Detecting Dirty Queries during Iterative Development of OWL Based Applications

Ramakrishna Soma[1] and Viktor K. Prasanna[2]

[1] Computer Science Department, University of Southern California, Los Angeles, CA 90089
rsoma@usc.edu
[2] Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089
prasanna@usc.edu

Abstract. Incremental/iterative development is often considered to be the best approach to develop large scale information management applications. In an application using an ontology as a central component at design and/or runtime (*ontology based* system) that is built using this approach, the ontology itself might be constantly modified to satisfy new and changing requirements. Since many other artifacts, e.g., queries, inter-component message formats, code, in the application are dependent on the ontology definition, changes to it necessitate changes to other artifacts and thus might prove to be very expensive. To alleviate this, we address the specific problem of detecting the SPARQL queries that need to be modified due to changes to an OWL ontology (T-Box). Our approach is based on a novel evaluation function for SPARQL queries, which maps a query to the extensions of T-Box elements. This evaluation is used to match the query with the semantics of the changes made to the ontology to determine if the query is *dirty*- i.e., needs to be modified. We present an implementation of the technique, integrated with a popular ontology development environment and provide an evaluation of our technique on a real-life as well as benchmark applications.

1 Introduction

OWL and RDF, the ontology languages proposed as a part of the semantic web standards stack, provide a rich set of data modeling primitives, precise semantics and standard XML based representation mechanisms for knowledge representation. Although originally intended to address the problem of finding and interpreting content on the World Wide Web, these standards have been proposed as an attractive alternative to traditional technologies used for addressing the information and knowledge management problems in large enterprises [7]. As more robust and scalable tools appear in the market, the motivation for applying semantic web technologies in large-scale enterprise wide solution increases steadily. A common use of these technologies is to build *ontology-based* systems [4]. In such systems an ontology, which is a formal model of the problem domain, is a key entity in the design of other system elements like the knowledge bases (KBs), queries to the KBs, messages between the components in the system, and the code itself.

Vast experience in building large-scale information systems, including those based on traditional RDBMS, data warehouses etc., point to the use of an incremental/iterative

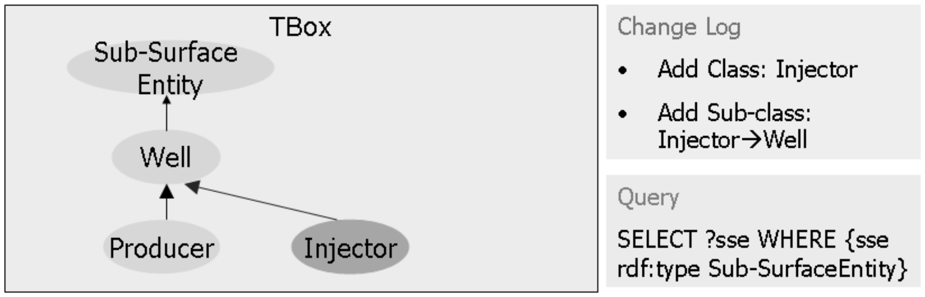

TBox
Sub-Surface Entity
Well
Producer
Injector
Change Log
Add Class: Injector
Add Sub-class: Injector→Well
Query
SELECT ?sse WHERE {sse rdf:type Sub-SurfaceEntity}

Fig. 1. A ontology change scenario

approach as being the most effective [2,6,13]. In such an approach, the requirements are assumed to be evolving as the system is developed (and used). The software itself is built in phases- with new requirements added at the beginning of each phase and working sub-systems delivered at the end of it. Frequent changes may thus be made to the ontology in order to accommodate the new requirements. Since other artifacts that form a part of the system are closely tied to the ontology, changes to the ontology necessitates changes to them. This may lead to a situation in which changes to an ontology could trigger an expensive chain of changes to other artifacts. Tools that ease the detection and performance of such changes can increase the productivity of the software engineers and hence reduce the cost of building software are thus very important for the success of such a development methodology.

We address a specific case of this broader problem for the class of applications that use OWL for representing the ontologies and SPARQL for the queries. In general, we assume that the ontologies are built ground up- perhaps reusing existing ontologies. Our technique uses the changes made to an OWL TBox to detect which queries need to be modified due to it- we call such queries *dirty* queries. To understand why this is non-trivial consider the following simple scenario shown in Fig. 1.

The original setup consists of a TBox with three classes (`Sub-SurfaceEntity, Well, Producer`) and a query to retrieve all `Sub-SurfaceEntity` elements. A new class `Injector` is then added to the TBox and specified as a sub-class to `Well` (this is recorded in the change log). A naive change detection algorithm [10] would have compared the entity names from the log (`Well, Injector`) to those in the query (`Sub-SurfaceEntity`), and determined that the query need not be modified. However, there could be a knowledge base consistent with the new ontology which contains statements asserting certain elements to be of type `Injector`. Since these elements/type assertions are not valid in any knowledge base consistent with the original ontology and will be returned as the results of the said query, we consider the query to be dirty. The naive algorithm does not detect this invalid example because the *semantics* are not considered in this approach (in this case the class hierarchy).

Thus our approach goes beyond the simple entity matching by considering the semantics of the ontology, the changes and the queries. A challenge we face in our approach arises because SPARQL is defined as a query language for RDF graphs and its relationship with OWL ontologies is not very obvious. We address this challenge by defining a novel evaluation function that maps the SPARQL queries to the domain of OWL semantic elements (Sect. 4). Then the semantics of the changes are also defined on the same set of semantic elements (Sect. 5). This enables us to compare and match the the SPARQL queries with that of the changes made to the ontology and hence determine which queries have been effected (Sect. 6). We present an implementation of our technique, which seamlessly integrates with a popular, openly available OWL ontology development environment (Sect. 7). We show an evaluation of our technique for a real-life application for the oil industry and two publicly available OWL benchmark applications (Sect. 8). Finally we describe some related work (Sect. 9) and discussions and conclusions (Sect. 10).

2 Preliminaries

2.1 OWL

The OWL specification is organized into the following three sections [16]:

1. **Abstract Syntax:** In this section, the modeling features of the language are presented using an abstract (non-RDF) syntax.
2. **RDF Mapping:** This section of the specification defines how the constructs in the abstract syntax are mapped into RDF triples. Rules are provided that map valid OWL ontologies to a certain sub-set of the universe of RDF graphs. Thus RDF mappings define a subset of all RDF graphs called well-formed graphs (WF_{OWL}) to represent valid OWL ontologies.
3. **Semantics:** The semantics of the language is presented in a model-theoretic form. The OWL-DL vocabulary is defined over an *OWL Universe* given by the three tuple $\langle$IOT, IOC, ALLPROP$\rangle$, where:
   (a) IOT is the set of all *owl:Thing*s, which defines the set of all individuals.
   (b) IOC is the set of all *owl:Class*es, comprises all classes of the universe.
   (c) ALLPROP is the union of set of *owl:ObjectProperty* (IOOP), *owl:Datatype Property* (IODP), *owl: AnnotationProperty* (IOAP) and *OWL:Ontology Property* (IOXP).

   In addition the following notations are defined in the specification, which we will use in the rest of this paper:
   - A mapping function T is defined to map the elements from OWL universe to RDF format.
   - An interpretation function EXT_I: ALLPROP $\rightarrow \mathrm{P}(\mathrm{R}_I \times \mathrm{R}_I)$ is used to define the semantics of the properties.
   - The notation CEXT_I is used for a mapping from IOC to $\mathrm{P}(\mathrm{R}_I)$ defining the extension of a class C from IOC.

From now on we will use *ontology* to refer to the TBox, ABox combine as commonly used in OWL terminology.

2.2 SPARQL

SPARQL is the language recommended by the W3C consortium, to query RDF graphs [20]. The language is based on the idea of matching *graph patterns*. We use the following inductive definition of *graph-pattern* [17]

1. A tuple of form (I ∪L ∪V)×(I ∪V)×(I ∪L ∪V) is a graph pattern (also a triple pattern). Where I is the set of IRIs, L set of literals and V is set of variables.
2. If P_1 and P_2 are graph patterns then P_1 *AND* P_2, P_1 *OPT* P_2, P_1 *UNION* P_2 are graph patterns
3. If *P* is a graph pattern and R is a built-in condition then *P FILTER R* is a valid graph pattern.

The semantics of SPARQL queries are defined using a mapping function μ, which is a partial function $\mu{:}V{\rightarrow}\tau$, where *V* is the set of variables appearing in the query and τ is the triple space. For a set of such mappings Ω, the semantics of the AND ($\bowtie$), UNION and OPT ($\overleftarrow{\bowtie}$) operators are given as follows

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}^1\}$$

$$\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$$
$$\Omega_1 \overleftarrow{\bowtie} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \backslash \Omega_2)$$

where

$$\Omega_1 \backslash \Omega_2 = \{\mu \in \Omega_1 | \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{are not compatible}\}$$

Since SPARQL is defined over RDF graphs, its semantics with respect to OWL is not very easy to understand. In an attempt to clarify this, a subset of SPARQL that can be applied to OWL-DL ontologies is presented in SPARQL-DL [22]. The kind of queries we use in our work are the same as those presented in their work but we also consider graph patterns that SPARQL-DL does not- more specifically the authors consider only conjunctive (AND) queries, where as we consider all SPARQL operators, viz, AND, UNION, OPTIONAL and FILTER. Note that the main goal of [22] is to define a (subset of the) language that can be implemented using current reasoners, whereas the goal of our work is to be able to detect queries that effected by ontology changes.

3 Overview

In Sect. 1 we provided the intuition that a dirty query with respect to two TBoxes is one which can match some triple from an ontology consistent with either one of the TBoxes but not both. We further formalize this notion here, using:

- *O* is the original ontology.
- *C* is the set of changes applied to *O*.
- *O′* is the new ontology obtained after *C* is applied to *O*.
- *Q* is a SPARQL query. We need to determine if it is dirty or not.

[1] μ_1 and μ_2 are compatible if for a variable *x*, $(\mu_1(x) = \mu_2(x)) \vee (\mu_1(x) = \phi) \vee (\mu_2(x) = \phi)$

- WF_O is the set of RDF graphs which represent ontologies that are consistent wrt. the statements in O.
- Similarly the set of well-formed OWL graphs wrt. $O\prime$ is given by $WF_O{}'$.

The *extension of a query* is defined as follows:

Definition: The extension of a query Q containing a graph pattern GP, wrt. an OWL T-Box O (denoted $EXT_O(Q)$ or $EXT_O(GP)$), is defined as the set of all triples that match GP and are valid statements in some RDF graph from WF_O.

A more formal definition of a dirty query is given as:

Definition: A query is said to be dirty wrt. two OWL TBoxes O and O', if it matches some triple in $WF_O{}' \setminus WF_O$ or $WF_O \setminus WF_O{}'$, i.e., $EXT_O(Q) \cap (WF_O{}' \setminus WF_O \cup WF_O \setminus WF_O{}') \neq \phi$.

Thus to determine if a given query is dirty we find the extension of the given query. Apart from this, we also need to determine and compare it with the set of triples that are present in $WF_O{}' \setminus WF_O \cup WF_O \setminus WF_O{}'$. To do this we consider the changes C and determine the set of triples added, removed or modified in WF_O due to it- we call this as the *semantics of the change*. Thus our overall approach to detect dirty queries consists of the following four steps:

1. Capture ontology change: The changes made to the ontology are logged. Ideally, the change capture tool must be integrated with the ontology design tool, so that the changes are tracked in a manner that is invisible to the ontology engineer. Since many other works [14,15,8] have focused on this aspect of the problem we re-use much of their work and hence do not delve into it.
2. Determine the extension of the query.
3. Determine the semantics of change.
4. Matching: Determine if the ontology change can lead to an inconsistent result for the given queries, by matching the extension of the query with the changed semantics of the ontology.

Each of these steps is detailed in the following sections.

4 Extension of SPARQL Queries

Due to the complexity of consistency checking for DL ontologies [3,5], it is very hard to accurately determine the EXT of a query. In order to alleviate this we use a simplified function called *NEXT*, which determines the set of triples that satisfy a graph pattern by using a necessary (but not sufficient) condition for a triple to be a valid statement in an ontology K_O. From the SPARQL semantics point of view, *NEXT* can be thought of as a function that provides the range for each variable in a query, in the evaluation function Ω. The range itself is defined in terms of the semantic elements of an OWL TBox. In other words, $\Omega : Q \rightarrow T(NEXT(Q))$- where T is the function to map OWL semantic elements to triples [16]. The semantics presented in the *RDF-Compatible Model-Theoretic Semantics* section of the OWL specification has been used as the basis for defining *NEXT*. We first show how *NEXT* is defined for simple triple patterns and then generalize it to complete queries.

4.1 Triple Patterns

Queries to OWL ontologies can be classified into three types: those that only query A-Box statements (A-Box queries), those that query only T-Box statements (T-Box queries) or those that contain a mix of both (mixed queries) [22]. In the interest of space and as all the queries in the applications/benchmarks we have considered are A-Box queries, we will only present the *NEXT* values for them. A similar method to the one below can be used to create the corresponding tables for the mixed and TBox queries.

Our evaluation of *NEXT* for triple patterns in A-Box queries is based on the following observations:

1. The facts/statements in an OWL A-Box can only be of three kinds: type assertions, or identity assertions (sameAs/ differentFrom) or property values.
2. A triple pattern contains either a constant (URI/literal) or a variable in each of the subject, object and predicate position. Correspondingly, triple patterns are evaluated differently based on whether a constant or a variable occurs in the subject, property, or object position of the query.

We illustrate how *NEXT* values for triple patterns are determined through an example. Consider a triple pattern of the form `?var1 constProperty ?var2`, where (`?var1` and `?var2` are variables and `constProperty` is a URI). We know that for this triple to match any triple in the A-Box, `constProperty` must be either `rdf:type` or `owl:sameAs/ differentFrom` or some datatype/object property defined in the T-Box.

Consider the case where `constProperty` is `rdf:type`. The only valid values that can be bound to `?var2` are the URIs that are defined as a class (or a restriction) in the T-Box. In other words it belongs to the set `IOC`. The valid values of the subject (`?var2`) is the set of all valid objects in O, i.e., `IOT`, because every element in `IOT` can have a type assertion. Therefore the *NEXT(tp)* is given as $P(\mathtt{IOT} \times \{rdf{:}type\} \times \mathtt{IOC})$- the power-set of all the triples from $\{\mathtt{IOT} \times \text{rdf:type} \times \mathtt{IOC}\}$.

Note that this is a necessary but not sufficient condition because, although every triple in *NEXT* cannot be proved to be a valid statement with respect to *O* (not sufficient), but by the definition of these semantic elements, it is necessary for a triple to be in it. As a simple example to illustrate this, consider a TBox with two classes `Man` and `Woman` that are defined to be *disjoint* classes and a triple pattern `?x rdf:type ?var`. An implication of `Man` and `Woman` being specified as disjoint classes is that an individual cannot be an instance of both these classes i.e. $EXT(tp) \notin \{\langle \mathtt{aInd\ rdf{:}type\ Man}\rangle \wedge \langle \mathtt{aInd\ rdf{:}type\ Woman}\rangle\}$. However as described above the *NEXT* for the triple pattern is $P(\mathtt{IOT} \times \{rdf{:}type\} \times \mathtt{IOC})$ and does not preclude such a combination of triples from being considered in it.

Using similar analysis, we evaluate the *NEXT* values for other kinds of triple patterns as shown in Table 1.

4.2 Compound Graph Patterns

We now extend this notion to arbitrary graph patterns. Recall from Sect. 2 that a graph pattern Q is recursively defined as Q = Q1 AND Q2 || Q1 UNION Q2 || Q1 OPT Q2 ||

Table 1. NEXT values for SPARQL queries to OWL A-Boxes

Type	**Triple Pattern (TP)**	**Case**	**NEXT$_O$(TP)**
1	*?var1 ?var2 ?var3*	-	*P*(IOT $\times$Prop $\times$(IOT $\cup$LV$_I$))
2	*?var1 ?var2* `Value`	`Value` is a URI from the TBox (Class Name)	P(IOT $\times\{$rdf:type$\}$ $\times\{$Value$\}$)
		`Value` is an unknown URI	*P*(IOT $\times\{$IOOP $\cup$owl:sameAs $\cup$owl:differentFrom$\}$ $\times\{$Value$\}$)
		`Value` is a literal	*P*(IOT $\times\{$IODP$\}$ $\times$Value)
3	*?var1* `Property`*?var3*	`Property` is rdf:type	*P*(IOT $\times\{$rdf:type$\}$ $\times$(IOC))
		`Property` is owl:sameAs(differentFrom)	*P*(IOT $\times\{$owl:sameAs$\}\times$IOT)
		`Property` is object property i.e. `Property` $\subset$IOOP	$P(\bigcup_{D\in DOM_P} CEXT(D) \times\{Property\}\times \bigcup_{R\in RAN_P} CEXT(R)))$
		`Property` is Data-type property i.e. `Property` $\subset$IODP	$P(\bigcup_{D\in DOM_P} CEXT(D)$ $\times\{$`Property`$\}$ $\times$LV)
4	*?var1* `Property` `Value`	`Property` is rdf:type (and Value $\subset$IOC)	*P*(CEXT(C)) $\times\{$rdf:type$\}$ $\times$Value) C = T^{-1}(Value)
		`Property` is sameAs/differentFrom (Value is a URI $\subset$IOT)	*P*(IOT $\times\{$owl:sameAs$\}$ $\times\{$Value$\}$)
		`Property` is a object property or data type property (Correspondingly Value is a URI or a literal)	$P(\cup_{D\in DOM_P} CEXT(D)$ $\times\{$`Property`$\}\times\{$Value$\}$)
5	*Value ?var1 ?var2*	-	*P*($\{$Value$\}$ $\times$ALLPROP $\times\{$IOT $\cup$LV $\cup$IOC$\}$)
6	*Value ?var1 Value2*	-	Same as case 2.
7	*Value* `Property` *?var2*	-	Same as case 3.
8	*Value* `Property` *Value2*	-	TP

Q1 FILTER R. The NEXT value for a query Q is defined based on what the connecting operator is as follows:

1. Consider a simple example of the first case in which both Q1 and Q2 are triple patterns connected through AND: *(?x type A AND ?x type B)*. For the variable *x* to satisfy the first (second) triple pattern, it has to have a value in `CEXT(A)` (`CEXT(B)`).

However, due to the AND, *x* has to be a compatible mapping. Thus the valid values of *x* are in (`CEXT(A)`$\cap$`CEXT(B)`). For a variable that only appears in either of the sub-patterns, the *NEXT* does not depend on the other sub-pattern.
2. For a UNION query, the mappings of the variables occurring in Q1 and Q2 are completely independent of each other and thus the evaluation can be independently performed.
3. If the two sub-queries are connected by OPT, which has the left join semantics, the variables on the left side (Q1) is independent of variables in Q2 . However the extension of the variables in Q2, is similar to the queries in an AND query.
4. When expressions are connected using the FILTER operator, the extension is determined as that of Q1 (we examine two special cases later).

Once the *NEXT* values of the variables in each sub-query are computed, the *NEXT* values of the query can be computed as follows:
For a constant c, *NEXT(c,Q)* = $\{c\}$. The extension of the query *Q* is given as

$$NEXT(Q){=}\bigcup_{tp\in Q} P(\{NEXT(sub_{tp},Q){\times}NEXT(prop_{tp},Q)\times NEXT(obj_{tp},Q)\})$$

where *tp* is each triple pattern in *Q* and sub_{tp}, $prop_{tp}$ and obj_{tp} represent the constant/variable in the respective position in *tp*.

This procedure is summarized in algorithm 4.2. The algorithm takes as input a SPARQL query that is fully parenthesized, such that the inner most parenthesis contains the expression that is to be evaluated next. For each of the expressions surrounded by a parenthesis, we maintain the value of the *NEXT* value to which the variable is mapped. When this is modified during the evaluation of the expression in a different sub-query, it is updated to be the new value of the variable, based on the operator semantics described above.

Exceptions: Two exception cases which are treated separately are:

– An interesting use of the FILTER expression is used to express negation in queries [21]. E.g., to query for the complement of instances of a class C one can write a query of the form:

  (?x type owl:Thing.OPT(?a type C.Filter(?x = ?a)).Filter(!Bound(a))

  In this query *?x* is bound to all objects that are *not* of type *C* i.e., the *NEXT* value for the variable *?x* should be assigned as `IOT` $\backslash$`CEXT(C)`.
– Another interesting case is the use of *isLiteral* condition in a FILTER expression. Consider the triple pattern *?c type Student. ?c ?p ?val.FILTER(isLiteral(val))*. Without the FILTER clause, we might conclude that the variable *p* is bound to all properties with domain Student. But since the filter condition specifies that *val* has to be a literal, *p* can be restricted to the set of data-type properties with domain C. Note that by not considering the FILTER we obtained the super-set of possible bindings. Therefore any change to one of these properties would have still been detected but some false positives may have been present.

Algorithm 1. Algorithm to compute the NEXT of a compound query

```
Require: Fully Parenthesized Query in Normal form Q, Ontology O
Ensure: NEXT of Q
 1. while all patterns are not evaluated do
 2.   P←innermost unevaluated expression in Q
 3.   if P is a triple pattern(tp) then
 4.     NEXT(var, P) ←$NEXT_S$(var, tp)
 5.     NEXT(var, tp) ←$NEXT_S$(var, tp)
 6.   else if P is of the form ($P_1$ AND $P_2$) then
 7.     for each variable v in $P_1$,$P_2$ do
 8.       if if v occurs in both $P_1$ and $P_2$ then
 9.         NEXT(v, P) = NEXT(v,$P_1$) ∩NEXT(v,$P_2$)
10.         Update the NEXT of v in $P_1$ and $P_2$ as well all sub-patterns it may occur in to
            NEXT(v, P)
11.       else if if v occurs in both $P_1$ and $P_2$ then
12.         NEXT(v, P) = NEXT(v,$P_i$)
13.       end if
14.     end for
15.   else if P is of the form ($P_1$ OPT $P_2$) then
16.     for each variable v in $P_1$,$P_2$ do
17.       if v occurs in $P_1$ then
18.         NEXT(v, P) = NEXT(v,$P_1$)
19.       else if v occurs in $P_2$ then
20.         if v also occurs in $P_1$ then
21.           NEXT(v, $P_2$) = NEXT(v, $P_1$) ∩NEXT(v,$P_2$)
22.         else if v occurs only in $P_2$ then
23.           NEXT(v, P) = NEXT(v,$P_2$)
24.         end if
25.       end if
26.     end for
27.   else if P is of the form ($P_1$ FILTER R) then
28.     for each variable v in $P_1$,$P_2$ do
29.       NEXT(v, P) = NEXT(v, $P_1$)
30.     end for
31.   else if P is of the form ($P_1$ UNION $P_2$) then
32.     NEXT(v, P) = NEXT(v, $P_1$)
33.   end if
34. end while
35. return The union of NEXT of each triple pattern in the query
```

5 Semantics of Change

The second step of our change detection process is to map the changes made to the ontology to OWL semantic elements, which will enable the queries and the changes to be compared. We observe that the changes to a TBox can be classified as *lexical changes* and *semantic changes*. Lexical changes represent the changes made to the names(URIs) of OWL classes or properties. Such changes can be handled easily by a simple string match and replace in the query.

Semantic changes are more interesting because they effect one or more OWL semantic elements and need to be carefully considered. They can be further classified as:

- *Extensional changes*: Extensional changes are the changes that modify the extensional sets of a class or a property. E.g., adding an axiom that specifies a class as a super-class of another is an example of this because, the extension of the super-class is now changed to include the instances of the sub-class.
- *Assertional/rule changes*: Assertional changes do not modify the extensions of TBox elements but add additional inference rules or assertions. E.g., specifying a property to be transitive does not change the extension of the domain or range of the property but adds a rule to derive additional triples from asserted ones.
- *Cardinality changes*. The cardinality changes specify constraints on the cardinality of the relationship.

The complete list of semantic changes that can be made to an OWL (Lite) ontology is presented in [8]. We have used it as the basis of capturing and representing the ontology changes in our system. The *semantics of a change*, is the effect of the change to extension of the model is represented as a set of all OWL semantic elements that are effected by the change. By matching this to the extension (*NEXT* value) of the query, we can determine if the query is dirty or not. In table 2, we show some examples of the changes and their semantics.

Table 2. Changes to OWL ontologies and their semantics

Object	**Operation**	**Argument(s)**	**Semantics of Change**
Ontology	Add_Class	Class definition (C)	IOC $\neq$IOC$'$
Ontology	Remove_Class	Class ID (C)	IOC $\neq$IOC$'$, CEXT(SC) $\neq$CEXT$'$(SC) CEXT(Dom(P)) $\neq$CEXT$'$(Dom(P)), CEXT$\neq$CEXT$'$(Ran(P))$\forall$P$\|$ C $\in$Dom(P) or Ran(P)
Class (C)	Add_SuperClass	Class ID (SC)	CEXT(SC) $\neq$CEXT$'$(SC)
Class(C)	Remove_SuperClass	Class ID (SC)	CEXT(SC) $\neq$CEXT$'$(SC)
Property (P)	Set_Transitivity	Property ID	- (Assertional Change)
Property (P)	UnSet_Transitivity	Property ID	- (Assertional Change)

- Example 1: A class C is added to the TBox- `IOC` [2] the set of classes defined in the TBox of the new ontology is different from the original one.
- Example 2: A more interesting case is when a class C is removed from the TBox. Not only is `IOC` changed as before, but also the extension of the super-classes of C because all the instances of C which were also instances of the super-class(es) in the original TBox are not valid in the modified TBox. The modification also effects the extensions of the classes (restrictions), *intersectionOf* in which the class C appears.

[2] The OWL spec [16] defines IOC as the set of all OWL classes, here we (ab)use the notation to denote the set of classes defined in the ontology (T-Box).

Finally, the domain and range resp. of the properties in which C appears are also modified. Note that a complete DL reasoner (like Pellet or Racer) can be used to fully derive the class subsumption hierarchy, which can then be used to derive the semantics of the change.

– Example 3: In the third example, an OWL axiom which defines a class C as subclass of SC is added. In this case, the extension of the class SC changes. Setting and un-setting transitivity of a property is an example of an assertional change to the ontology as described above.

In the interest of space the entire set of changes and the OWL semantic entities that it changes are not presented here but the interested reader can find it online[3].

6 Matching

The matching algorithm is fairly straight forward and is presented in pseudo-code form in Algorithm 2.

Algorithm 2. Algorithm to detect dirty queries for a set of ontology changes

```
Require: Ontology Change log L, Query Q
Ensure: Dirty queries
 1. Aggregate changes in L
 2. for each lexical change l in L do
 3.    Modify the name of the ontology entity if it appears in it
 4. end for
 5. Let N ←NEXT of Q
 6. Let C ←set of semantically changed extensions of O due to L
 7. for each element n in N do
 8.    if Check if n matches any element in C then
 9.       Mark Q as dirty
10.    end if
11. end for
12. return
```

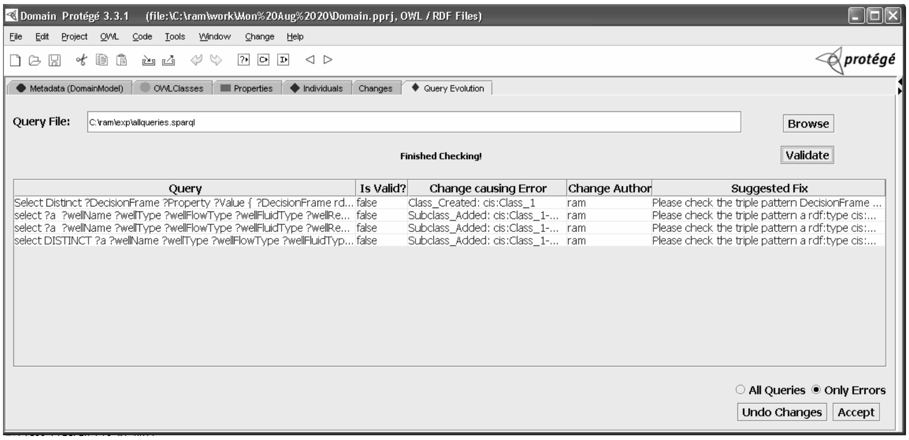
In the first step the log entries are aggregated to eliminate redundant edits. E.g., it is possible that the log contains two entries, one which deletes a class C and another which adds the same class C. Such changes are commonly observed when the changes are tracked through a user interface and the user often retraces some of the changes made. Clearly these need to be aggregated to conclude that the TBox has not been modified. Then the lexical changes are matched and the query is automatically modified to refer to the new names of the TBox elements. Finally the *NEXT* value of *Q* and the semantic implications of each change in *L* are matched. This is done by comparing the extension or element bound to the subject, object, property position of each triple pattern of *Q*, with extensions modified due to the changes made to the TBox. If any of these sets is effected, then the query is marked as dirty.

[3] http://pgroup.usc.edu/iam/papers/supplemental-material/SemanticsOfChange.pdf

7 Implementation

Our technique has been implemented as a plug-in to the popular and openly available Protégé ontology management tool [19]. Since we have used Protégé for ontology development in our work, providing this service as a plug-in enables a seamless environment to the ontology engineer. Moreover, the Protégé toolkit is equipped with another plug-in that tracks the changes made to an ontology. We utilise this to capture the changes made to the ontology. After the user makes the changes to the ontology in the design tab of the tool, he proceeds to the dirty query detection panel and points to a file containing the SPARQL files for validation. The dirty queries are highlighted and the user can then decide if the changes have to be kept or discarded. The query validation plug-in is shown in Fig. 2(a) and the Protégé change tracking plug-in in Fig. 2(b).

Our implementation of the dirty query detection algorithm is in Java and uses a openly available grammar for SPARQL to create a parser for the queries[4]. Since all


Domain Protégé 3.3.1 (file:\C:\ram\work\Mon%20Aug%2020\Domain.pprj, OWL / RDF Files)
File Edit Project OWL Code Tools Window Change Help
protégé
Metadata (DomainModel) OWLClasses Properties Individuals Changes Query Evolution
Browse
Validate
All Queries
Only Errors
Undo Changes
Accept

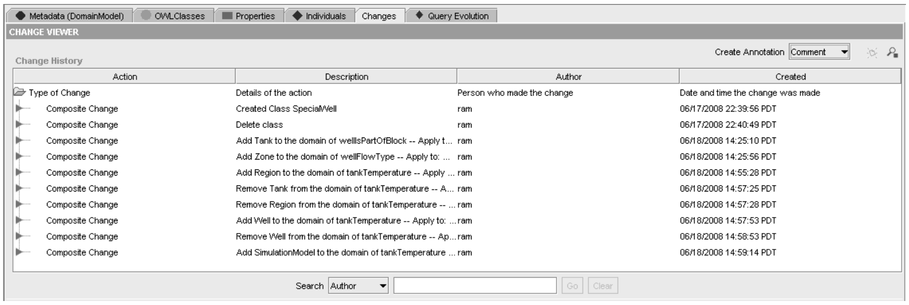
Query File: C:\ram\exp\allqueries.sparql

Finished Checking!

Query	Is Valid?	Change causing Error	Change Author	Suggested Fix
Select Distinct ?DecisionFrame ?Property ?Value { ?DecisionFrame rd...	false	Class_Created: cis:Class_1	ram	Please check the triple pattern DecisionFrame ...
select ?a ?wellName ?wellType ?wellFlowType ?wellFluidType ?wellRe...	false	Subclass_Added: cis:Class_1-...	ram	Please check the triple pattern a rdf:type cis:...
select ?a ?wellName ?wellType ?wellFlowType ?wellFluidType ?wellRe...	false	Subclass_Added: cis:Class_1-...	ram	Please check the triple pattern a rdf:type cis:...
select DISTINCT ?a ?wellName ?wellType ?wellFlowType ?wellFluidTyp...	false	Subclass_Added: cis:Class_1-...	ram	Please check the triple pattern a rdf:type cis:...

○ All Queries ● Only Errors

(a) Query validation service implemented as a plug-in to the Protégé toolkit


Metadata (DomainModel) OWLClasses Properties Individuals Changes Query Evolution
CHANGE VIEWER
Change History
Create Annotation Comment
Search Author
Go
Clear

Action	Description	Author	Created
Type of Change	Details of the action	Person who made the change	Date and time the change was made
Composite Change	Created Class SpecialWell	ram	06/17/2008 22:39:56 PDT
Composite Change	Delete class	ram	06/17/2008 22:40:49 PDT
Composite Change	Add Tank to the domain of wellIsPartOfBlock -- Apply t...	ram	06/18/2008 14:25:10 PDT
Composite Change	Add Zone to the domain of wellFlowType -- Apply to: ...	ram	06/18/2008 14:25:56 PDT
Composite Change	Add Region to the domain of tankTemperature -- Apply ...	ram	06/18/2008 14:55:28 PDT
Composite Change	Remove Tank from the domain of tankTemperature -- A...	ram	06/18/2008 14:57:25 PDT
Composite Change	Remove Region from the domain of tankTemperature -- ...	ram	06/18/2008 14:57:28 PDT
Composite Change	Add Well to the domain of tankTemperature -- Apply to: ...	ram	06/18/2008 14:57:53 PDT
Composite Change	Remove Well from the domain of tankTemperature -- Ap...	ram	06/18/2008 14:58:53 PDT
Composite Change	Add SimulationModel to the domain of tankTemperature ...	ram	06/18/2008 14:59:14 PDT

(b) Change tracking service in Protégé

Fig. 2. Support for incremental development in Protégé environment

[4] http://antlr.org/grammar/1200929755392/index.html

the queries in our applications were stored in a file, the problem of finding the queries was made easy. However, many of the queries were parameterized and we had to pre-process the queries to convert it to a form that was compliant with the specification. E.g., a query for a keyword search to find people with a user specified name is usually parameterized as follows "`SELECT ?persons WHERE {?persons rdf:type Person. ?persons hasName $userParm$}`". Here `$userParm$` is replaced with a dummy string literal to make it into a parseable SPARQL query.

Once the queries are parsed, the *NEXT* values for the queries are evaluated as described in Sect. 4. We have used the Pellet reasoner[5] for determining the class and property lattices needed for evaluating *NEXT*. The changes tracked by Protégé ontology plug-in are logged in a RDF file. We extract these changes using the Jena API[6], perform the necessary aggregations and evaluate the semantics of the changes. Again the Pellet reasoner is used here for computing the class lattices etc. Finally the matching is done and all the details of the dirty queries- the triple pattern in the query that is dirty, the TBox change that caused it to be invalidated, the person who made the change and a suggested fix to the problem is displayed to the user.

8 Evaluation

We have evaluated our algorithm on three data-sets: the first two, LUBM [11] and UOBM [12] are two popular OWL knowledge base benchmarks which consist of OWL ontologies related to universities and about 15 queries. The third benchmark we have used (called CiSoft) is based on a real application that we have built for an oil company [23]. The schema for LUBM is relatively simple- it has about 40 classes. Although UOBM has a similar size it ensures that all the OWL constructs are exercised in the TBox. Both these benchmarks have simple queries- each query on an average has about 3 triple patterns in it and they are all conjunctive queries. The TBox of the Cisoft benchmark is larger than the other two (about 100 classes) and the queries we have chosen from the Cisoft application is a set of about 25 queries, and each query has on an average 6 triple patterns. These queries exercise all the SPARQL connectors (AND, OPT, UNION, FILTER).

To evaluate our algorithm, we compare it with two other algorithms. The simpler of these two is the *Entity name* algorithm which checks if the name of the entities modified in the TBox occur in the triple patterns of the query by string matching. If it occurs, then it declares the query to be dirty and if not it declares it clean. The second algorithm called the *Basic Triple Pattern* is a sub-set of our *Complete* algorithm. This algorithm does not consider the connectors between the SPARQL operators i.e., it only implements the rules presented in table 1.

We have used the two standard metrics from information retrieval- *precision* and *recall* for evaluation. Recall is given as the ratio of the no. of dirty queries retrieved by the algorithm to the total no. of dirty queries in the data-set. Precision is given by the ratio of the total no. of dirty queries detected by the algorithm to the total no. of results returned by the algorithm. The results show in Table 3 are the average of 50 runs for

[5] http://pellet.owldl.com/

[6] http://jena.sourceforge.net/

Table 3. Dirty query detection results for three algorithms

Benchmark	Algorithm	Recall	Precision
Cisoft	Complete	1	1
	Basic T.P	1	0.2
	Entity name	0.4	0.45
LUBM	Complete	1	1
	Basic T.P	1	0.6
	Entity name	0.4	0.85
UOBM	Complete	1	1
	Basic T.P	1	0.7
	Entity name	0.25	0.6

each data-set; in each run a small number (< 10) of random changes to the ontology was simulated and the algorithms were then used to detect the dirty queries with respect to those changes.

We see that the Basic TP algorithm has a recall of 1 i.e., always returns all the dirty queries in the data-set but it also returns a number of false positives (low precision). Since the results returned by BTP is always a super-set of the complete algorithm- the recall is always 1. To understand why a low precision is observed for BTP (especially for the Cisoft data-set), consider a query of the form *?a rdf:type Student.?a ?prop ?value*. Since the algorithm considers each triple pattern in isolation, it infers that every valid triple in the ontology will match the second triple pattern (*?a ?prop ?value*). Therefore any ontology change will invalidate the query. However, this is incorrect because the first triple pattern ensures that only triples which refer to instances of *Student* will match the query and therefore only changes related to the OWL class *Student* will invalidate the query. The LUBM and UOBM queries do not have many triple patterns of this form, thus the precision of BTP for these data-sets is higher.

The entity name algorithm does not always pick out the dirty queries (recall < 1). The main shortcoming of this algorithm is that, it cannot detect the ontology changes that might affect values that might be bound to a variable.

9 Related Work

Much work has been done in the general area of ontology change management [8,9,24]. Most of these works deal with the semantic web applications in which ontologies are imported or built in a distributed setting. In such a setting, the main challenge is to ensure that the ontologies are kept consistent with each other. In our work, we address the problem of keeping the SPARQL queries consistent with OWL ontologies. Although, some aspects of the problem- e.g., the set of changes that can be made to an OWL ontology are the same, our key contributions are in defining the notion of dirty queries and the evaluation function which maps queries and (implications) of ontology changes onto the OWL semantic elements, which makes it possible to compare them to decide if the query is invalidated.

In [24], although the authors define evolution quite broadly as *timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artifacts*, they do not address the issue of keeping queries based on ontology definitions consistent with the new ontology. The authors do define a generic four stage change handling mechanism- (change) representation, semantics of change, propagation, and implementation, which is applicable to any artifact that depends on the ontology. Our own four step process is somewhat similar to this.

An important sub-problem in the ontology evolution problem is the change detection problem. Various approaches have proposed in literature to address this problem. Most of these address the problem setting of distributed ontology development and thus provide sophisticated mechanisms to compare and thus find the differences between two ontologies [15,18]. On the other hand we assume a more centralized setting in which we assume that the ontology engineers modify the same copy of the ontology definition file. We have used an existing plug-in developed for the Protégé toolkit [14], which tracks the changes made to the ontology.

An important artifact in a ontology based system is the knowledge base. In [25], the authors address the problem of efficiently maintaining a knowledge base when the ontology (logic program) changes. Similar to the work of view maintenance in the datalog community, the authors use the delta program to efficiently detect the data tuples that need to be added or deleted from the existing data store. This is an important piece of work addressing the needs of the class of applications that we target, and is complimentary to our work.

In the area of software engineering, the idea of agile database [1] addresses the similar problem of developing software in an environment in which the database schema is constantly evolving. The authors present various techniques and best practices to facilitate efficient development of software in such a dynamic methodology. Unlike our work, the authors however, do not address the problem of detecting the queries that are affected by the changes to the schema.

10 Discussion and Conclusions

We have addressed a problem seen in the context of OWL based application development using an iterative methodology. In such a setting as the (OWL) TBox is frequently modified, it becomes necessary to check if the queries used in the application also need to be modified. The key element of our technique is a SPARQL evaluation function that is used to map the query to OWL semantic elements. This is then matched with the semantics of the changes to the TBox to detect dirty queries. Our evaluation shows that simpler approaches might not be enough to effectively detect such queries.

Although originally intended to detect *dirty* queries we have found that our evaluation function can be used as a quick way to check if a SPARQL query is *semantically incorrect* with respect to an ontology. Semantically incorrect queries are those that do not match any valid graph for the ontology- i.e., always return an empty result-set. For such queries, our evaluation function will not find a satisfactory binding for all the triple patterns.

An assumption we make in our work is that all the queries to the A-Box are available for checking when changes are made. In many application development scenarios this may not be feasible. Therefore whenever possible it is a good practice for an application development team working in such an agile methodology to structure the application so that the queries used in the application can be easily extracted for these kinds of analysis. If it is not possible to do so, one option for an ontology engineer is to use the OWL built-in mechanism to mark the changed entity as *deprecated*, and phase it out after a sufficiently long time.

Often times, the queries are dynamically generated based on some user input. In such cases it might be harder to check the validity of the queries. However, it might still be possible to detect dirty queries because, such queries are generally written as parameterized templates which are customized to the user input. If such templates are made available, it might still be possible to check if they are valid.

Acknowledgment

This research was funded by CiSoft (Center for Interactive Smart Oilfield Technologies), a Center of Research Excellence and Academic Training and a joint venture between the University of Southern California and Chevron. We are grateful to the management of CiSoft and Chevron for permission to present this work.

References

1. Ambler, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley and Sons, Chichester (2003)
2. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Computer 21(5), 61–72 (1988)
3. Donini, F.M.: Complexity of reasoning. In: Description Logic Handbook, pp. 96–136 (2007)
4. Guarino, N.: Formal ontology and information systems. In: 1st International Conference on Formal Ontologies in Information Systems, FOIS 1998 (1998)
5. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. J. of Automated Reasoning 39(3), 249–276 (2007)
6. Inmon, W.H.: Building the Data Warehouse. John Wiley and sons Inc., Chichester (2002)
7. Jorge, C., Martin, H., Miltiadis, L. (eds.): The Semantic Web. Real-world Applications from Industry. Springer, Heidelberg (2007)
8. Klein, M.: Change Management for Distributed Ontologies. Ph.D thesis, Vrije Universiteit Amsterdam (August 2004)
9. Leenheer, P.D., Mens, T.: Ontology evolution: State of the art and future directions. In: Hepp, M., Leenheer, P.D., Moor, A.D., Sure, Y. (eds.) Ontology Management. Springer, Heidelberg (2007)
10. Liang, Y.: Enabling active ontology change management within semantic web-based applications. Technical report, School of Electronics and Computer Science, University of Southampton (2006)
11. Lehigh university benchmark, `http://swat.cse.lehigh.edu/projects/lubm/`
12. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete owl ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006)

13. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River (2003)
14. Noy, N.F., Kunnatur, S., Klein, M., Musen, M.A.: Tracking changes during ontology evolution. In: Third International Conference on the Semantic Web (2004)
15. Noy, N.F., Musen, M.A.: Promptdiff: A fixed-point algorithm for comparing ontology versions. In: Eighteenth National Conference on Artificial Intelligence (AAAI),
16. Patel-Schneider, P.F., Horrocks, I.: Owl web ontology language semantics and abstract syntax, w3c recommendation, `http://www.w3.org/tr/owl-semantics/`
17. Perez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
18. Plessers, P., Troyer, O.D., Casteleyn, S.: Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the World Wide Web 5 (2007)
19. The protege ontology editor and knowledge acquisition system, `http://protege.stanford.edu/`
20. Prudhommeaux, E., Seaborne, A.: Sparql query language for rdf, w3c recommendation, `http://www.w3.org/tr/2008/rec-rdf-sparql-query-20080115/`
21. Schenk, S., Staab, S.: Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In: WWW 2008: Proceeding of the 17th international conference on World Wide Web, pp. 585–594. ACM, New York (2008)
22. Sirin, E., Parsia, B.: Sparql-dl: Sparql query for owl-dl. In: 3rd OWL Experiences and Directions Workshop (OWLED) (2007)
23. Soma, R., Bakshi, A., Prasanna, V., Sie, W.D., Bourgeois, B.: Semantic web technologies for smart oil field applications. In: 2nd SPE Intelligent Energy Conference and Exhibition (February 2008)
24. Stojanovic, L.: Methods and Tools for Ontology Evolution. Ph.D thesis, University of Karlsruhe (2004)
25. Volz, R., Staab, S., Motik, B.: Incremental maintenance of materialized ontologies. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 707–724. Springer, Heidelberg (2003)