International Journal of Web & Semantic Technology (IJWest) Vol.1, Num.4, October 2010

Detecting Ontological Conflicts in Protocols between Semantic Web Services

Priyankar Ghosh and Pallab Dasgupta

Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, India
{priyankar, pallab}@cse.iitkgp.ernet.in

Abstract. The task of verifying the compatibility between interacting web services has traditionally been limited to checking the compatibility of the interaction protocol in terms of message sequences and the type of data being exchanged. Since web services are developed largely in an uncoordinated way, different services often use independently developed ontologies for the same domain instead of adhering to a single ontology as standard. In this work we investigate the approaches that can be taken by the server to verify the possibility to reach a state with semantically inconsistent results during the execution of a protocol with a client, if the client ontology is published. Often database is used to store the actual data along with the ontologies instead of storing the actual data as a part of the ontology description. It is important to observe that at the current state of the database the semantic conflict state may not be reached even if the verification done by the server indicates the possibility of reaching a conflict state. A relational algebra based decision procedure is also developed to incorporate the current state of the client and the server databases in the overall verification procedure.

1 Introduction

Ontology is regarded as a formal specification of a (usually hierarchical) set of concepts and the relations between them. The need for developing intelligent web services that can automatically interact with other web services has been one of the primary forces behind recent research towards standardization of ontologies of specific domains of interest [1, 2, 3, 4, 5]. For example, if several online book stores follow the same ontology for the *book* domain, then it facilitates an intelligent web service to automatically search these book stores to find books in a particular category.

In the context of next generation of web, it is envisaged that intelligent agents will find, combine, and act upon information on the web, thereby perform the routine day-to-day jobs independently. The protocols that will be used by such intelligent agents to communicate with the semantic web services, will play an extremely important role towards materializing the next generation of web. The protocol may contain branches which are decisions made on the basis of the previous information exchange. Along with defining the information exchange between the client and server in the form of a set query-answer, independent actions will be described as a part of the protocol. The action may be automatically executed or may need manual intervention for completion, but the information required to initiate the action is provided by answer of the previous queries. We present an example of such protocol in Section 2.

When two communicating web services use ontologies, with respect to semantic conflict the following scenarios are possible.

Scenario-1 : If the web services choose to use the same ontology, there will be no semantic conflict. In this paper we observe that the requirement that the ontologies used by communicating web services must match is a very strong requirement which is often not needed in practice.

Scenario-2 : If two communicating web services use different ontologies, then they may potentially reach a state where there is a semantic conflict/mismatch arising out of the differences between their ontologies. For example, suppose the ontologies of web service A and web service B recognize the class *vehicle* and its sub-classes, namely, *car*, *truck* and *bike*. The ontology of

DOI : 10.5121/ijwest.2010.1403

A defines *color* as an attribute of class *vehicle*, where as the ontology of B defines *color* as an attribute of the sub-classes *car* and *bike* only. Now suppose A wants to follow the following protocol with B:
Step-1: Ask B for the registration number of a vehicle which is owned by a given person.
Step-2: If B finds the registration number, then ask B for the color of the vehicle.
Several executions of this protocol are possible for different valuations of the data exchanged by the protocol. Semantic conflicts arising out of the differences in ontologies may occur in some of these cases, but not always. For example:

- If B does not find the registration number, then Step-2 is not executed and there is no semantic conflict.
- If B finds the registration number and the vehicle happens to be a truck, then Step-2 of the protocol will lead to a semantic conflict, since in B's ontology, the *color* attribute is not defined for trucks.
- If B finds the registration number and the vehicle happens to be a car or a bike, then Step-2 will not lead to a semantic conflict, since in B's ontology, the *color* attribute is defined for cars and bikes.

If the ontology of A and the protocol is made available to B, then B can formally verify whether any execution of the protocol may lead to a semantic conflict and warn A accordingly before the actual execution of the protocol begins.
There has been considerable research in the recent past on matching ontologies and finding out semantic conflicts/mismatches among two ontologies [6, 7, 8]. In many cases, two web services may have conflicting ontologies, but the protocol between them may avoid the conflict scenarios. Consider the scenario where the direction of query-answer is reversed, that is the same sequence of queries are made by A and answered by B. Also A makes the query about the color of vehicle only if the vehicle is not a truck. In this case the conflict will not be sensitized by the protocol. In other words, two agents may not agree on all concepts in their universe, but may still be able to support certain protocols as long as they avoid the contentious issues – a fact which is often ignored in world politics! Therefore an approach which rules out communication between two services on the grounds that their ontologies do not match is too conservative in practice. Since the standardization of ontologies and their acceptance in industrial practice seems to be a distant possibility, we believe that the verification problem presented in this paper and its solution is very relevant at present.

Scenario-3 : The ontologies can be visualized as a combination of meta-data and a set of instances. Classes, relations and data-types form the meta-data part of the ontology, whereas the individuals and the valuations of the attributes are the actual data. It is often the case that the actual data is stored in a database, and ontologies are used as a wrapper on top of the databases. Therefore the state of the database has to be incorporated, while the server checks whether the protocol can possibly reach conflict state. Since the protocol between the client and the server typically have branches and the decision for making the next query is dependent on the answer of the current query, the conflict that is present at the ontology level may not be sensitized due to the the answers generated from the back-end database. We present a relation algebra based decision procedure to check whether the conflict, that are present in the ontology level, are actually present with respect to the current state of the back-end database.

Scenario-4 : It is important to observe that the protocol has different runs depending on the instantiation of the variables that are used in the protocol. Since the conflict may not be sensitized in a particular run of the protocol, the server may choose to start the protocol and check the possibility to get into a conflict after every information exchange. Depending on how the conversation progresses the server may either continue to run protocol, or may terminate the conversation when it finds that the conflict is inevitable.

A preliminary version of this work is published in [9]. In that version we presented the verification algorithm for Scenario-2. In this paper we include the algorithms for Scenario-3, i.e. the verification of the spuriousness of an ontological conflict with respect to the current state of the back-end database. We also show that the same algorithm can be used by the server for Scenario-4. The

paper is organized as follows. The syntax for describing a protocol is described in Section 2. In Section 3 we present a graph based model for representing the ontologies. The proposed formal method for detecting semantic conflicts at the ontology level is presented in Section 4. The notion of ontology with database and query answering with the back-end database and the algorithm to verify the conflicts at the ontology level in the presence of the database are presented in Section 5. Related works are briefly discussed in Section 6. Finally we present the conclusion in Section 7.

2 Protocol and Conflict

In this section we present a formalism similar to SQL for the specification of the protocol. It may be noted that other formalisms can also be used to specify a protocol as long as the formalism has expressive power similar to the formalism used in this paper. We present two example protocols and also describe the notion of the conflict that we have addressed in this paper.

2.1 Formal Description of the Protocol

Typically, a protocol consists of a sequence of queries and answers. The query specifies a set of variables through *"Get"* keyword and specifies a set of classes using *"from"* keyword. The valuations corresponding to the variable set are generated from those classes. Also an optional *"where"* keyword is used to specify the conditions on the variables. The answer of a query is a tuple of valuations corresponding to the variable set specified in the query. The branching is specified using *"if-else"* statements.

2.2 Example of Protocol

[Protocol - 1 :] Consider the protocol shown in Figure 1. The protocol depicts a conversation between a client and a server over the publication domain. The query of the client is about the author of some specific manual. Then the client makes a query to retrieve a book by the author of that manual. According to the ontology of the client, *'Proceedings'* is a subclass of *'Book'* and the client makes the next query to retrieve the proceedings by the same author. If the server does not recognize *'Proceedings'* as a sub class of *'Book'*, the query can not be answered by the server due to the mismatch in the ontologies.

[Protocol - 2 :] In Figure 2 we present another protocol that exchanges information about the automobile domain. The client makes a query to retrieve a brand which has sold more than a specific number of vehicles in a particular year. Then next query is made in the context of the previous query to check whether that brand manufacture *'Red Trucks'*. According to the ontology of the client the color is a property of the vehicle class and therefore all subclasses of vehicle class will have the color attribute. However if the server recognizes *'color'* as an attribute of some of the sub-classes(suppose *'car'* and *'two-wheeler'*) instead of as an attribute of the class *'Vehicle'* itself, the query can not be answered by the server due to the mismatch in the ontology.

[Protocol - 3 :] In this example we present a protocol of an *intelligent agent*. Consider the semantic web service for an online store. The online store can queried to retrieve the relevant information about the available items. Also consider a multi-cuisine restaurant which is a client of that store. Whenever the stock of some item, say i_1, falls below some level, the intelligent agent that works on behalf of the restaurant, searches the availability of i_1 by querying the online store. Suppose i_1 comes in two qualities, q_1 and q_2. The protocol, that is used by that agent to find and buy the item under consideration, is presented below using a format similar to pseudo code. Here the *buy* action is carried out by the agent automatically, if the precondition is satisfied.

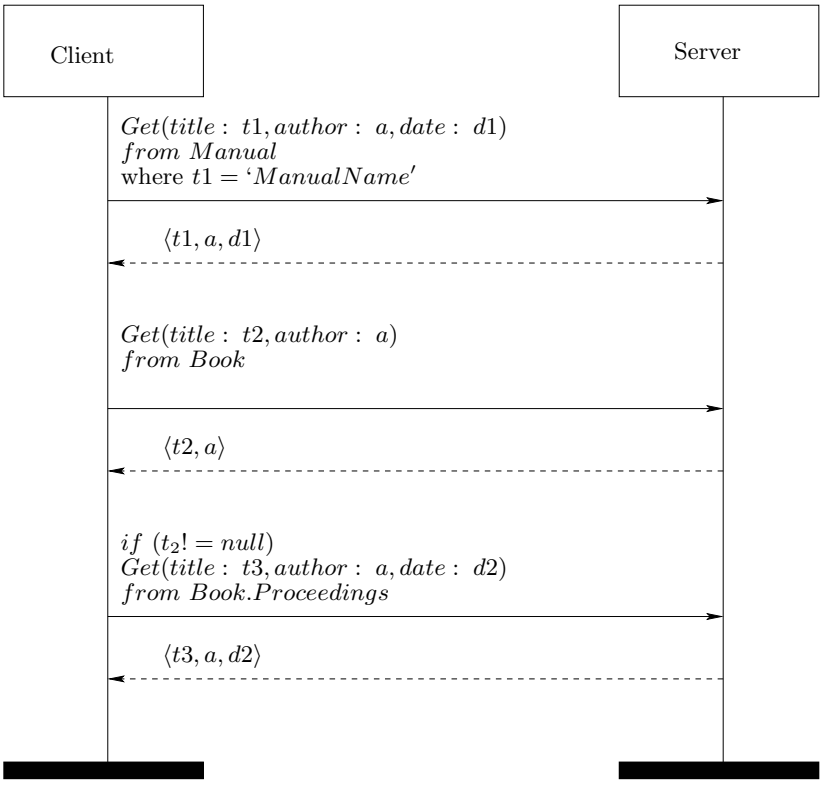

Fig. 1. Protocol on Publication Domain

Protocol for Buying an Item

Get the availability i_1 of quality q_1;
If (i_1 of quality q_1 is available)
    Get the price of i_1 of quality q_1;
    If (the price is less than C_1)
        Buy i_1 of quantity Q_1;
    Else
        Inform the Manager of the store;
Else
    Get the price of i_1 of quality q_2;
    If (the price is less than C_2)
        Buy i_1 of quantity Q_2;
    Else
        Inform the Manager of the store;

2.3 Notion of Mismatch between two Ontologies

We focus on the following two types of mismatch between the client and server ontologies in this paper.

Specialization Mismatch(Type-1): In this type of incompatibility the client recognizes a class c_2 as the specialization of another class c_1 whereas the server recognizes c_2 as the specialization of some other class c_1'. Our first example (Figure 1) is an instance of this type.

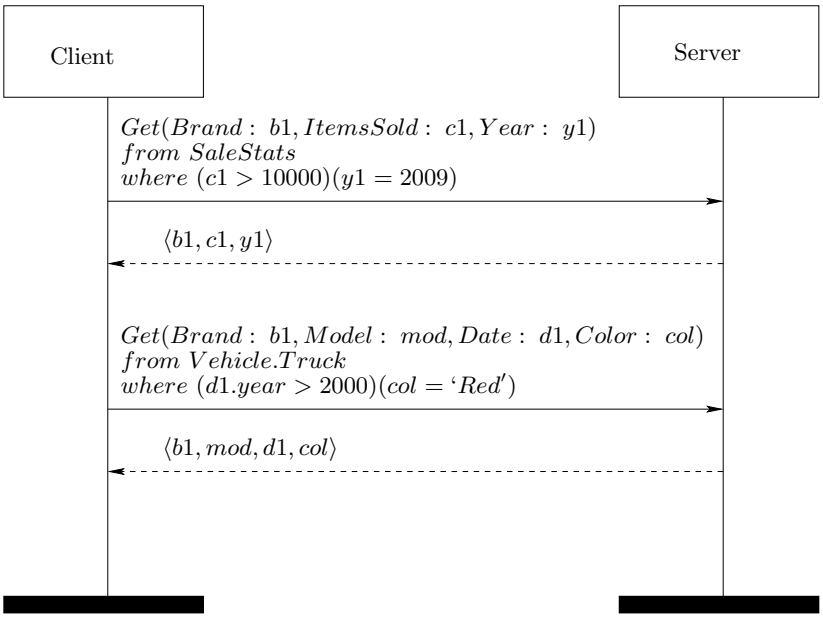

Fig. 2. Protocol on Automobile Domain

Attribute Assignment Mismatch(Type-2): A very common type of incompatibility arises where the client and the server both recognize classes $c'_1, \ldots, c'_n$ as the specializations of another class c_1, but the client associates an attribute α with the super class c_1, whereas the server associates α with some of the sub classes $c'_i, \ldots, c'_j$, $0 < i, j \leq n$. Since we view the mismatches from the query answering perspective, we use the notion of this conflict from the query perspective. If the set of variables that is used in a query q, is not available at server side, we denote that as *attribute level(Type-2)* mismatch. Our first example (Figure 2) is an instance of this type.

3 Graph Model of Ontology

While describing an ontology using OWL, the class and the attributes(modeled as properties in the context of OWL) are used to represent the meta-data. We use a graph based approach to model the meta-data that are described as classes and attributes in OWL. While using OWL, the *properties* are used to express the attributes. Therefore we use the term property and attribute interchangeably. We define the ontology graph as follows.

Definition 1. *A **graph model** for an ontology O is $\mathcal{G} = (V, E)$ where, V is the set of vertices and E is the set of directed edges. Each node $v_i \in V$ represents a class in the OWL ontology and v_i is associated with a **property list** $\mathcal{L}(v_i)$ whose elements are the data properties of the class. The directed edges can be of the following types*

Inheritance-Edge : *An* inheritance-edge $e_{ij} \in E$ *from* v_i *to* v_j*, where* $v_i, v_j \in V$*, if* v_j *is a sub class of* v_i*.*

Property-Edge : *An* property-edge $e_{ij} \in E$ *from* v_i *to* v_j*, where* $v_i, v_j \in V$*, if* v_j *is an object property of* v_i*.*

4 Overview of the Method

In this section we present the relevant formalisms and present the overall algorithm for solving the problem. The *variable set* and the *class set* specified in the query q are denoted by $S_v(q)$ and $S_c(q)$

respectively. We present a graph search based structural matching algorithm to check the semantic safety of the protocol.

Definition 2. *The* specialization sequence $\sigma = \langle c_1.c_2. \cdots .c_k \rangle$ *in a query q is the sequence of classes that are concatenated through the '.' operator, and for any two consecutive classes* c_i *and* c_{i+1} *in the sequence,* c_i *is the super class of* c_{i+1}*. Therefore the elements of* $S_c(q)$ *can be individual classes or specification sequences.*

4.1 Structural Algorithm to Check the Semantic Consistency

Algorithm 1: Check-Consistency

input : The Protocol $\mathcal{P}$ and the Server Ontology $\mathcal{O}_s$

$V \leftarrow \{\}$;
foreach *query q in the protocol* $\mathcal{P}$ **do**
  foreach *element* τ *in* $S_c(q)$ **do**
    if τ *is a specialization sequence* **then**
      $c_1 \leftarrow$ the first concept of τ;
      $c_t \leftarrow$ FindMatch($\mathcal{O}_s$, c_1);
      for $i \leftarrow 2$ **to** $length(\tau)$ **do**
        $c_m \leftarrow$ the i^{th} concept of τ;
        if *any class* c'_t *equivalent to* c_m *is not found as a sub class of* c_t *in* $\mathcal{O}_s$ **then**
          Report Mismatch at c_m;
        else
          $c_t \leftarrow c'_t$
        end
      end
      $V \leftarrow V \cup$ property set for c_t;
    else
      /* c is an individual class */
      $c_1 \leftarrow \tau$;
      $c_t \leftarrow$ FindMatch($\mathcal{O}_s$, c_1);
      $V \leftarrow V \cup$ property set for c_t;
    end
  end
  if $S_v(q) \subsetneq V$ **then**
    Report $\{S_v(q) - V\}$ as unmatched variables;
  end
end

Function `FindMatch(`$\mathcal{O}_s$, c_i`)`

Find the class c_t which is equivalent to c_i in $\mathcal{O}_s$;
if c_t *is not found in* $\mathcal{O}_s$ **then**
  Report Mismatch at c_i;
  exit;
end
return c_i;

4.2 Working Example

We present a working example to describe how the algorithm works. Consider the protocol shown in Figure 1. We elaborate the steps of applying Algorithm 1 with respect to the fragments of the

client and server ontologies shown in Figure 3 and Figure 4 respectively. These fragments are taken from the benchmark provided by [10]. The benchmark has one reference ontology and four other real ontologies and the domain of these ontologies is bibliographic references. We have used the reference ontology as the server ontology and another real ontology named INRIA as the client ontology. We have used a pictorial representation which is similar to entity-relationship diagram to show the fragments of the ontologies. The classes are represented by the rounded rectangles and the ovals represent the properties of a particular class. The class hierarchy is shown using arrows, that is a sub class is connected to its super class by an arrow which is directed towards the sub class. The properties that belong to a particular class are connected to the rounded rectangle corresponding to that class through a line.

Step-1: While applying Algorithm 1 to the server ontology, the individual class *'Manual'* is searched and since the search is successful, it is checked that the attributes that are associated with class *'Manual'* in the query in the protocol are actually answerable by the server and this check turns out to be successful for the ontologies that are presented here.

Step-2: The next query uses the class *'Book'*. Algorithm 1 performs the consistency checking in the way that is similar to the previous query and the check is successful.

Step-3: The third query uses a specialization sequence *'Book.Proceedings'*. Algorithm 1 searches for the *'Book'* class in the server ontology and then checks whether *'Proceedings'* is a sub class of *'Book'* in the server ontology. Algorithm 1 reports a failure since in the server ontology *'Proceedings'* is not a sub class of *'Book'*.

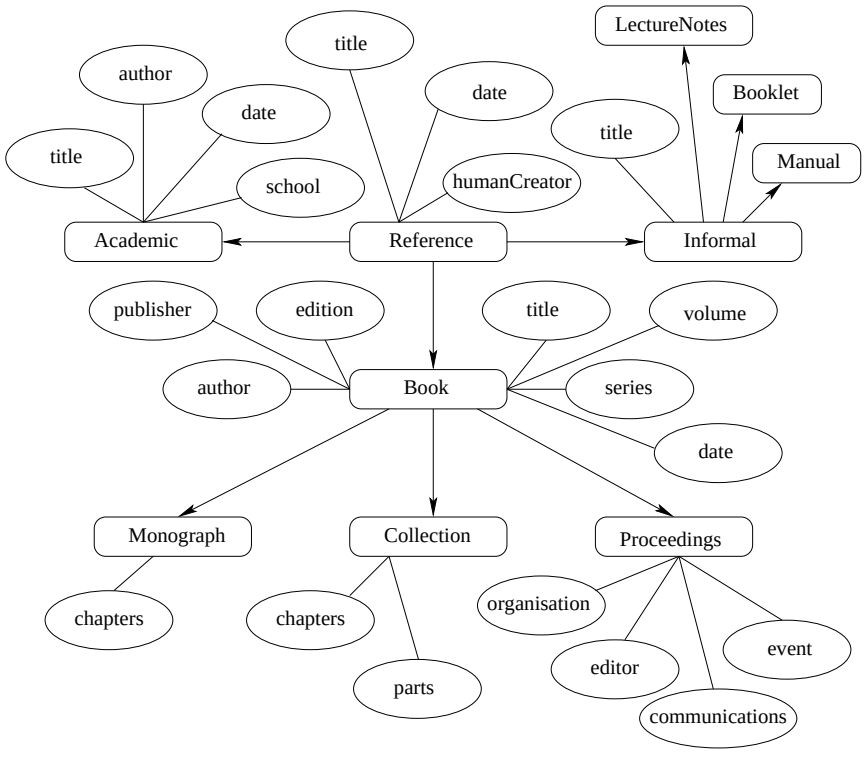

Fig. 3. Fragment of Client Ontology

4.3 Proof of Correctness

Theorem 1. *[Soundness] The mismatches returned by Algorithm 1 are correct.*

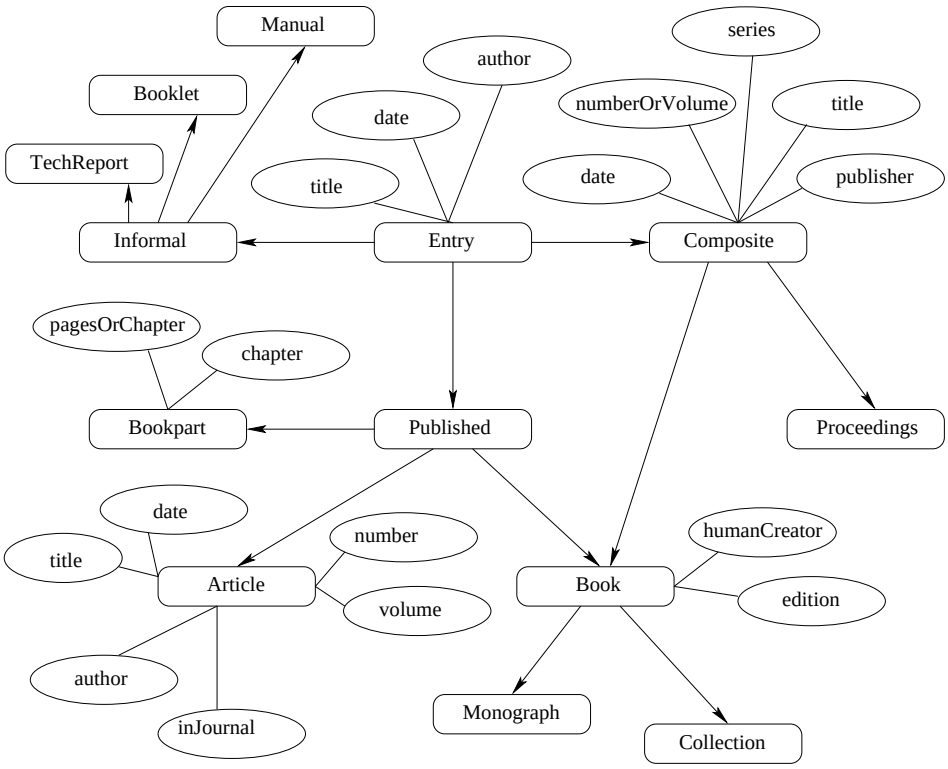

Fig. 4. Fragment of Server Ontology

Proof. Algorithm 1 reports mismatch in three cases. We observe each of the cases as follows.

Mismatch in individual class: If Algorithm 1 does not find a matching class c which is used in a query, a conflict is reported. Since the class is not recognized by the server, it is not possible for the server to answer the query. Therefore the outcome of the algorithm is correct.

Mismatch in specialization sequence: Consider a specialization sequence $\sigma = \langle c_1.c_2. \cdots .c_k \rangle$ in a query q on which Algorithm 1 returns a mismatch. We prove the correctness of the consistency checking by induction on the length k of σ.

Basis($k = 1$): In this case there is only one class in the specialization sequence and this case falls under the case of mismatch in individual classes.

Inductive Step: Suppose Algorithm 1 returns the mismatch correctly for specialization sequences having length k. We prove that Algorithm 1 reports the conflicts correctly for the specialization sequences having length $k + 1$. There can be two possible cases.

a. The conflict is reported for a class that appears in the i^{th} location of the sequence, where $1 < i < k + 1$. The reported mismatch is correct according to the inductive hypothesis.
b. The conflict is reported for the $k + 1^{th}$ class of the sequence. In this case there exists a matching specialization sequence at server ontology up to length k. But c_{k+1} is not a sub class of class c_k according to the server ontology. Therefore the conflict reported by Algorithm 1 is correct.

Mismatch on variables: Suppose the set of variables that are specified by the client is V_c in a query q corresponding to the class set $S_c(q)$ and the failure is reported on some variable in V_c. Since Algorithm 1 first finds the matches corresponding to the classes in $S_c(q)$ and then checks for the answerability with respect to the variable set, in this case every class in $S_c(q)$ is matched with suitable classes in the server side. Now Algorithm 1 reports conflict if there exists any variable that is not recognized by the server as an attribute of at least one of the classes that correspond to the classes in $S_c(q)$. Therefore the reported conflict falls under the *Type-2 or attribute level* conflict category. □

Theorem 2. *[Completeness] For any protocol $\mathcal{P}$, if there is any mismatch of type-1 or type-2, Algorithm 1 reports it.*

Proof. This proof is done by construction. For each of the type of the mismatches we show that Algorithm 1 uses a sequence of operations through which the mismatch is detected. We present the proof for each mismatch type.

Type-1 Mismatch: Consider a specialization sequence $\sigma = \langle c_1.c_2. \cdots .c_k \rangle$ which is used in query q. Algorithm 1 starts by finding the class that is equivalent to c_1 at the server side. If there is only one class in σ then Algorithm 1 reports mismatch when the corresponding class is not found in the server ontology. When the length of σ is greater than 1, Algorithm 1 continues to check whether c_i is a subclass of c_{i+1} where $1 < i < k$. A mismatch is reported by Algorithm 1 whenever c_i is a subclass of c_{i+1} for $1 < i < k$. Hence if there exists any mismatch in any specialization sequence, the algorithm reports it.

Type-2 Mismatch: Consider a query q made by the client and the set of variables is V_c in q. The set of classes is denoted by $S_c(q)$. We argue that, if there exists a *Type-2* mismatch for query q, Algorithm 1 reports it. For *Type-2* mismatches Algorithm 1 first checks the presence of the equivalent classes c_i^s in the server ontology and computes the union V_s of the attributes corresponding to every c_i^s. If there is any variable/s in V_c that are not present in V_s, a conflict is reported by Algorithm 1. Hence if there exists a *Type-2* mismatch for a query, Algorithm 1 reports it. □

5 Ontology with Back-end Database

In this section we describe the two level representation for describing ontologies – using OWL to describe the classification and using database to store the instances. This type of representation is helpful for describing domains with large number of instances. From the point of view of the instances of classes, the classes in an ontology can be categorized as follows.

a. Classes of Abstract Type – these classes are used for purely the purpose of describing a domain in hierarchically. These classes does not have any instances. They act only as the super class of other classes.
b. Classes with Instances – these classes may act as super class of other classes but they have a non-empty set of instances.

Consider the ontology fragment in Fig. 4. Here *Entry*, *Informal*, and *Composite* are the example of abstract classes. On the other hand, *Book*, *Monograph* etc. are the example of classes with instances. Although *Book* is a super class of *Monograph* and *Collection*, it is possible to have instances of *Book* which are neither *Monograph* nor *Collection*.

While using the two level representation, it is important to keep the database schema consistent with the wrapper ontology. A choice of describing the database schema could be maintaining a table for each of the non-abstract classes present in the ontology. Alternative ways of describing the database are possible, but we use this simplistic representation of the database schema to present the proposed algorithm.

5.1 Query Answering in the Presence of the Database

When the server side adheres the two layer structure for its ontology, every query in the protocol is answered by generating corresponding tuples from the back-end database. In the context of the back-end database the occurrences of variables in a protocol, can be categorized into the following types.

Uninstantiated: When a variable is placed in a query for the first time without initialization, it is referred to as an *uninstantiated* occurrence of variable or in short *uninstantiated variable*. The values for the variables are instantiated at the side where the query is evaluated.

Instantiated: Other than the first occurrence without initialization, all other occurrences of a variable is referred to as *instantiated* occurrence of that variable or in short *instantiated variable*. At these occurrences, the variables are already assigned to some value by the server. These occurrences are used for value propagation.

[Evaluation Semantics of a Query :] The semantics of the evaluation of the query is similar to the *Conjunctive Datalog*. The evaluator of the query tries to assign value to uninstantiated variables and forms a tuple which satisfies logical *and* of the conditions specified in the *where* clause of the query. Same variables in different classes specified in the *where* clause of the query have to be assigned to the same value.

Consider the protocol presented in Fig. 1. In Section 4.2 we have shown that the protocol has an ontological conflict, when the client and the server uses the ontologies in Fig. 3 and Fig. 4 respectively. Consider the fact, that the condition, $(t_2! = null)$ may always evaluate false due to the actual data that is stored in the database of the server. In that case, the ontological conflict in the last query, $[\mathsf{Get}(title : t3, author : a, date : d2)\ \mathsf{from}\ Book.Proceedings]$, will never be sensitized. In other words the conflicts at the ontology level may turn out to be spurious. We define the *spuriousness* of an ontological conflict as follows.

Definition 3. *An ontological conflict is* spurious*, when for all possible correct instantiations of the variables, the conflict is not reachable from the start state of the protocol, due to the decisions taken at different stages of the protocol. By* correct *instantiations we mean the instantiations that conform to the evaluation semantics defined earlier.*

5.2 Related Formalisms

Here we present the relevant formalisms for describing the algorithm to check the presence of the conflict detected by Algo. 1 at the current state of the server database.

Definition 4. *The* assignable set *of values for a variable φ is the set of values that can be assigned to φ during the instantiation and it is denoted as* AssignableSet(φ).

Suppose in a protocol $\mathcal{P}$, a query q has variable set $v = \{\varphi_1, ..., \varphi_n\}$ and concept set $C = \{C_1, ..., C_m\}$. Let us also assume that in $\mathcal{P}$ all the variables of q are *uninstantiated* variables. The notion of assignable set in the presence of the previously instantiated variables is discussed later. The evaluation of the query basically assigns a values to each of the variables in that query. All the variables together form a tuple $\tau = \langle val_1, val_2, \ldots, val_n \rangle$ such that if any variable φ_k is common between class C_i and class C_j then both the classes have to assign same value to the variable φ_k. All such possible tuples that can be populated by the evaluator side, form the assignable set of values for v and the assignable set for a variable φ_i is:

$$AssignableSet(\varphi_i) = \{val \mid \exists \tau \in AssignableSet(v) \wedge \tau = \langle val_1, val_2, ..., val_n \rangle \wedge val_i = val\}$$

The *dependencies* among the variables play an important role for determining the AssignableSet for a variable.

Definition 5. *In a query, if some of the variables are previously instantiated, we say that the previously instantiated set of variables is* constraining *the set of values of the uninstantiated variables. Suppose in the same query q, among the variables specified in q, $\varphi_1, \cdots, \varphi_k$ are previously instantiated and $\varphi_{k+1}, \cdots, \varphi_n$ are the variables that are instantiated by the evaluation of q. We define the* constrain relation $\mathcal{R}_\mathcal{C}$ *and the* ConstrainSet *as follows.*

$$\mathcal{R}_\mathcal{C} = \{(\varphi_i, \varphi_j) \mid \text{ where } \varphi_i \in \{\varphi_1, \cdots, \varphi_k\} \text{ and } \varphi_j \in \{\varphi_{k+1}, ..., \varphi_n\}\}$$
$$ConstrainSet(\varphi_i) = \{\varphi_{k+1}, \varphi_{k+2}, \cdots, \varphi_n\}$$

Consider the same query q. The AssignableSet for the set of variables of q is the set of all tuples $\tau = \langle val_1, val_2, \ldots ..., val_n \rangle$ such that the following conditions hold.

- If any variable φ_k is placed in more than one concepts, all the concepts assign same values to φ_k.
- $(val_1 \in A_1) \wedge ... \wedge (val_k \in A_k)$, where $A_1, \cdots, A_k$ are the assignable sets of variable $\varphi_1, \cdots, \varphi_k$ respectively.

Definition 6. *The* RestrictSet *for a variable set v is obtained by computing the transitive closure of the $\mathcal{R}_\mathcal{C}$ on v.*

We use the notion of the *split* operation on the assignable set of values of a variable and it works as follows. Let a query, q, consists of concept C_i with a uninstantiated variable φ_i, and a previously instantiated variable φ_j. Suppose a decision is made on the variable φ_j. In each branch, the possible values of φ_j forms a subset of its assignable set. Since the value of φ_i is dependent on φ_j, in each branch the possible values for φ_i also forms a subset of the assignable set of φ_i.

Definition 7. *The* SplitSet *for a variable set v is a subset of RestrictSet(v) and is defined as:*

$$SplitSet(v) = \{\varphi_j \,|\, \varphi_j \in RestrictSet(\varphi_i) \text{ and } \varphi_j \text{ appears in a condition in the path of the protocol from the start of the protocol to the query with ontological conflict } \varphi_i\}$$

Definition 8. RelevantConditionSet *of a variable set v is the set of conditions in true form on the variable set v_{split}, which have to be true for reaching the conflicting query.*

5.3 Algorithm for Detecting Spurious Conflicts with respect to the Back-end Databases

Algorithm 3: Verify the Conflicts on Back-end Database

```
1  Initialize a hash table H^t ;
   /* In the hash table H^t, a set of variables v forms the key, which is
      mapped to the AssignableSet of the variable set v                    */
2  foreach conflicting query q do
3      v ← The set of instantiated variables specified in q;
4      if VerifyConflict(v) then
5          Report mismatch on variable v at database level;
6      else Report the conflict as spurious;
7  end
```

Function `MakeSets(`v`)`

```
1  Initialize set of variable sets v^ret = {};
2  while v is not empty do
3      Find a query q that instantiates some of the variables in v;
4      Initialize variable set v_temp = {};
5      forall variable φ_i ∈ v and φ_i is instantiated by q do
6          v ← v − {φ_i};
7          v_temp ← v_temp ∪ {φ_i};
8      end
9      v^ret ← v^ret ∪ {v_temp};
10 end
```

Function `SplitAssignableSet(`δ, v_{split}, c`)`

```
   /* Suppose c_1, ··· , c_i ∈ c                                          */
1  Relational algebra query q^Rel ← σ_(c1∨c2∨...∨ci)(δ);
2  Compute q^Rel and return the set of tuples;
```

Function `VerifyConflict(`v`)`

```
1  v_restrict ← The RestrictSet for the variable set v;
2  v_split ← The SplitSet for the variable set v;
3  v^s_restrict ← MakeSets(v_restrict);
4  Construct a priority queue Γ of variable sets;
   /* Γ is ordered according to the order of the instantiations of its variable sets
      */
5  forall variable set v_i ∈ v^s_restrict do
6      Enqueue v_i in Γ;
7  end
8  Table set S^t ← {};
9  while Γ is not empty do
10     u ← Dequeue (Γ);
11     if (VerifyConflict(u)) then
           /* The set of possible valuations for u is not empty                  */
12         t ← Search H^t and return the table containing u ;
13         if t ∉ S^t then
14             S^t ← S^t ∪ {t};
15         end
16     else
           /* The set of possible valuations for u is empty, so the conflict is
              spurious                                                           */
17         return false;
18     end
19 end
20 Find the query q that instantiates variable set v;
21 if v_split != ∅ then
22     c ← The RelevantConditionSet on the variable set v_split;
23     δ ← SplitAssignableSet(δ, v_split, c);
24 end
25 if δ == ∅ then
26     Report the conflict on v as spurious;
27     return false;
28 else
29     Insert δ in H^t ;
30     return true;
31 end
```

Function `GenerateAssignableSet(`q, S^t`)`

```
   /* Suppose q is made with the concepts C_1, ..., C_n and φ_i1, ..., φ_ik are the
      uninstantiated variables corresponding to the concept C_i                  */
1  v ← {φ_ij | φ_ij ≠ *};
2  if S^t == Φ then
       /* All the variables of q are uninstantiated                             */
3      Tuple set T ← (C_1 ⋈ C_2 ⋈ ... ⋈ C_n);
4  else
       /* Some of the variables of q are previously instantiated and t_1, ..., t_m ∈ S^t are
          the tuple sets corresponding to those variables                        */
5      Tuple set T ← (C_1 ⋈ C_2 ⋈ ... ⋈ C_n ⋈ t_1 ⋈ ... ⋈ t_m);
6  end
7  Relational algebra query q^Rel ← π_v(T);
8  Compute q^Rel and return the set of tuples;
```

This algorithm can also be used by the server as the protocol progresses(described as Scenario-4 in Section 1). In that case, the variables in the queries which are already executed, have some value assigned to them and those variables will be considered as *instantiated* by the algorithm.

5.4 Proof of Correctness

The proof of correctness of Algo. 3 is presented below. Algo. 3 verifies the spuriousness of conflicts returned by Algo. 1 on the server database.

Theorem 3. *[Soundness] Algorithm 3 correctly reports the spuriousness of conflict on the set of variables v', where $v' = v \cup RestrictSet(v)$ and v is the set of previously instantiated variables in a query q of protocol $\mathcal{P}$ with ontological conflict.*

Proof. The proof is done using induction. We do the induction on the integer parameter n, where n is the total number of *VerifyConflict* function calls done by Algorithm 3 for q. Among the different *VerifyConflict* function calls, first call is done by Algorithm 3 and the others are recursive calls.

[Basis (n = 1) :] In this case RestrictSet$(v) = \phi$. In this case if the $AssignableSet(v)$ is $\varnothing$ Algo. 3 correctly reports the conflict as spurious, otherwise Algo. 3 reports the conflict as not spurious, which is correct.

[Inductive Step :] We assume that the spuriousness of a conflict reported for the queries with ontological conflict in n steps are true. We now prove that the spuriousness of a conflict that is reported in $(n + 1)$ steps are correct. Consider the *VerifyConflict* function call at Algo. 3 and without loss of generality, we can assume this function call as the $(n + 1)^{th}$ function call (in the order of returning of the function calls). Therefore the other calls are recursive calls done by the *VerifyConflict* to itself. The following two cases are possible.

a. The conflict may be detected as spurious by some call which is not the $(n + 1)^{th}$ call. In this case the spuriousness of the conflict is correct by the inductive hypothesis.
b. The conflict is detected as spurious at the $(n + 1)^{th}$ call to *VerifyConflict*. All other previous calls to *VerifyConflict* add a table to H^t and the set of tables are kept in S^t. After that, function GenerateAssignableSet is called to compute the assignable set for the set of previously instantiated variables v in the query q with ontological conflict. It follows from the description of the function, that this function restricts the set of valuations of v by taking the natural join with the valuations of variables in RestrictSet(v). Since the conflict is not detected as spurious in the variables in RestrictSet(v), when the function detects the conflict as spurious, the statement $\delta == \varnothing$ is true. Therefore in the protocol q is not reachable from the start state of the protocol. □

Theorem 4. *[Completeness] If there is a spurious conflict on the set of variables v', where $v' = v \cup RestrictSet(v)$ and v is the previously instantiated variable set specified in a query q of protocol $\mathcal{P}$ with ontological conflict, the algorithm reports it. We do the proof by establishing the contrapositive of the statement, i.e. Algorithm 3 reports the ontological as not spurious, if q is reachable from the start state of $\mathcal{P}$.*

Proof. Suppose $v' = \{\varphi_1, ..., \varphi_n\}$. Let the valuations of the variables in v' are $(val_1, ..., val_n)$ when the conflict in q is not spurious. In this case the conflict may occur in the following way. Consider the *VerifyConflict* function calls made to determine the spuriousness of the ontological conflict in q, among which the first call is done by Algo. 3 and the subsequent calls are recursive calls. The conflict is detected as *not* spurious, only if all the recursive calls to *VerifyConflict* add a table to H^t and the set of tables are kept in S^t. Since the conflict is determined as *not* spurious, the statement δ is not empty. Therefore in $\mathcal{P}$, q is reachable from the start state of the protocol using any instantiation of variables belonging to δ. □

6 Related Works

Different aspects of web service interaction have been an active area of research. However most of these researches consider the interaction at syntactic level. Foster *et. al.* addressed the compatibility verification of web services in [11]. They adopted a model based approach for checking the

compatibility of web services at different level of abstraction. However the semantics of exchanged data is not addressed by the researchers. In [12] researchers address the interaction among web services which is asynchronous in nature and propose a design pattern to help the development of composite web services based on asynchronous interaction. Zhao *et. al.* provides a formal treatment of web service choreography in [13]. They define a formal model of the of WS-CDL and propose a methodology to formally verify the correctness of a choreography using the model checker SPIN. In [14] authors proposed a formalism for specifying the web service interfaces. They discuss about three kind of constraints which can be put by a web service interface. The *propositional constraints* are imposed by an interface by specifying the methods that can be invoked by the clients along with the constraints on the input and output parameters(*signature constraints*). *Protocol Constraints* specify the temporal requirements on the sequence of the method invocations. An algorithm is proposed to check compatibility among the web services based on the mentioned constraints. However all the proposed verification strategies work at a syntactic level, without considering the semantics of the exchanged data.

On the other hand the current research in semantic web is focused towards the standardization of the ontology used by the web services with a vision of computers becoming capable of analyzing all web data. Semantic matchmaking [15, 1] and discovery of semantic web services [16, 17, 18] are two important research directions in semantic web. The underlying objective of these approaches is to compare facts belonging to different ontologies and to evaluate their compatibility. Standards like RDF, OWL, WSML etc. are developed for this purpose.

Ontology plays an important role towards enhancing the integration and interoperability of the semantic web services. A significant amount of research has been done towards formalizing the notion of conflict between two ontologies. In [6], authors present a detailed classification of conflicts by distinguishing between *conceptualization* and *explication* mismatches. In [19] authors further generalize the notion of conflicts and classify semantic mismatches into language level mismatches and ontology level mismatches. Then ontology level mismatches are further classified into conceptualization mismatch and explication mismatch. Further research in the same direction [20] adds few new types of conceptualization mismatches. Researchers in [21] present alternative types of conflicts that are primarily relevant to OWL based ontologies. However primary focus of these works is towards the interoperability between two ontologies – rather than the correctness of the protocol for information exchange with respect to the interpretation.

Ontology mapping primarily focuses on combining multiple heterogeneous ontologies. In [22] authors address the problem of specifying a mapping between a global and a set of local ontologies. In [23] authors discuss about establishing a mapping between local ontologies. In [24] the problem of ontology alignment and automatic merging is addressed.

Significant amount of research has been done towards the development of the protocol. In [25] researchers proposed a methodology for developing protocols in a multi agent environment. They extend propositional dynamic logic to formally specify the protocol and also use an extension of state-charts for visual representation. In [26] a step by step procedure is presented for the development of web service interaction protocols from the problem definition to the final specification. However these approaches are focused towards the development of protocol for multi agent environment. The semantics of the exchanged data is not addressed in these works.

The problem of checking compatibility between two ontologies with respect to a protocol is new and to the best of our knowledge there is no prior work on this topic.

7 Conclusion

In this paper we addressed the problem of detecting the presence of semantic mismatch where the data exchange between two ontologies is defined in terms of a protocol. We believe that the proposed methodology will be very helpful for the integration of web services that are developed independently. Moreover the future of internet applications lie in exchanging knowledge, where semantic conflict will be a major issue.

Bibliography

[1] Guo, R., Chen, D., Le, J.: Matching semantic web services across heterogeneous ontologies. In: CIT. (2005) 264–268

[2] Noia, T.D., Sciascio, E.D., Donini, F.M., Mongiello, M.: Semantic matchmaking in a p-2-p electronic marketplace. In: SAC. (2003) 582–586

[3] OWL Web Ontology Language: http://www.w3.org/TR/owl-ref/

[4] Web Service Modeling Language: http://www.wsmo.org/wsml/

[5] The Dublin Core Metadata Initiative: http://dublincore.org/

[6] Visser, P.R.S., Jones, D.M., Bench-Capon, T.J.M., Shave, M.J.R.: An analysis of ontology mismatches; heterogeneity versus interoperability. AAAI Spring Symposium on Ontological Engineering (1997)

[7] Castano, S., Ferrara, A., Montanelli, S.: Matching ontologies in open networked systems: Techniques and applications. (2006) 25–63

[8] Hameed, A., Sleeman, D.H., Preece, A.D.: Detecting mismatches among experts' ontologies acquired through knowledge elicitation. Knowl.-Based Syst. **15**(5-6) (2002) 265–273

[9] Ghosh, P., Dasgupta, P.: A formal method for detecting semantic conflicts in protocols between services with different ontologies. In Meghanathan, N., Boumerdassi, S., Chaki, N., Nagamalai, D., eds.: Recent Trends in Networks and Communications. Volume 90 of Communications in Computer and Information Science., Springer Berlin Heidelberg (2010) 553–562

[10] OAEI Benchmark: http://oaei.ontologymatching.org/2009/benchmarks/

[11] Foster, H., Uchitel, S., Magee, J., Kramer, J.: Compatibility verification for web service choreography. In: ICWS. (2004) 738–741

[12] Betin-Can, A., Bultan, T., Fu, X.: Design for verification for asynchronously communicating web services. In: WWW. (2005) 750–759

[13] Zhao, X., Yang, H., Qiu, Z.: Towards the formal model and verification of web service choreography description language. In: WS-FM. (2006) 273–287

[14] Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web service interfaces. In: WWW. (2005) 148–159

[15] Guo, R., Le, J., Xia, X.: Capability matching of web services based on owl-s. In: DEXA Workshops. (2005) 653–657

[16] Pathak, J., Koul, N., Caragea, D., Honavar, V.: A framework for semantic web services discovery. In: WIDM. (2005) 45–50

[17] Klusch, M., Fries, B., Sycara, K.P.: Automated semantic web service discovery with owls-mx. In: AAMAS. (2006) 915–922

[18] Vu, L.H., Hauswirth, M., Aberer, K.: Towards p2p-based semantic web service discovery with qos support. In: Business Process Management Workshops. (2005) 18–31

[19] Klein, M.: Combining and relating ontologies: an analysis of problems and solutions. In: Workshop on Ontologies and Information Sharing, IJCAI'01, Seattle, USA (2001)

[20] Qadir, M.A., Fahad, M., Noshairwan, M.W.: On conceptualization mismatches between ontologies. In: GrC. (2007) 275–278

[21] Li, C., Ling, T.W.: Owl-based semantic conflicts detection and resolution for data interoperability. In: ER (Workshops). (2004) 266–277

[22] Calvanese, D., Giacomo, G.D., Lenzerini, M.: A framework for ontology integration, IOS Press (2001) 303–316

[23] Madhavan, J., Bernstein, P.A., Domingos, P., Halevy, A.Y.: Representing and reasoning about mappings between domain models. (2002) 80–86

[24] Noy, N.F., Musen, M.A.: Anchor-prompt: Using non-local context for semantic matching. In: In Proceedings of the Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI. (2001) 63–70

[25] Paurobally, S., Cunningham, J.: Developing agent interaction protocols using graphical and logical methodologies. In: PROMAS, volume 3067 of LNCS, Springer (2003) 149–168

[26] Oluyomi, A., Sterling, L.: A dedicated approach for developing agent interaction protocols. In: PRIMA. (2004) 162–177