes as components of an M-dimensional vector, and the similarity is the cosine of the angle between these vectors (their dot product divided by their magnitudes). This similarity is given by the expression:

$$CS_{XY} = \frac{\sum_{i=1}^{L} (X_i * Y_i)}{\sqrt{\sum_{i=1}^{L} X_i^2 \sum_{i=1}^{L} Y_i^2}}$$, where X and Y are two vectors

The original function is modified to fit our approach. As all classes have the same attributes (but different attribute values), the attributes cannot distinguish the difference between classes. In this case, the comparison of classes is done by the attribute values of each class, then, the dot product of classes takes the attribute values into account. Thus the revised formula is:

Given: $C_A = \langle c_a , \boldsymbol{A}_a \rangle$, $C_B = \langle c_b , \boldsymbol{A}_b \rangle$, $\langle a_i, \boldsymbol{V}_{ai}, d_{ai} \rangle \in \boldsymbol{A}_a$, $\langle a_i, \boldsymbol{V}_{bi}, d_{bi} \rangle \in \boldsymbol{A}_b$, $1 \le i \le n$

$$\text{Cosine\_Similarity}(C_A, C_B) = \frac{\sum_{i=1}^{n} \{\text{AV}(\langle a_i, \boldsymbol{V}_{ai}, d_{ai} \rangle) * d_{ai}\}\{\text{AV}(\langle a_i, \boldsymbol{V}_{bi}, d_{bi} \rangle) * d_{bi}\}}{\sqrt{\sum_{i=1}^{n} d_{ai}^2 \sum_{i=1}^{n} d_{bi}^2}}$$

Method of calculating attribute values:

Attribute value for attributes in $\boldsymbol{A_a}$: $\text{AV}(\langle a, \boldsymbol{V}_a, d_a \rangle) = \dfrac{|\boldsymbol{V}_a \cap \boldsymbol{V}_b|}{|\boldsymbol{V_a}|}$

Attribute value for attributes in $\boldsymbol{A_b}$: $\text{AV}(\langle a, \boldsymbol{V}_b, d_b \rangle) = \dfrac{|\boldsymbol{V}_a \cap \boldsymbol{V}_b|}{|\boldsymbol{V}_b|}$

```
Delete_Class(T, C, T¢ ){     // a class C to be deleted
  Attribute_Whose_Value_C (T, C, T¢ ) //edit attributes whose value is referring to class C
   for each C_k ∈  C do
          if ( (C, C_k) ∈ ? )     C_sub =  ∪ C_k            // C_sub is a set of subclasses
   switch( class_choice )
          case (delete ):         C¢ = C − C_sub − C
          case (reconnect-to-the-root ):
             for each C_k ∈  C do
                    if ( (C, C_k) ∈ ? ) {                   // C_k  is a subclass
                        set  ( C_root , C¢_k ) ∈ ? , where C_root = root ( T )
                        C¢_sub =  ∪ C¢_k              // C¢_sub  is a set of  reconnected subclasses
                    }
             C¢ = C ∪ C¢_sub − C_sub − C
          case (reconnect-to-the-super-class ):
            for each C_k ∈  C do
                    if ( (C, C_k) ∈ ? ) {                   // C_k  is a subclass
                        set ( C_super , C¢_k ) ∈ ? ,  where ( C_super , C ) ∈ ?
                        C¢_sub =  ∪_k C¢_k,              // C¢_sub  is a set of  reconnected subclasses
                    }
           T¢  =⟨ C ∪ C¢_sub − C_sub − C , t, ? ⟩
}
```

Figure 3. Algorithm 5: Deleting a Class

5.3.4 Splitting a class

Any class can be split into two or more classes.

Algorithm 7: A class is split into *m* new classes, the attributes of the split classes are the same as the original class by default, and therefore, the user is required to either create a new dominant attribute or edit their dominant attributes, so as to distinguish them. The subclasses will be re-classified according to their dominant attributes. Similar to the "Delete a Class" section, all references in the attribute lists to other classes will need to be updated appropriately.

5.4 Refinement Strategies

It is important to ensure that any taxonomy has a consistent structure, which we do in ReTAX by specifying a number of consistency rules. (Ideally we would also like to ensure that the taxonomy accurately models some aspect of the world, but this is far harder to achieve, as is involves an evaluation of the semantics of the taxonomy.) Here, we define three consistency rules for taxonomies, the system then checks for inconsistencies based on the rules, and suggests appropriate refinement strategies to the user to achieve a consistent taxonomy. The rules constrain the attributes and relationships among the classes.

Rule 1. All the children siblings of a class are more specific than their parent

$$\text{R1}(T) =_{\text{def}} \forall C \Big((\boldsymbol{V} \subset \boldsymbol{V_P} \mid \exists \langle a, \boldsymbol{V}, d \rangle \in \text{dom}(C), \exists \langle a, \boldsymbol{V_P}, d_p \rangle \in \text{dom}(C_p)) \wedge$$
$$(\boldsymbol{V}\text{¢} \subseteq \boldsymbol{V}\text{¢}_{\boldsymbol{P}} \mid \forall \langle a\text{¢}, \boldsymbol{V}\text{¢}, d\text{¢} \rangle \in \boldsymbol{A}, \forall \langle a\text{¢}, \boldsymbol{V}\text{¢}_{\boldsymbol{p}}, d_p\text{¢} \rangle \in \boldsymbol{A_p},$$
$$\boldsymbol{V} \neq \boldsymbol{V}\text{¢}, \ \boldsymbol{V_P} \neq \boldsymbol{V}\text{¢}_{\boldsymbol{P}}, (C_p, C) \in ?, C \in \boldsymbol{C}, C_p \in \boldsymbol{C} \Big)$$

Rule 1 basically determines the membership of a class in the taxonomy [1]. All the attribute values of a child are inherited from its parent, these values may be either the same as the corresponding attribute of its parent or a specialization of the corresponding attribute. In order to discriminate the child from its parent, there must be at least one dominant attribute which is more specific than its parent. This rule is violated if one of the attribute values of the child is not found in its parent, or none of attribute values in the child are more specific than its parent.

Rule 2. All the children of a particular class are distinct

$$R2(T) =_{\text{def}} \forall C \left(\boldsymbol{V} \perp \boldsymbol{V_s} \mid \exists \langle a, \boldsymbol{V}, d \rangle \in \text{dom}(C), \exists \langle a, \boldsymbol{V_s}, d_s \rangle \in \text{dom}(C_s), \forall C_s \in \boldsymbol{C_{sibling}}, \boldsymbol{C_{sibling}} \subset \boldsymbol{C}, (C_p, C) \in ?, (C_p, C_s) \in ? \right)$$

For each class, there exists at least a dominant attribute whose value is disjoint with its sibling classes. This rule is violated when a child or instance can belong simultaneously to two classes at the same level of the hierarchy.

Rule 3. There is at least one dominant attribute in each class. Every class has at least one dominate attribute to describe its characteristics.

$$R3(T) =_{\text{def}} \forall C \, (\text{dom}(C) \neq \varnothing \mid C \in \boldsymbol{C})$$

Definition 6 The taxonomy is consistent if and only if none of above rules are violated, that is,

$$\text{Consistent}(T) \leftrightarrow R1(T) \wedge R2(T) \wedge R3(T)$$

5.4.1 Consistency Checking

While checking inconsistencies, the user can choose the appropriate refinement strategies from those suggested by the system, and the user is often required to provide additional information to complete the operation.

<u>Roles of Sets and Elements:</u>

$C = \langle c, \boldsymbol{A} \rangle$, $\langle a, \boldsymbol{V}, d \rangle \in \boldsymbol{A}$;

$C_{parent} = \langle c_p, \boldsymbol{A_p} \rangle$, $\langle a, \boldsymbol{V_p}, d_p \rangle \in \boldsymbol{A_p}$;

$C_{sibling} = \langle c_b, \boldsymbol{A_s} \rangle$, $\langle a, \boldsymbol{V_s}, d_s \rangle \in \boldsymbol{A_s}$; $C_{sibling} \in \boldsymbol{C_s}$, $\boldsymbol{C_s} \subset \boldsymbol{C}$

$C_{sub} = \langle c_{sub}, \boldsymbol{A_{sub}} \rangle$, $\langle a, \boldsymbol{V_{sub}}, d_{sub} \rangle \in \boldsymbol{A_{sub}}$; $C_{sub} \in \boldsymbol{C_{sub}}$, $\boldsymbol{C_{sub}} \subset \boldsymbol{C}$

where $(C_{parent}, C) \in ?$, $(C_{parent}, C_{sibling}) \in ?$, $(C, C_{sub}) \in ?$

We now present four situations contradicting the taxonomy rules, and we show how the inconsistencies are resolved.

1. There is no dominant attribute in a class (Rule 3 is violated).
   Algorithm 8: Check if there is a subsidiary attribute in the class which is distinct among all the siblings. If found, set it to be dominant by increasing its *dri* to be higher than the threshold value *t*. Otherwise, the system asks the user to add a new attribute $\boldsymbol{A_{new}}$ to the sibling classes.
2. The values of a dominant attribute in a class are the same as its siblings (Rule 2 is violated).
   Algorithm 9: The user decides either to set that attribute in its siblings to be subsidiary (by decreasing the *dri* below the threshold value), or decides to edit its attribute values to make it distinct from its siblings (see Figure 4).

3. A dominant attribute of a child class is more general than its parent (Rule 1 is violated).
   Algorithm 10: The user decides either to set the dominant attribute to be subsidiary, or edit the dominant attribute to be more specific than its parent (see Figure 4).
4. An attribute value in a child class is not found in its parent (Rule 1 is violated).
   Algorithm 11: This new attribute value in a child class is added to its parent and its superclasses.

5.4.2 Redundancy Checking

The first three rules are the constraints that guarantee the well form of the taxonomy. However, it may not be enough just to assist domain experts building a consistent taxonomy. The usability, usefulness and quality of the taxonomy are important requirements, especially when one has to develop or expand a taxonomy. There are two subsidiary checks to eliminate redundant information in the taxonomy. When redundant information is detected, the user is reminded that that value may not be useful. He can choose either to delete or retain that value, which will be highlighted in a distinct colour if it is retained.

Redundant attribute value - A child inherits all attributes of its parent class. The values of all attributes of a class must appear in at least one of its children. If a value of an attribute does not appear in at least one subclass, then the user should be reminded that that value may be redundant.

$$R4(T) =_{\text{def}} \forall C \left(\boldsymbol{V_p} = \cup_k \boldsymbol{V_k} \mid \forall \langle c, \boldsymbol{A_k} \rangle \in \boldsymbol{C_{sub}}, \langle a, \boldsymbol{V_p}, d_p \rangle \in \boldsymbol{A_p}, \langle a, \boldsymbol{V_k}, d_k \rangle \in \boldsymbol{A_k}, (\langle c_p, \boldsymbol{A_p} \rangle, \langle c, \boldsymbol{A_k} \rangle) \in ?, 1 = k = |\boldsymbol{C_{sub}}|, \boldsymbol{C_{sub}} \subset \boldsymbol{C} \right)$$

Redundant attribute - Every attribute in a taxonomy should act as a dominant attribute in at least one class. This is because each attribute in the taxonomy should be used to distinguish at least one class, otherwise, it is redundant. The system informs the user there is such an attribute that is subsidiary in all classes in the taxonomy, and asks him either to delete or retain that attribute.

$$R5(T) =_{\text{def}} \forall \langle a, \boldsymbol{V}, d \rangle \left(d \geq t \mid \langle a, \boldsymbol{V}, d \rangle \in \boldsymbol{A}, C = \langle c, \boldsymbol{A} \rangle, \exists C \in \boldsymbol{C} \right)$$

6. Implementation

ReTAX+ is implemented in JAVA. It provides a user friendly interface to create and manage taxonomies. Figure 5 shows its interface components.

6.1 Planned Enhancements

Enhancing the User Interface: However, the step-wise guidance for the refinements needs to be improved; we aim to minimize the user's inputs by giving him options to choose from (e.g. a Yes/No option).

Refinement Logging: The taxonomy refinement audit trail can help experts keep track of the changes they make. Whenever a change is made to a taxonomy, the meta-information, such as change of description, reason of change, cost of change,

time of change and identity of the editor are all recorded in a log file [4]. The detailed log of all performed changes not only supports the reversibility requirement (explained below), but also allows other experts to appreciate the reasons for the changes made.

Reversibility Functionality: There are numerous circumstances where the user needs to reverse the effects of changes. The reversibility functionality allows undoing changes at the user's request [4], thus, the user can choose one of the options without worrying whether an inconsistent taxonomy might result.

7. Conclusion

In this paper, we presented the ReTAX+ framework for creating and managing taxonomies. The framework includes functions for editing classes and attributes, merging and splitting classes and inconsistency refinements. To enable the user to obtain the taxonomy most suitable for his needs, we introduce a list of possible refinement strategies, enable him to resolve the inconsistencies as best suites his requirements. Our hope is that the taxonomies which are produced by ReTAX+ will be of a higher quality and hence they will be reused in a number of different applications.

```
Siblings_Not_Distinct ( T, C , action,  T¢ ){
        //action = set the attribute to be subsidiary or edit the dominant attribute
        boolean same = false;
        for each C_sk ∈  C_s do {          //update its sibling classes
            if ( V = V_sk | ⟨a, V_sk, d_sk ⟩ ∈ dom(C_sk ) ){
              switch ( action)
                case(set-to-be-subsidiary ):
                    set d_sk¢ < t
                    A_sk¢ = A ∪ {⟨a, V_sk, d¢_sk ⟩} -  {⟨a, V_sk, d_sk ⟩}
                    C_s¢ = C_s ∪ {⟨c, A_sk¢⟩} – {⟨c, A_sk⟩}
                     same = true;
                case (edit-attribute):
                    Edit_Attribute ( T, ⟨a, V_sk, d_sk ⟩ , C_sk, T¢ )
            }
        }
        if(same){        //update C itself, if its attributes are the same as its siblings
            switch ( action)
                case(set-to-be-subsidiary ):
                    set d¢ < t
                    A¢ = A ∪ {⟨a, V, d¢⟩} -  {⟨a, V, d ⟩}
                    C¢ = C ∪ {⟨c, A¢⟩} – {⟨c, A⟩}
                case (edit-attribute):
                    Edit_Attribute ( T, ⟨a, V, d ⟩ , C, T¢ )
            T¢ = ⟨C¢ ∪C_s¢ – C_s , t, ? ⟩
        }
}
```

Figure 4. Algorithm 9: Siblings are not distinct

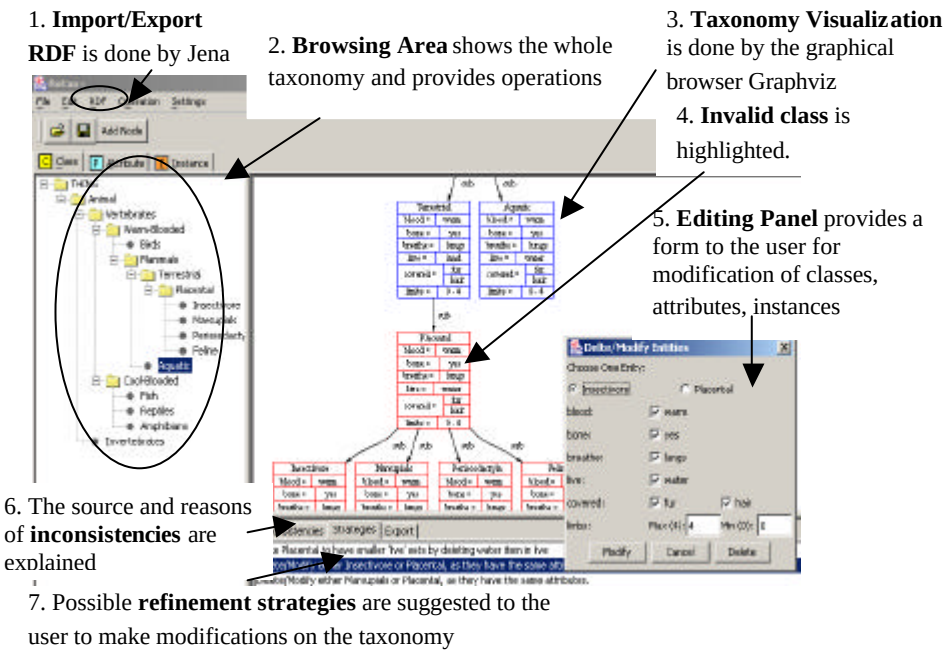

Figure 5. Screen Shot of ReTAX+ User Interface

References

[1] E. Alberdi, D. Sleeman, "ReTAX: a step in the automation of taxonomic revision". In Artificial Intelligent 91, p257 – 279, 1997

[2] O. Corcho, M. Fernández-López, A. Gómez-Pérez, O. Vicente, "WebODE: an integrated workbench for ontology representation, reasoning and exchange". In: 13th International Conference on Knowledge Acquisition and Knowledge Management, 2002

[3] Y. Sure, "On-To-Knowledge – Ontology based Knowledge Management Tools and their Application". In: German Journal Kuenstliche Intelligenz, Special Issue on Knowledge Management (1/02), 2002

[4] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, "User-driven Ontology Evolution Management", In Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW, Madrid, Spain, 2002.

[5] R. Mizoguchi, "Ontology Engineering Environments", Handbook of Ontologies, chapter 14, Springer, 2004

[6] L. Stojanovic, B. Motik, "Ontology Evolution within Ontology Editors", In ECON: Evaluation of Ontology-based Tools, 2002

[7] N. Guarino, C. Welty, "Evaluating Ontological Decisions with OntoClean", In: Communications of the ACM, Feb 2002

[8] F. N. Natalya, M. A. Musen, "The PROMPT suite: interactive tools for ontology merging and mapping". In: International Journal of Human-computer Studies, 2003

[9] Y. Sure, J. Angele, S. Staab, "OntoEdit: Guiding Ontology Development by Methodology and Inferencing", 2002

[10] R. Brachamn, "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", IEEE Computer, page 30-36, 1983

[11] B. T. Luke, "Distances in Clustering", Online Lecture Notes, 2003