Screening for Inconsistencies and Changes in Semantic Web Ontologies: Experiments with Protégé

Saumil Shah, David Thiery, Neli Zlatareva
Department of Computer Science,
Central Connecticut State University
1615 Stanley Street, New Britain, CT 06050, USA

ABSTRACT

In this paper, we discuss and characterize some common types of deficiencies in Semantic web ontologies. We are especially interested in various types of inconsistencies that may occur as a result of ontology merger. The current state-of-the-art in ontology evaluation is briefly reviewed, and some pitfalls of ontology merging are outlined. Our experiences with the state-of-the-art ontology development environment Protégé are presented and the need for more powerful ontology evaluation techniques able to tolerate some types of inconsistencies in merged ontologies is identified. To address this need, we discuss possible ways for resolving inconsistencies, and review one algorithmic solution to allow for automatic rewrite of certain types of inconsistencies in terminological hierarchies. An experimental evaluation of the letter is presented.

Keywords: Semantic Web, Protégé, Ontology Evaluation, Ontology Merging, Ontology Verification.

1. INTRODUCTION

Semantic Web is expected to improve and extend current web services in order to make them more useful for human use. It is to be built by using knowledge representation languages that create explicit domain conceptualizations called ontologies. Ontologies are intended to support various types of information management, including storage and sharing data on the web. Building web ontologies requires various software tools. On the shelf commercial tools, such as Medius Visual Ontology Modeler [1], LinKFactor Workbench [2], and K_infinity [3], can be applied to several stages of ontology life cycle including development, implementation, and maintenance, but they are limited in Java platforms, interoperability and interface with commercial databases. The latest ontology environments such as Web Onto [4] and Protégé [5] provide more advanced development capabilities including tools for screening ontologies for inconsistencies. As the size and the number of web ontologies grow, the necessity to evaluate and ensure their consistency and interoperability is becoming a major issue.

In this paper, we discuss and characterize some common types of deficiencies in web ontologies with emphasis on merged ontologies. We are especially interested in various types of inconsistencies relevant to merged ontologies. A common approach to handle inconsistencies is to attempt to dump problematic information. We feel, however, that this approach limits the utility of the Semantic Web, and instead we need reasoning systems that can tolerate inconsistent information such as the one presented in [6]. To study the limitations of current ontology evaluation tools employing Description Logics (DL), we experimented with Protégé [5], which is a free, open source ontology editor and knowledge-base development framework. Its flexible open-source platform is easy to combine with custom tailored components to build real-world applications. In our experiments, we used Protégé OWL version, which is a plug-in to the Protégé ontology development platform. Protégé OWL allows users to not only develop ontologies, but also to perform various evaluation tests on them including consistency checking. For that, Protégé uses Racer [7], a Description Logics reasoner which is able to compute infered ontology class hierarchy, perform consistency checking, and examine the possibility for a class to have instances. Racer inference capabilities, however, are limited to satisfiability, subsumption, equivalence and disjointness, thus leaving some common types of inconsistencies undetectable. In the paper, we present examples to illustrate such inconsistencies, and discuss ways to extend the logical power of ontology consistency checker.

The paper is organized as follows. First, we briefly review the state-of-the-art in ontology evaluation and characterize common deficiencies in single and merged web ontologies. Section 4 describes some pitfalls of ontology merging, and presents a case study performed with Protégé to illustrate the need for more powerful ontology evaluation techniques capable to handle different types of inconsistencies in merged ontologies.
2. COMMON DEFICIENCIES IN SINGLE AND MERGED WEB ONTOLOGIES

The current state-of-the-art in ontology evaluation is rather amorphous. Existing approaches address more or less specific evaluation issues, and often more than one quality-criterion is discussed at the same time, thus only partially clarifying the problems at stake. Even the process of evaluating web ontologies has not been formally established. In this section, we discuss some common types of ontology deficiencies, and outline some of the techniques and tools for ontology evaluation that are beginning to emerge.

2.1 Inconsistency Errors

There are three main types of errors that cause inconsistency in the ontology:

- **Circulatory errors.** These occur when a class is defined as a subclass or a superclass of itself at some level of ontology hierarchy. OWL ontologies provide constructs to form property hierarchies, where circulatory errors are likely to occur.
- **Partition errors.** We have encountered two types of partition errors depending on the type of breakdown of superclass into subclasses. When all the features of subclasses are independently described and subclasses do not overlap we have a disjoint decomposition error. In this case, the ontology contains an instance that belong to various disjoint subclasses or a common class which is a subclass of disjoint classes. An exhaustive decomposition error occurs when ontology defines a complete decomposition or partition of a class into many subclasses but not all the instances of the base class belong to the subclasses.
- **Semantic Inconsistency Errors.** Such errors occur when a concept is associated with a subclass that it does not belong to.

2.2 Incompleteness Errors

Incompleteness often creates ambiguity which leads to unreliable inferences. We have encountered the following types of incompleteness errors:

- **Incomplete Concept Classification.** This error occurs when ontology overlooks some of the concepts present in the domain while classifying a particular concept. For example, when classifying concept (Animal) Ability into WalkingAbility, FlyingAbility, and overlooking SwimAbility.
- **Partition Errors.** In [15], Gomez states that an ontology may lack important axioms or information about the classification of a concept, thus reducing its inferencing power. He has identified two types of errors that cause incomplete partition errors: (i) disjoint knowledge omission, which occurs when ontology classifies the concept into many subclasses and partitions, but omits disjoint knowledge axiom between them, and (ii) Exhaustive knowledge Omission, which occurs when an ontologies does not satisfy the completeness constraint while performing decomposition of concepts into subclass and partitions.

2.3 Redundancy Errors

Redundancy occurs when particular information is inferred more than once from the relations, classes and instances found in ontology. The following are common types of redundancies in web ontologies [8, 15]:

- **Redundancies of “Instance-of” relations.** Example: an ontology specifies instance Flock as an Instance-of Qty and FlyAbility classes, and it is already defined that Ability is a subclass of Animal. The explicit Instance-of relation between Flock and Animal creates redundancy as Flock is indirect instance of Animal as Animal is a superclass of Ability.
- **Redundancies of “Subclass-of” relations.** These occur when ontology specifies classes that have more than one Subclass-of relation directly or indirectly. For example, ontology specifies FlyAbility as a subclass of Ability and Animal, and Ability is also defined as a Subclass-of Animal. Here indirect Subclass-of relation exists between FlyAbility and Animal creating redundancy.
- **Redundancies of Subproperty-of relations.** Likewise, redundancy of Subproperty-of can exist while building property hierarchies.
- **Identical formal definition of classes, properties and instances.** These may occur when ontology defines different (or same) names for two classes, properties or instances respectively, but provides the same formal definitions.

3. ONTOLOGY EVALUATION TOOLS

Some of the most popular ontology evaluation techniques and tools are briefly reviewed below.

3.1 DL Implementation Group (DIG) Reasoners

OWL-DL is based on Description Logic (DL), which is a decidable fragment of First Order Logic (FOL). For a particular task, a logic is decidable if it is possible to design an algorithm that will terminate in a finite number of steps. For example, in DL it is possible to write an
algorithm that calculates whether or not one concept is a subclass of another concept, which is guaranteed to terminate after a finite number of steps. Because OWL-DL ontologies can be translated into a DL representation, it is possible to perform automated reasoning over the ontology using a Description Logic reasoner. A Description Logic reasoner performs various inference services, such as computing the inferred superclasses of a class, determining whether or not a class is consistent (a class is inconsistent if it cannot have any instances), deciding whether or not one class is subsumed by another class. Some of the most popular Description Logic reasoners are RACER [11], FaCT [13], FaCT++ [14], and Pellet [10], which we briefly review next.

3.1.1 Racer (Robust Server for Scalable Ontology Reasoning) Racer handles large Aboxes in combination with large and expressive Tboxes. It provides highly optimized standard and non-standard inference services for sophisticated ontology applications. Racer offers much more than OWL by supporting rules, constraint reasoning, and expressive query answering (e.g., in SPARQL syntax). Racer query language nRQL offers grounded conjunctive queries with head projection operators, negation as failure, aggregation operators, and server-side processing of query results (e.g., XML generation). Data persistency is provided by referring to AllegroGraph, an extremely fast triple store providing access to billions of triples.

3.1.2 The FaCT System A result of optimizing tableaux subsumption algorithms has been the development of the FaCT system. FaCT (Fast Classification of Terminologies) is a Description Logic (DL) classifier that can also be used for modal logic satisfiability testing. The FaCT system includes two reasoners, one for the SHF logic (ALC augmented with transitive roles, functional roles and a role hierarchy) and the other for the SHIQ logic (SHF augmented with inverse roles and qualified number restrictions), both of which use sound and complete tableaux algorithms. FaCT’s features: (1) expressive logic (especially SHIQ, which is sufficiently expressive to be used as a reasoner for the DLR logic and to reason with database schemata; (2) supports reasoning with arbitrary knowledge bases (i.e. those containing general concept inclusion axioms), (3) offers optimized tableaux implementation, which has become the standard for DL systems, and (4) supports CORBA-based client-server architecture.

3.1.3 FaCT++ FaCT++ is the new generation of FaCT OWL-DL reasoner. FaCT++ uses the established FaCT algorithms, but with a different internal architecture. Additionally, FaCT++ is implemented in C++ for increased efficiency and maximal portability.

3.1.4 Pellet Pellet is an open-source Java based OWL-DL reasoner. It can be used in conjunction with both Jena and OWL API libraries and also provides a DIG interface. Pellet API provides functionalities for ontology validation and consistency checking, taxonomy classification, checking of entailments, and answering a subset of RDQL queries (similar to ABox queries in DL terminology). Pellet is an OWL DL reasoner based on the tableaux algorithms developed for expressive Description Logics. It supports the full expressivity of OWL DL including reasoning about nominals (enumerated classes), which allows OWL constructs owl:oneOf and owl:hasValue to be used freely. Currently, Pellet is the first and only sound and complete DL reasoner that can handle this expressivity. Pellet ensures soundness and completeness by incorporating the recently developed decision procedure for SHOIQ (the expressivity of OWL-DL plus qualified cardinality restrictions in DL terminology).

3.2 DIG Compliant Description Logic Reasoners

Because writing a Description Logic reasoner is a non-trivial task and highly optimized third party Description Logic reasoners have already been developed, it does not make sense to write a reasoner for a specific application. Fortunately, a standard exists that provides a specification for a common way of connecting to third party DL reasoners. This standard is called the "DIG interface" (DIG is short for DL Implementation Group). A DIG compliant reasoner is a Description Logic reasoner that provides a standard access interface which enables the reasoner to be accessed over HTTP, using the DIG language. The DIG language is an XML-based representation of ontological entities such as classes, properties, and individuals, and also axioms such as subclass axioms, disjoint axioms, and equivalent class axioms. The DIG language contains constructs that allow clients to "tell" a reasoner about an ontology (i.e., describe an ontology to a reasoner), and also "ask" the reasoner about what it has inferred, such as subclass relationships, type relationships, etc. The advantage of DIG is that applications can communicate with any DIG compliant reasoner, without needing to know specific reasoner details or reasoner interaction protocols. This means that it is possible to "plug in" any DIG compliant reasoner into any DIG aware application. For example, RACER, FaCT++ or any other DIG compliant reasoner may be used with Protégé-OWL. Protégé-Owl provides an API that can be used to interact with an external DIG reasoner. The reasoning services are being provided by an external DIG reasoner and Protégé-Owl does not need to know the fine grain details of DIG in order to operate [10].
4. MERGING WEB ONTOLOGIES

When merging web ontologies, the following ontology properties must be accounted for:

- **Logical framework.** Ontologies to be merged may support different subsets and variations of DL. Sharing information between them can be done automatically if that information is expressed in a common representation framework. Other kinds of transfers may be possible, but some of the information may be lost or modified.

- **Terminology.** Ontologies may use different names for the same kinds of entities or may use the same names for different kinds. Sometimes, two concepts with different definitions are intended to be the same, but the task of proving their semantic consistency may be difficult or even impossible.

- **Implementation framework.** Even when the names and definitions of ontologies to be merged are identical, computational or implementation side effects may cause the same knowledge to behave differently in different systems. In some implementations, the order of entering rules and data may have an effect on the possible inferences and the results of computations. Sometimes, the side effects may cause a simple inference on one system to get hung up in an endless loop on another system.

4.1 Merging ontologies using PROMPT

PROMPT is a plug-in suite for Protégé used to manage multiple ontologies. It has four main functions [9]:

1. Compare the current ontology to a different version of the same ontology;
2. Move frames between the current including project and one of the included projects;
3. Merge two ontologies and added the resulting merged ontology to the current project;
4. Extract a portion of ontology and add it to the current project.

The compare function is the most relevant to our experiments. Within compare, there are two kinds of views: Tree View and Table View. In the Tree View, ontology structure is displayed and the changes are highlighted. This view aids the ontology user to accept/reject the changes made to the ontology. The Table View enables the user to save the changed record as an output file (text file) for further use. By enabling the “journaling” preference, information about the changes made to the ontology (e.g. author, time, item changed) will be saved to a file with the extension ‘.pjrn’. PROMPT is able to read such files and retrieve information about those changes.

PROMPT takes two ontologies as input and guides the user in the creation of one merged ontology as output. To do that, PROMPT first creates an initial list of matches based on class names, and then the following two-step process continues until all conflicts between ontologies to be merged are resolved:

1. The user triggers an operation by either selecting one of PROMPT’s suggestions from the list or by using an ontology-editing environment to specify the desired operation directly.
2. PROMPT performs the operation, automatically executes additional changes based on the type of the operation, generates a list of suggestions for the user based on the structure of the ontology around the arguments to the last operation, and determines conflicts that the last operation introduced in the ontology and finds possible solutions for those conflicts.

4.2 Case study: experiments with PROMPT

To experiment with Protégé and PROMPT, we have created two separate ontologies for animal classification, Animal1 and Animal2. Our goal was to merge them in a single ontology, Animalm, to study the limitations of PROMPT in detecting various types of deficiencies in the latter. We must note that Animalm includes some, but not necessarily all, frames from Animal1 and Animal2 with the frames that represent similar concepts merged into one frame. Also note that description of a merging task is necessarily informal one since many of the decisions in the merging process rest with the ontology designer and depend on the task for which the designer plans to use the merged ontology. Our approach was the following:

1. Identify a set of knowledge-base operations for ontology merging or alignment.
2. For each operation in this set, define:
   a. Changes that PROMPT performs automatically.
   b. New merging suggestions that PROMPT presents to the user, and
   c. Inconsistencies and potential problems in the emerging merged ontology that the operation may introduce and that require user’s attention.

When the user invokes an operation, PROMPT creates members of these three sets based on the arguments to the specific invocation of the operation. Automatically, an image of a frame Fs from original ontology is a frame Fi in the merged ontology Animalm that represents the same concept in the domain that Fs. Finally, we can say that a frame Fi in the merged ontology Animalm is an image of a frame Fs from original ontology if: (1) Fs is a result of copying Fs into Animalm, or (2) Fi is a result of merging Fs with another frame, or (3) Fi is a result of merging and image of Fs with another frame.
4.2.1 Utilizing the Compare Task
PromptDiff allows you to compare two different versions of the same ontology. PROMPT uses a set of heuristics to compare two versions of an ontology and find corresponding frames in the two versions. In addition to comparing names and types of the frames, PromptDiff looks at the structure of two versions, suggesting, for example, that a single unmatched sibling of the same parent in the two different versions may be the same frame with a different name. By concentrating on the differences between two similar projects (rather than the similarities of two different projects as in merge), compare can give much better results for the type of reconciliation required in version control.

4.2.2 Inconsistencies in the Merged Ontology
Inconsistencies in OWL ontologies can easily occur, and it is a challenging task for ontology developers to resolve such inconsistencies. PROMPT can provide guidance on:

1. Selecting the axioms to modify.
2. Minimizing the impact of proposed ontology modifications.

For a bird-penguin example, two axioms cause Penguin to be unsatisfiable:

\[
\begin{align*}
\text{Bird} & \subseteq \text{Animal} \cap \text{CanFly} \quad (1) \\
\text{Penguin} & \subseteq \text{Bird} \cap \neg \text{CanFly}. \quad (2)
\end{align*}
\]

To resolve this problem, if we remove the part Bird \subseteq \text{CanFly} from axiom (1), then the implicit information Eagle \subseteq \text{CanFly} will be lost (Figure 1a). If we remove the part Penguin \subseteq \text{Bird} from axiom (2), then the implicit information Penguin \subseteq \text{Animal} will be lost (see Figure 1b). To minimize the loss of information, when Bird \subseteq \text{CanFly} is removed from axiom (1), the user should be notified that the information Eagle \subseteq \text{CanFly} must be added back to the ontology. Similarly, when Penguin \subseteq \text{Bird} is removed from axiom (2), the user should be notified that the information Penguin \subseteq \text{Animal} must be added back to the ontology. That is, whenever parts of an axiom or a whole axiom are removed, it frequently happens that intended entailments are lost. In order to minimize the impact on the ontology, we must account for the lost information of concepts due to the removal of (parts of) axioms. To handle such case, we can use the fine-grained approach [16] which makes it possible to distinguish between changes that are helpful in that they restore lost information due to removal of axioms, and those that are harmful in that they cause additional unsatisfiability.

Continuing the above example, when Bird \subseteq \text{CanFly} from axiom (1) is removed, the helpful change is Eagle \subseteq \text{CanFly}. When Penguin \subseteq \text{Bird} from axiom (2) is removed, the helpful change is Penguin \subseteq \text{Animal}.

![Figure 1.a](image1a.png) ![Figure 1.b](image1b.png)

Harmful changes to replace the part CanFly in axiom (1) are \neg \text{Animal}, Penguin, and Eagle. This is because if the part CanFly in axiom (1) is replaced by \neg \text{Animal}, then Bird becomes unsatisfiable. If that part is replaced by Penguin, then Penguin is defined as a type of Penguin. If it is replaced by Eagle, then Penguin is defined as a type of Eagle.

Consider the following incomplete definition of the predicate \text{animal}(X):

\[
\begin{align*}
1. \quad \text{animal}(X) & \iff \text{birds}(X) \\
2. \quad \text{animal}(X) & \iff \text{penguin}(X) \\
3. \quad \text{birds}(X) & \iff \text{fly}(X) \\
4. \quad \text{penguin}(X) & \iff \text{cant\_fly}(X)
\end{align*}
\]

Using completion as above, we can derive the following constraint:

\[
\begin{align*}
1. \quad \text{animal}(X) & \implies \text{birds}(X) \lor \text{penguin}(X) \\
2. \quad \text{penguin}(X) & \implies \text{cant\_fly}(X) \\
3. \quad \text{birds}(X) & \implies \text{fly}(X)
\end{align*}
\]

While performing the above operations, PROMPT checks for merged classes, merged slots, and merged instances by performing a deep or a shallow copy of a class from one ontology to another, as well as for inconsistencies and potential problems in the merged ontology as a result of these operations. Note that not all of the above cases constitute a semantic inconsistency; some of the items in the list below indicate only a potential problem in the merged ontology that requires the user’s attention.

**Name conflicts:** A name conflict occurs when there is more than one frame with the same name in the same ontology. For example, we can copy the class Birds from Penguin_Cant_Fly ontology into the merged ontology. If we then copy a slot Birds that may exist in the Birds_Fly ontology, there will be a name conflict. To resolve this problem, PROMPT suggests renaming one of the offending frames.
**Dangling references:** When we bring in frame from one ontology to another, we may not bring all the frames that the moved frames refer to. As a result, a frame will refer to another frame that does not yet exist in the merged ontology. For example, the definition for the merged class Birds contains a slot Penguin which has the class Pingu as its range. However, if we have not yet copied the Pingu class, the reference to it will be a dangling one. To resolve this problem, PROMPT suggests copying the dangling frame into the merged ontology.

**Redundancy in the class hierarchy:** Links in a class hierarchy may be redundant if there is more than one path from a class to a superclass.

### 4.2.3 Merged-Classes Operation

We have found merge-classes operation to be especially useful, because it lists the changes that iPrompt makes automatically, offers new suggestions, and detects inconsistencies in merged ontology. Suppose the user is merging two ontologies and performs a merge-classes operation for two classes A and B to create a new class M. To execute merge-classes operation, iPrompt performs the following actions:

1. Creates a new class M. If classes A and B had the same names, assign that name to class M. Otherwise, asks the user which name to choose (unless he has designated one ontology as preferred).
2. For each superclass C of A or B that has an image \( C_i \) in \( O_m \), it makes \( C_i' \) a superclass of M (thus restoring the original relation). The same is done for subclasses. For example, if we have previously copied the Penguin (rdf comment: Penguin Name) class from the Penguin_Cant_Fly into the merged ontology, the merged Penguin class would become a subclass of this copied Penguin class. When later we merge the two Penguin class, the merged Penguin class will become a subclass of the merged Bird class.
3. For each slot S that was attached to A or B in the original ontologies, if there is no image of S in \( O_m \), it copies S to \( O_m \). Then, for each slot S that was attached to A or B, it attaches its image to M.
4. For each pair of superclasses and subclasses of M that have similar names, it suggests that they are merged.

### 5. CONCLUSION

In this paper, we have briefly reviewed the current state-of-the-art in web ontology evaluation and concluded that more powerful evaluation techniques are needed to enable interoperability, improve the accuracy of ontology mapping and merging, and diminish the need for human interference during ontology evaluation process. We have identified and characterized some common ontological deficiencies and errors, and described our experiences with Protégé [10] and PROMPT (a plug-in suite for Protégé) [9] in building, managing, and merging multiple ontologies. We have shown that logical power of DL-based reasoners is not sufficient to identify and handle some types of inconsistencies and more powerful reasoning techniques should be employed to handle such cases. To address this need, we discussed possible ways of resolving inconsistencies, and reviewed one algorithmic solution to allow for automatic rewrite of certain types of inconsistencies in terminological hierarchies [16]. An experimental evaluation of the latter was presented.

### 6. REFERENCES