Resolving Inconsistencies in Evolving Ontologies

Peter Plessers, Olga De Troyer

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{Peter.Plessers, Olga.DeTroyer}@vub.ac.be

Abstract. Changing a consistent ontology may turn the ontology into an inconsistent state. It is the task of an approach supporting ontology evolution to ensure an ontology evolves from one consistent state into another consistent state. In this paper, we focus on checking consistency of OWL DL ontologies. While existing reasoners allow detecting inconsistencies, determining why the ontology is inconsistent and offering solutions for these inconsistencies is far from trivial. We therefore propose an algorithm to select the axioms from an ontology causing the inconsistency, as well as a set of rules that ontology engineers can use to resolve the detected inconsistency.

1 Introduction

More and more, ontologies are finding their way into a wide variety of software systems. Not only do they serve as the foundation of the Semantic Web [3], ontologies are starting to be applied in content and document management, information integration and knowledge management systems. The use of ontologies enhances systems with extensive reasoning capabilities, improve query possibilities, and ease integration and cooperation between systems.

Until recently, ontologies were mainly *treated* as being static i.e. once the ontology was developed and deployed, the knowledge captured by the ontology was considered to be fixed. Nevertheless, there is a need for ontologies to evolve in course of their lifetime. Reasons for ontology evolution includes changes to the domain represented, modifications of user requirements and corrections of design flaws. As ontologies may be shared by different applications and extended by other ontologies, a manual and ad-hoc handling of such an ontology evolution process is not feasible nor desirable as it is a too laborious, time intensive and complex process [17]. A structured approach is therefore essential to support the ontology engineer in this evolution process.

As ontologies are used to reason about and to infer implicit knowledge from, it is essential for an approach supporting ontology evolution to ensure that ontologies evolve from one consistent state into another consistent state. As changes to an ontology may possibly introduce inconsistencies, a method to detect and resolve inconsistencies in the ontology is required. For OWL DL[1], several reasoners capable of checking for inconsistencies have been developed (e.g., RACER [8], Fact [5], Pellet [9]). These reasoners are based on the description logics tableau algorithm. While such

[1] http://www.w3.org/TR/owl-ref/

reasoners allow detecting inconsistencies, determining *why* the ontology is inconsistent and *how* to resolve these inconsistencies is far from trivial. However, pinpointing the concepts that lead to an inconsistent ontology, determining the reasons for the inconsistencies and using these to offer the ontology engineer suggestions how to resolve these inconsistencies *should* be part of an ontology evolution approach.

In literature, three forms of ontology consistency are in general distinguished: Structural Consistency, Logical Consistency and User-defined Consistency [4]. The difference between these three forms is as follows:

- **Structural Consistency:** an ontology is considered structural consistent when the structure of the ontology conforms to the language constructs imposed by the underlying ontology language (e.g., OWL). Structural consistency can be enforced by checking a set of structural conditions defined for the underlying ontology language. Examples of such structural conditions include: 'the complement of a class must be a class', 'a property can only be a subproperty of a property', etc. In case of OWL, the set of structural conditions depends on the variant of OWL used. E.g., the set of structural conditions for OWL Lite will be more restrictive than these for OWL DL.
- **Logical Consistency:** an ontology is considered logical consistent when the ontology conforms to the underlying logical theory of the ontology language. In the case of OWL, this is a variant of description logics. E.g., specifying the range of a property requires the objects of all instantiations of this property to be in this range.
- **User-defined Consistency:** this form of consistency means that users can add their own, additional conditions that must be met in order for the ontology to be considered consistent. E.g., users could require that classes can only be defined as a subclass of at most one other class (i.e. preventing multiple inheritance) in order for the ontology to be considered consistent.

Most research in the field of ontology evolution concerning consistency has been focused on structural consistency. In this paper however, we focus on the problem of logical consistency. We therefore extend our previous work on ontology evolution [10, 11] to support the detection and resolving of logical inconsistencies within OWL DL (and by definition OWL Lite) ontologies. Checking logical consistency can be achieved by running a reasoner on the ontology. To achieve this, most state-of-the-art reasoners have adopted a description logic tableau algorithm as mentioned earlier. Although reasoners can be used to identify unsatisfiable concepts, they provide very little information about which axioms are actually causing the inconsistency. This makes it extremely difficult to offer the ontology engineer solutions to solve the inconsistency.

The contribution of this paper is twofold. First we present an approach that determines the axioms causing a logical inconsistency. We do this by extending the tableau algorithm so that it keeps track of both the transformations performed during the preprocessing step of the algorithm and the axioms (in transformed format) leading to a clash used in the execution of the tableau algorithm itself. Based on this extra information, we have defined an algorithm to determine the axioms causing the inconsistency. The second contribution concerns a set of rules that can be applied by the ontology engineer to actually resolve the inconsistency detected.

The paper is structured as follows. Section 2 describes the process of consistency checking and discusses the principles of a tableau algorithm. Section 3 introduces an algorithm to determine those axioms causing an inconsistency. In section 4, we present the set of rules that can be used by an ontology engineer to resolve the inconsistency detected. Section 5 discusses related work, and finally, Section 6 presents some conclusions.

2 Consistency Checking

The objective of our approach is to verify whether an ontology remains logical consistent after changes have been applied. We differentiate between two possible scenarios based on the common distinction found in literature between TBox (terminological or concept knowledge) and ABox (assertional or instance knowledge):

1. **An axiom was added to the TBox or an existing axiom from the TBox was modified.** To check logical consistency of the ontology, we require two tasks performed sequentially. First, we verify whether the concepts of the TBox itself are still satisfiable (without considering a possible ABox). We refer to this task as the *TBox Consistency Task*. Second, we verify if the ABox remains consistent w.r.t. the modified TBox, called the *ABox Consistency Task*.
2. **An axiom was added to the ABox or an existing axiom from the ABox was modified.** We verify if the ABox remains consistent w.r.t. its TBox (called *ABox Consistency Task*).

Note that we don't take the deletion of an axiom from either the TBox or ABox into account. Because OWL DL is based on a monotonic logic, an ontology can only become inconsistent when new axioms are added or existing ones are changed. An overview of the consistency checking process is shown in Figure 1.

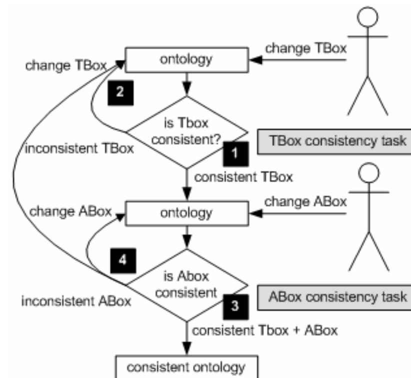

Fig. 1. Overview of consistency checking process.

When the user applies changes to the TBox, first the TBox Consistency Task (see ❶) is performed. An inconsistent TBox can be resolved by changing particular axi-

oms of the TBox (see ❷). Note that resolving inconsistencies is an iterative process as new changes may introduce new inconsistencies. When the TBox is consistent, the ABox Consistency Task is performed (see ❸). Inconsistencies in the ABox can be resolved either by changing particular axioms of the TBox (see ❷) so that the TBox conforms to the changed ABox, or by changing axioms of the ABox so that the changed ABox forms a valid model for the TBox (see ❹). In Section 3, we present an algorithm to determine which axioms are causing the inconsistency, while in Section 4 we introduce a set of rules that specify which changes can be applied to these axioms to resolve the inconsistency. The checking of TBox and ABox consistency is based on existing OWL reasoners. As the state-of-the-art reasoners are based on a tableau algorithm, we first give a short introduction of the tableau algorithm in the next subsection.

2.1 Tableau Algorithm

We focus in this paper, as already mentioned in the introduction, on the DL variant of OWL. OWL DL conforms to the $\mathcal{SHOIN}(\mathbf{D})$ description logic. The syntax of $\mathcal{SHOIN}(\mathbf{D})$ is summarized in Table 1. We adopt the following convention: A and B are atomic concepts, C and D are complex concepts, R is an abstract role, S is an abstract simple role, T and U are concrete roles, d is a datatype, a, b and c are individuals, and n is a non-negative integer. Based on this syntax, different types of axioms can be formed: concept equivalent axioms $C \equiv D$, concept inclusion axioms $C \sqsubseteq D$, role equivalent axioms $R \equiv S$, role inclusion axioms $R \sqsubseteq S$, transitivity axioms $Trans(R)$, inverse role axioms $R \equiv S^-$, symmetric role axioms $R \equiv R^-$, concept assertions $C(a)$, role assertions $R(a, b)$, individual equalities $a \approx b$ and inequalities $a \not\approx b$. Subsequently, we define an ontology O as a finite set of axioms.

Table 1 - SHOIN(D) syntax

Syntax	Description	Syntax	Description
$C \sqcap D$	Conjunction	$\leq n\ S$	Atmost restriction
$C \sqcup D$	Disjunction	$\geq n\ S$	Atleast restriction
$\neg C$ or $\neg d$	Negation	$\exists T.d$	Datatype exists
$\exists R.C$	Exists restriction	$\forall T.d$	Datatype value
$\forall R.C$	Value restriction	$\leq n\ T$	Datatype atmost
$\{a, b, c\}$	Individuals	$\geq n\ T$	Datatype atleast

The tableau algorithm allows verifying both the *satisfiability* of a concept C w.r.t. a given TBox i.e. whether C doesn't denote the empty concept, as well as the *consistency* of a given ABox w.r.t. a TBox i.e. whether the assertions in the ABox form a valid model for the axioms defined in the TBox. An ontology O (composed of a TBox $\mathcal{T}$ and ABox $\mathcal{A}$) is considered to be logical consistent if all concepts of the TBox $\mathcal{T}$ are satisfiable and the ABox $\mathcal{A}$ is consistent w.r.t. to this TBox $\mathcal{T}$.

The basic principle of the tableau algorithm used when checking the satisfiability of a concept C is to gradually build a model $\mathcal{I}$ of C, i.e. an interpretation $\mathcal{I}$ in which $C^{\mathcal{I}}$ is not empty. The algorithm tries to build a tree-like model of the concept C by de-

composing C using tableau expansion rules. These rules correspond to constructors in the description logic. E.g., $C \sqcap D$ is decomposed into C and D, referring to the fact that if $a \in (C \sqcap D)^I$ then $a \in C^I$ and $a \in D^I$. The tableau algorithm ends when either no more rules are applicable or when a clash occurs. A clash is an obvious contradiction and exists in two forms: $C(a) \Leftrightarrow \neg C(a)$ and $(\leq n\ S) \Leftrightarrow (\geq m\ S)$ where $m > n$. A concept C is considered to be satisfiable when no more rules can be applied and no clashes occurred. The tableau algorithm can be straightforwardly extended to support consistency checking of ABoxes. The same set of expansion rules can be applied to the ABox, requiring that we add inequality assertions $a \not\approx b$ for every pair of distinct individual names.

Important to note is that, although the tableau algorithm allows us to check ontology consistency, the algorithm doesn't provide us any information regarding the axioms causing the inconsistency, neither does it suggest solutions to overcome the inconsistency. In the remainder of this paper we discuss how we can overcome these shortcomings.

3 Selecting Axioms Causing Inconsistency

In this section we discuss how we extend the tableau algorithm by keeping track of the internal transformations that occur during the preprocessing step and the axioms leading to a clash used in the execution of the algorithm. We therefore introduce *Axiom Transformation Trees* and *Concept Dependency Trees*. Next, we explain how such a Concept Dependency Tree is used to determine the axioms causing the inconsistency. In the last subsection, we explain the overall algorithm and illustrate it with an example.

3.1 Axiom Transformations and Concept Dependencies

To be able to determine the axioms causing an inconsistency, we keep track of both the axiom transformations that occur in the preprocessing step, and the axioms leading to a clash used during algorithm execution. The result of the preprocessing step is a collection of axiom transformations, represented by a set of *Axiom Transformation Trees* (ATT), while the axioms used are represented in a *Concept Dependency Tree* (CDT). We explain the construction of both the ATT and the CDT by means of a simple example. We consider for our example the following TBox $\mathcal{T}$ consisting of the following axioms: $\{PhDStudent \sqsubseteq \neg\forall enrolledIn.\neg Course$, $\exists enrolledIn.Course \sqsubseteq Undergraduate$, $Undergraduate \sqsubseteq \neg PhDStudent$, $PhDStudent_CS \sqsubseteq PhDStudent\}$.

3.1.1 Axiom Transformation Tree

We give an overview of the different kind of transformations that occur during the preprocessing step of the tableau algorithm:

- **Normalization:** The tableau algorithm expects axioms to be in Negation Normal Form (NNF) i.e. negation occurs only in front of concept names. Axioms can be transformed to NNF using De Morgan's rules and the usual rules for quantifiers.

For our example, this means that *PhDStudent* $\sqsubseteq$ ¬∀*enrolledIn*.¬*Course* is transformed to *PhDStudent* $\sqsubseteq$ ∃*enrolledIn*.*Course*. Other forms of normalization can be treated in a similar way.

- **Internalization:** Another task in the preprocessing step concerns the transformation of axioms to support General Concept Inclusion (GCI) of the form $C \sqsubseteq D$ where C and D are complex concepts. In contrast to subsumption relations between atomic concepts ($A \sqsubseteq B$), which are handled by expansion, this is not possible with GCI. To support GCI, $C \sqsubseteq D$ must first be transformed into $\top \sqsubseteq \neg C \sqcup D$ (meaning that any individual must belong to $\neg C \sqcup D$). In our example, ∃*enrolledIn*.*Course* $\sqsubseteq$ *Undergraduate* is transformed to $\top \sqsubseteq$ *Undergraduate* $\sqcup$ ∀*enrolledIn*.¬*Course*.
- **Absorption:** The problem with GCI axioms is that they are time-expensive to reason with due to the high-degree of non-determinism that they introduce [1]. They may degrade the performance of the tableau algorithm to the extent that it becomes in practice non-terminating. The solution of this problem is to eliminate GCI axioms whenever possible. This is done by a technique called absorption that tries to absorb GCI axioms into primitive axiom definitions.
- **Axiom composition:** different axioms can be composed together into one axiom. E.g., the axioms $C \sqsubseteq A$ and $C \sqsubseteq B$ can be transformed to $C \sqsubseteq A \sqcap B$.

We introduce the notion of an *Axiom Transformation Tree* (ATT) to keep track of the transformations that occur during the preprocessing step i.e. an ATT stores the step-by-step transformation of the original axiom (as defined by the ontology engineer) to their transformed form. When later on the tableau algorithm ends with a clash, the ATTs can be used to retrieve the original axioms by following the inverse transformations from the axioms causing the clash (as found by the tableau algorithm) to the original ones. We define an Axiom Transformation Tree as follows:

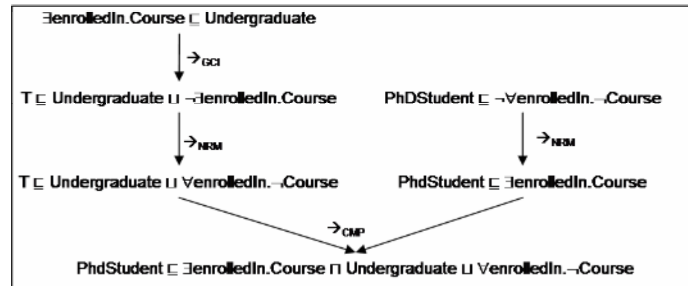

Fig. 2. An ATT for the given example.

Definition (ATT). An Axiom Transformation Tree, notation ATT, is a tree structure starting from one or more axioms $\phi_1, ..., \phi_n$, and ending with a transformed axiom ϕ'. Each branch of the tree represents a transformation and is accordingly labeled as follows:

- $\rightarrow_{NRM}$: transformation into normal form;
- $\rightarrow_{ABS}$: absorption of axioms into primitive axiom definitions;
- $\rightarrow_{GCI}$: transformation of General Concept Inclusion axioms;

• $\rightarrow_{CMP}$: composition of axioms.

Figure 2 shows the ATT for the axioms ∃*enrolledIn*.*Course* ⊑ *Undergraduate* and *PhDStudent* ⊑ ¬∀*enrolledIn*.¬*Course* in our example.

3.1.2 Concept Dependency Tree

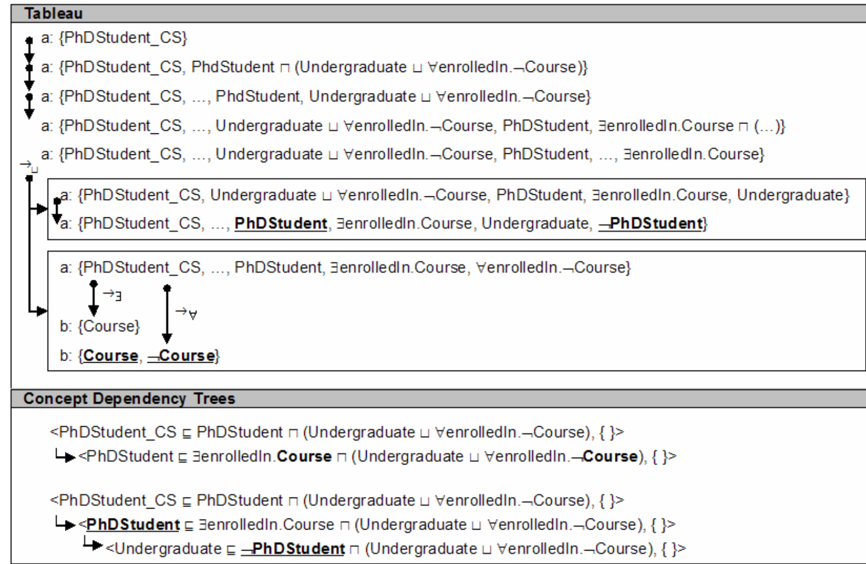

Fig. 3. Example tableau algorithm result and associated CDTs.

The tableau algorithm reasons with the transformed set of axioms resulting from the preprocessing step. In our example this means the following set: {*PhDStudent* ⊑ ∃*enrolledIn*.*Course* ⊓ (*Undergraduate* ⊔ ∀*enrolledIn*.¬*Course*), *Undergraduate* ⊑ ¬*PhDStudent* ⊓ (*Undergraduate* ⊔ ∀*enrolledIn*.¬*Course*), *PhDStudent_CS* ⊑ *PhDStudent* ⊓ (*Undergraduate* ⊔ ∀*enrolledIn*.¬*Course*)}. To test the satisfiability of a concept *C*, the set of tableau rules are applied to expand this concept until either a clash occurs or no more rules are applicable. We now want to store explicitly the different axioms that are used during the tableau reasoning process leading to a clash. We therefore introduce a *Concept Dependency Tree* (CDT):

Definition (CDT). We define a Concept Dependency Tree for a given concept *C*, notation CDT(*C*), as an n-ary tree where N_1, ..., N_n are nodes of the tree and a *child*(N_i,N_j) relation exists to represent an edge between two nodes in the tree. Furthermore, we define *parent* as the inverse relation of *child*, and *child** and *parent** as the transitive counterparts of respectively *child* and *parent*. A node N_i is a tuple of the form ⟨ϕ, **RA**⟩ where ϕ is a concept axiom and **RA** is a set of role axioms and assertions.

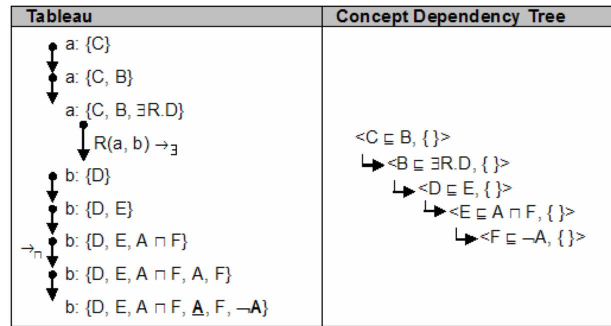

Fig. 4. Example of a CDT in the TBox Consistency Task

To construct a CDT, we keep track, for each node added to the tableau, of the path of axioms leading to the addition of that node. When a clash is found between two nodes, the paths of axioms associated with both nodes are used to construct the CDT. For each concept axiom ϕ represented in a path, we add a new node N to the CDT (unless such a node already exists) as child of the previous node (if any) so that N = $\langle \phi, \{\} \rangle$. When we encounter a role axiom or assertion ψ, we add it to the **RA** set of the current node N of the CDT so that $\psi \in$ **RA** where N = $\langle \phi,$ **RA**$\rangle$. Note that cyclic axioms (e.g., $C \sqsubseteq \forall R.C$) don't lead to the construction of an infinite CDT, as reasoners normally include some sort of cycle checking mechanism, such as blocking.

The result of the tableau algorithm testing the satisfiability of the concept *PhDStudent_CS* in our example is shown in Figure 3 at the top, while the CDTs are shown below. The tableau algorithm terminates with a clash between *PhDStudent*(*a*) ⇔ ¬*PhDStudent*(*a*) and between *Course*(*a*) ⇔ ¬*Course*(*a*). Note that non-deterministic branches in the tableau result in more than one CDT i.e. one for each non-deterministic branch. The CDTs contain the different axioms that lead from the concept examined (in our example *PhDStudent_CS*) to the cause of the inconsistency (the concepts involved in the clash).

3.2 Interpretation of Concept Dependency Trees

We use the CDTs to determine the axioms causing the inconsistency. The interpretation of a CDT differs for the TBox and ABox consistency task. In this section, we will discuss both interpretations.

3.2.1 TBox Consistency Task

The set of axioms of a CDT(*C*) can be seen as a MUPS (Minimal Unsatisfiability Preserving Sub-TBox) of the unsatisfiable concept *C*, i.e. the smallest set of axioms responsible for the unsatisfiable concept *C* [13]. Although removing one of the axioms of the CDT will resolve, by definition of a MUPS, the unsatisfiability of *C*, we consider it in general bad practice to take all axioms of a CDT into consideration to resolve inconsistencies. We will explain this by means of an example. Assume the fol-

lowing TBox: $\{C \sqsubseteq B, B \equiv \exists R.D, D \sqsubseteq E, E \sqsubseteq A \sqcap F, F \sqsubseteq \neg A\}$. Checking the satisfiability of C will reveal that C is unsatisfiable due to a clash between $A(b) \Leftrightarrow \neg A(b)$. The left side of Figure 4 shows the tableau, the right side the associated CDT.

Although removing for example the axiom $C \sqsubseteq B$ resolves the unsatisfiability of C, this change fails to address the true cause of the unsatisfiability as the overall TBox remains inconsistent. A concept is considered unsatisfiable if a clash is found in two deterministic branches of the tableau. This implies that the axioms containing the concepts involved in the clash must have a common parent in the CDT. Otherwise, no clash could have occurred between both concepts. Therefore, only the first common parent of these axioms and the axioms along the paths from this first common parent to the clashes are directly involved in the unsatisfiability problem. Changing axioms leading to this common parent (e.g. $C \sqsubseteq B$ or $D \sqsubseteq E$) may resolve the unsatisfiability of the concept under investigation, but doesn't tackle the true cause. We therefore introduce the notion of a *FirstCommonParent* for the CDT, and define it as follows:

Definition (FirstCommonParent). We define ϕ_c as the first common parent for two axioms ϕ_1 and ϕ_2, notation *FirstCommonParent*(ϕ_c, ϕ_1, ϕ_2), iff $\exists N_c \in$ CDT (*parent**(N_c, N_1) $\wedge$ *parent**(N_c, N_2) $\wedge$ $\neg\exists N_3$ (*parent**(N_3, N_1) $\wedge$ *parent**(N_3, N_2) $\wedge$ *child**(N_3, N_c) $\wedge$ $N_3 \neq N_c$)) where $N_c = \langle \phi c$, **RA**$\rangle$, $N_1 = \langle \phi_1$, **RA'**$\rangle$ and $N_2 = \langle \phi_2$, **RA''**$\rangle$.

In our example, the axiom $E \sqsubseteq A \sqcap F$ is the first common parent for the axioms containing the concepts involved in the clash. We therefore restrict the set of axioms causing the inconsistency to the following set: $\{E \sqsubseteq A \sqcap F, F \sqsubseteq \neg A\}$.

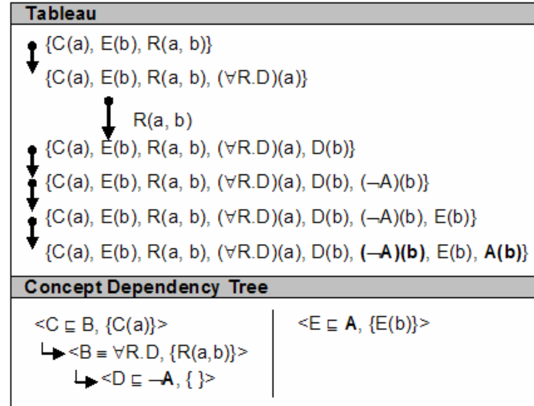

Fig. 5. Example of CDTs in the ABox consistency task.

3.2.2 ABox Consistency Task

The interpretation of the CDT differs for the ABox consistency task from the TBox consistency task. Consider the example with TBox: $\{C \sqsubseteq B, B \equiv \forall R.D, E \sqsubseteq A, D \sqsubseteq \neg A\}$ and ABox: $\{C(a)$, E(b)$\}$. Note that the TBox doesn't contain any unsatisfiable concepts (as we assume that the TBox consistency task was performed previously). Adding the assertion $R(a, b)$ to the ABox, will result in an inconsistent ABox as a

clash occurs between $A(b) \Leftrightarrow \neg A(b)$. At the top of Figure 5 the tableau is shown and at the bottom the CDTs.

Checking the ABox consistency for our example results in two CDTs, one for each individual checked (i.e., *a* and *b*). The axioms causing the inconsistency are the axioms resulting from both CDTs, together with axioms of the ABox used during the reasoning process (e.g., $R(a, b)$ as it allowed to trigger the $\rightarrow_\forall$ expansion rule). Note that we only consider axioms present in the original ABox i.e. no individuals added by the tableau algorithm to direct reasoning.

3.3 Axiom Selection

In this section, we give an overview of the overall algorithm to determine the axioms causing an inconsistency based on the interpretations of the CDT given in the previous section. Note that the axioms that will be considered differ for the TBox and ABox consistency task. The algorithm takes as input the clash information, CDTs and ATTs and outputs a set of axioms causing the inconsistency. Before explaining the complete algorithm, we fist need to address the following issues:

- **Mark axioms.** A complete axiom is not necessarily the cause of an inconsistency; instead only parts of the axiom may be the cause. Parts of axioms are causing an inconsistency either because they are the direct cause of the inconsistency, or because they are leading to a concept directly causing the inconsistency. The algorithm therefore marks those parts of the axioms. In order to do so, we introduce the *markAllParents* function that marks all parent nodes of the nodes containing a concept involved in the clash. The pseudo-code of the function is given below:

```
markAllParents(N):
  if not rootNode(N) then
    ϕ = getConceptAxiom(N);
    C_ϕ = getLeftPart(ϕ);
    N_parent = getParentNode(N);
    ϕ_parent = getConceptAxiom(N_parent);
    mark(def_ϕ, ϕ_parent);
    call markAllParents(N_parent)
  end if
```

- **Non-inconsistency-revealing clashes.** Clashes found between transformed axioms by the tableau algorithm, may not always indicate conflicting concepts in the original axioms as defined by the ontology engineer. Figure 3 (see Section 3.1.2) illustrates this. The clash $Course(b) \Leftrightarrow \neg Course(b)$ seems to reveal a contradiction , but when we transform the axioms back to their original form (i.e., $PhDStudent \sqsubseteq \neg\forall enrolledIn.\underline{\boldsymbol{\neg Course}}$ and $\exists enrolledIn.\underline{\boldsymbol{Course}} \sqsubseteq Undergraduate$) it is clear that they both refer to the same concept $\exists enrolledIn.Course$ (although one is in NNF while the other is not). The clash found guided the tableau algorithm, rather than revealing an actual inconsistency.

The structure of the overall algorithm is as follows:

1. For each clash $C(a) \Leftrightarrow D(a)$, lookup the concepts C and D in the leaf nodes of the associated CDTs, and mark these concepts.
2. For each marked node N, mark all parent nodes using the *markAllParents*(N) function.
3. Depending on the task performed (TBox or ABox consistency task) select for each CDT the axioms as described in Section 3.2. This results for each CDT in a set **S** containing the selected axioms.
4. For each set **S**, transform all axioms $\phi \in \mathbf{S}$ into their original form by applying the inverse transformations of the correct ATT.
5. The union of all sets **S** is the desired set of axioms.

Applying this algorithm to the example introduced in Section 3.1 results in the following set **S**. Underlined concepts are the concepts marked by the algorithm, underlined and bold concepts are the concepts involved in the consistency-revealing clash. $\mathbf{S} = \{\underline{\boldsymbol{PhDStudent}} \sqsubseteq \neg\forall enrolledIn.\underline{\neg Course}, \exists enrolledIn.\underline{Course} \sqsubseteq \underline{Undergraduate}, Undergraduate \sqsubseteq \underline{\boldsymbol{\neg PhDStudent}}\}$;

4 Resolving Inconsistencies

When an ontology is logical inconsistent this is because the axioms of the ontology are too restrictive as axioms are contradicting each other. To resolve the inconsistency, the restrictions imposed by the axioms should be weakened. In the previous section (see Section 3.3), we have defined an algorithm to determine the set of axioms causing the inconsistency. Changing one of these selected axioms will resolve the detected inconsistency. In the remainder of this section, we present a collection of rules that guides the ontology engineer towards a solution. A rule either calls another rule or applies a change to an axiom. Note that it remains the responsibility of the ontology engineer to decide which axiom he wants to change from the set provided by the approach.

Before we define the different rules, we first introduce the notion of class- and property hierarchy. We call $\mathcal{H}_c$ the class hierarchy of all classes present in the set **S** so that if $(C, D) \in \mathcal{H}_c$ then $C \sqsubseteq D$, and $\mathcal{H}_p$ the property hierarchy of all properties present in the set **S** so that if $(R, S) \in \mathcal{H}_p$ then $R \sqsubseteq S$. Note that these hierarchies don't include classes or properties not included in **S**. Furthermore, we define ψ_t as the top of a hierarchy $\mathcal{H}$ for a concept ψ, notation $top(\psi_t, \psi, \mathcal{H})$, iff $\psi_t \sqsubseteq \psi \wedge \neg\exists\omega \in \mathbf{S}: \omega \sqsubseteq \psi_t$. Analogous, we define ψ_l as the leaf of a hierarchy $\mathcal{H}$ for a concept ψ, notation $leaf(\psi_l, \psi, \mathcal{H})$, iff $\psi \sqsubseteq \psi_l \wedge \neg\exists\omega \in \mathbf{S}: \psi_l \sqsubseteq \omega$.

In the remainder of this section, we present a set of rules that guide the ontology engineer to a solution for the detected inconsistency. Note that we don't list the complete set of rules due to space restrictions. First, we define a set of rules that handle the different types of axioms. Secondly, we define the necessary rules to weaken or strengthen the different types of concepts. Note that axioms can always be weakened by removing the axiom. We therefore won't mention this option explicitly in the rules below. The rules for weakening axioms are given below:

- A concept definition $C \equiv D$ can be weakened either by removing the axiom or by weakening C or D (assuming $C \equiv D$ resulted from $C \sqsubseteq D$ in the CDT):
  (4.1) $weaken(C \equiv D) \Longrightarrow strengthen(C)$
  (4.2) $weaken(C \equiv D) \Longrightarrow weaken(D)$
- A concept inclusion axiom $C \sqsubseteq D$ can be weakened by removing the axiom, strengthening C or weakening D. The same rule applies for role inclusion axioms:
  (4.3) $weaken(C \sqsubseteq D) \Longrightarrow strengthen(C)$
  (4.4) $weaken(C \sqsubseteq D) \Longrightarrow weaken(D)$
- A concept assertion $C(a)$ can be weakened by either removing the axiom or by replacing C with a superclass:
  (4.5) $weaken(C(a)) \Longrightarrow change(C(a), D(a))$ where $X \sqsubseteq D$ and $leaf(X, C, \mathcal{H}_c)$
- A role assertion $R(a, b)$ can be weakened by either removing the axiom or by replacing R with a super-property:
  (4.6) $weaken(R(a, b)) \Longrightarrow change(R(a, b), S(a, b))$ where $X \sqsubseteq S$ and $leaf(X, R, \mathcal{H}_p)$

The second part of rules deal with the weakening and strengthening of concepts:

- A conjunction relation $C \sqcap D$ can be weakened (strengthened) by weakening (strengthening) either C or D. The rules for weakening are given below; the rules for strengthening are analogous:
  (4.7) IF $marked(C)$: $weaken(C \sqcap D) \Longrightarrow weaken(C)$
  (4.8) IF $marked(D)$: $weaken(C \sqcap D) \Longrightarrow weaken(D)$
  (4.9) IF $marked(C) \wedge marked(D)$: $weaken(C \sqcap D) \Longrightarrow weaken(C) \vee weaken(D)$
- A disjunction relation $C \sqcup D$ can be weakened (strengthened) by weakening (strengthening) C, D, or both C and D. The rules for weakening are given below; the rules for strengthening are analog:
  (4.10) IF $marked(C)$: $weaken(C \sqcup D) \Longrightarrow weaken(C)$
  (4.11) IF $marked(D)$: $weaken(C \sqcup D) \Longrightarrow weaken(D)$
  (4.12) IF $marked(C) \wedge marked(D)$: $weaken(C \sqcup D) \Longrightarrow weaken(C) \vee weaken(D)$
- An existential quantification $\exists R.C$ can be weakened and strengthened in two manners as it represents both a cardinality restriction ("at least one") and a value restriction. To weaken $\exists R.C$, we either remove $\exists R.C$ if it concerns a cardinality restriction violation, or we weaken C if it concerns a value restriction violation. To strengthen $\exists R.C$, we either add a minimum cardinality restriction if it concerns a cardinality restriction violation, or we strengthen C if it concerns a value restriction violation:
  (4.13) IF $marked(C)$: $weaken(\exists R.C) \Longrightarrow weaken(C)$
  (4.14) IF $marked(R)$: $strengthen(\exists R.C) \Longrightarrow add((\geq 2\ R))$
  (4.15) IF $marked(C)$: $strengthen(\exists R.C) \Longrightarrow strengthen(C)$
- A universal quantification $\forall R.C$ can be weakened (strengthened) by weakening (strengthening) C. The rule for weakening is given below; the rule for strengthening is analogous:
  (4.16) IF $weaken(\forall R.C) \Longrightarrow weaken(C)$
- A maximum cardinality restriction $(\leq n\ R)$ can be weakened either by raising n or by removing the cardinality restriction altogether. To strengthen $(\leq n\ R)$, we can lower n:

(4.17) $weaken((\leq n\ R)) \Longrightarrow changeCardinalityRestriction(R, m)$ where $m \geq 1$ if $(\leq n\ R)$ conflicts with $\exists R.C$, or $m \geq \alpha$ if $(\leq n\ R)$ conflicts with $(\geq \alpha\ R)$

(4.18) $weaken((\leq n\ R)) \Longrightarrow remove((\leq n\ R))$

(4.19) $strengthen((\leq n\ R)) \Longrightarrow changeCardinalityRestriction(R, m)$ where $m = 0$ if $(\leq n\ R)$ conflicts with $\exists R.C$, or $m \leq \alpha$ if $(\leq n\ R)$ conflicts with $(\geq \alpha\ R)$

- A minimum cardinality restriction $(\geq n\ R)$ can be weakened by either lowering n or by removing the cardinality restriction altogether. To strengthen $(\geq n\ R)$, we can raise n:

  (4.20) $weaken((\geq n\ R)) \Longrightarrow changeCardinalityRestriction(R, m)$ where $m \leq \alpha$ if $(\geq n\ R)$ conflicts with $(\leq \alpha\ R)$

  (4.21) $weaken((\geq n\ R)) \Longrightarrow remove((\leq n\ R))$

  (4.22) $strengthen((\geq n\ R)) \Longrightarrow changeCardinalityRestriction(R, m)$ where $m \geq \alpha$ if $(\geq n\ R)$ conflicts with $(\leq \alpha\ R)$

- A negation $\neg C$ is weakened by either removing $\neg C$ or by strengthening C. To strengthen $\neg C$, we need to weaken C:

  (4.23) $weaken(\neg C) \Longrightarrow strengthen(C)$

  (4.24) $strengthen(\neg C) \Longrightarrow weaken(C)$

- A concept A is weakened either by removing the concept or by replacing it with a superclass of A. To strengthen an atom concept A, we replace it with a subclass of A. When no (appropriate) sub- or superclass exists, we can create one first:

  (4.25) $weaken(A) \Longrightarrow change(A, B)$ where $X \sqsubseteq B$ and $leaf(X, A, \mathcal{H}_c)$

  (4.26) $strengthen(A) \Longrightarrow change(A, B)$ where $B \sqsubseteq X$ and $top(X, A, \mathcal{H}_c)$

We conclude this section with our example. The selection of axioms consisted of the following axioms: $\underline{\boldsymbol{PhDStudent}} \sqsubseteq \neg\forall enrolledIn.\underline{\neg Course}$, $\exists enrolledIn.\underline{Course} \sqsubseteq \underline{Undergraduate}$, $Undergraduate \sqsubseteq \underline{\boldsymbol{\neg PhDStudent}}$. If, for example, the ontology engineer beliefs that the axiom $\exists enrolledIn.\underline{Course} \sqsubseteq \underline{Undergraduate}$ doesn't reflect the real world situation, he could for example change the axiom to $\exists enrolledIn.Course \sqsubseteq Student$, assuming $Undergraduate \sqsubseteq Student$, by following the rules 4.4 and 4.25.

5 Related Work

Change management has been a long-term research interest. Noteworthy in the context of this paper is certainly the work on database schema evolution [13] and maintenance of knowledge-based systems [7]. When considering the problem of ontology evolution, only few approaches have been proposed. [15] defines the process of ontology evolution as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to depending artifacts. In [16], the authors propose a possible ontology evolution framework. They introduce a change representation and discuss the semantics of change for the KAON ontology language. A similar approach has been taken by [6] for the OWL language. The authors of [11][12] propose another approach for the OWL language based on the use of a version log to represent evolution. They define changes in terms of temporal queries on this version log.

On the topic of dealing with consistency maintenance for evolving ontologies, only very little research has been done. [4] presents an approach to localize an inconsistency based on the notion of a minimal inconsistent sub-ontology. The notion of a minimal inconsistent sub-ontology is very similar to the concept of a MUPS introduced by [14]. Although removing one axiom from the minimal inconsistent sub-ontology will resolve an unsatisfiable concept, it can not be guaranteed that this will solve the true cause of the inconsistency (as discussed in this paper). Furthermore, the approach doesn't marking indicating parts of axioms as cause of the inconsistency, but rather treats axioms as a whole.

Some related work has been carried out in explaining inconsistencies in OWL ontologies. The authors of [2] present a Symptom Ontology that aims to serve as a common language for identifying and describing semantic errors and warnings. The Symptom Ontology doesn't identify the cause of the ontology nor does it offer possible solutions to resolve an inconsistency.

Another interesting research area is the field of ontology debugging [10][18]. Their aim is to provide the ontology engineer with a more comprehensive explanation of the inconsistency than is generally provided by 'standard' ontology reasoners. We distinguish two types of approaches: black-box versus glass-box techniques. The first treats the reasoner as a 'black box' and uses standard inferences to locate the source of the inconsistency. The latter modifies the internals of the reasoner to reveal the cause of the problem. While black-box techniques don't add an overhead to the reasoner, more precise results can be obtained using a glass-box technique. Therefore, glass-box techniques are considered a better candidate in the context of ontology evolution. The authors of [9] discuss a glass-box approach which offers the users information about the clash found and selects the axioms causing the inconsistency (similar to a MUPS). The disadvantage of a MUPS is that it doesn't necessarily pinpoints the true cause of the inconsistency. Furthermore, the approach doesn't offer solutions to resolve the detected problem.

6 Conclusion

Ontologies are in general not static, but do evolve over time. An important aspect for evolving ontologies is that they evolve from one consistent state into another consistent state. Checking whether an ontology is consistent can be achieved by means of a reasoner. The problem is that it is in general extremely challenging for an ontology engineer to determine the cause of an inconsistency and possible solutions for the problem based on the output of a reasoner. We therefore presented an algorithm to select the axioms causing the inconsistency. Furthermore, we have presented a set of rules that ontology engineers can use to change the selected axioms to overcome the detected inconsistency.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The description logic handbook: theory, implementation and applications. Cambridge University Press. ISBN 0-521-78176-0 (2003)
2. Baclawski, K., Matheus, C., Kokar, M., Letkowski, J., Kogut, P.: Towards a symptom ontology for semantic web applications. In Proceedings of 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan (2004) 650-667
3. Berners Lee, T., Hendler, J., Lassila, O.: The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American (2001) 5(1)
4. Haase, P., Stojanovic, L.: Consistent evolution of OWL ontologies. Asunción Gómez-Pérez, Jérôme Euzenat (Eds.), The Semantic Web: Research and Applications, Second European Semantic Web Conference (ESWC 2005), Lecture Notes in Computer Science 3532 Springer 2005, ISBN 3-540-26124-9, Heraklion, Crete, Greece (2005) 182-197
5. Horrocks, I.: The fact system. In Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98, Springer-Verlag (1998) 307-312
6. Klein, M.: Change Management for Distributed Ontologies. PhD Thesis (2004)
7. Menzies, T.: Knowledge maintenance: the state of the art, The Knowledge Engineering Review (1999) 14(1) 1-46
8. Moller, R., Haarslev, V.: Racer system description. In Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2001), Siena, Italy (2001)
9. Parsia, B., Sirin, E.: Pellet: An OWL DL reasoner. In Ralf Moller Volker Haaslev (Eds.), Proceedings of the International Workshop on Description Logics (DL2004) (2004)
10. Parsia, B., Sirin, E., Kalyanpur, A.: Debugging OWL ontologies. In Proceedings of the 14th International World Wide Web Conference (WWW2005), Chiba, Japan (2005)
11. Plessers, P., De Troyer, O., Casteleyn, S.: Event-based modeling of evolution for semantic-driven systems. In Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Publ. Springer-Verlag, Porto, Portugal (2005)
12. Plessers, P., De Troyer, O.: Ontology change detection using a version log, In Proceedings of the 4th International Semantic Web Conference, Eds. Yolanda Gil, Enrico Motta, V.Richard Benjamins, Mark A. Musen, Publ. Springer-Verlag, ISBN 978-3-540-29754-3, Galway, Ireland (2005) 578-592
13. Roddick, J.F.: A survey of schema versioning issues for database systems, Information and Software Technology (1995) 37(7): 383-393.
14. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In Proceedings of IJCAI 2003 (2003)
15. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: Userdriven Ontology Evolution Management. In Proceeding of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW, Madrid, Spain (2002)
16. Stojanovic, L.: Methods and Tools for Ontology Evolution. Phd Thesis (2004)
17. Tallis, M., Gil, Y.: Designing scripts to guide users in modifying knowledge-based systems. In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI/IAAI 1999), Orlando, Florida, USA (1999) 242-249
18. Wang, H., Horridge, M., Rector, A., Drummond, N., Seidenberg, J.: Debugging OWL-DL ontologies: A heuristic approach. In Proceedings of the 4th International Semantic Web Conference, Eds. Yolanda Gil, Enrico Motta, V.Richard Benjamins, Mark A. Musen, Publ. Springer-Verlag, ISBN 978-3-540-29754-3, Galway, Ireland (2005)