AN API FOR DISTRIBUTED REASONING ON NETWORKED ONTOLOGIES WITH ALIGNMENTS

Chan Le Duc, Myriam Lamolle
IUT Montreuil, LIASD-University Paris 8, 140 rue de la Nouvelle France, 93100 Montreuil, France

Antoine Zimmermann
Digital Enterprise Research Institute, IDA Business Park, Lower Dangan, Galway, Ireland

Keywords: Distributed reasoning, Ontology, Alignment, Networked ontologies, OWL.

Abstract: In this paper, we describe design and implementation of a Java interface for distributed reasoning on networked ontologies with alignments. This API is built over the standard OWLlink interface which is a communication protocol between OWL2 components. It is compatible with usual reasoners based on OWL such as Pellet and FaCT++ in centralized contexts. In this API, we have implemented an optimized distributed reasoning algorithm which, on the one hand requires specific primitives to manipulate OWL ontologies and alignments between them, and on the other hand, provides services to check local and global consistency of networked ontologies with alignements. The main feature of this API is that it can be easily integrated in various editors and tools dedicated to ontology manipulation.

1 INTRODUCTION

The Semantic Web is founded on technologies related to ontologies. An ontology can be built by using a language whose semantics is formal, e.g. OWL (Ontology Web Language[1]). This allows us to define unambiguously the meaning of new terms from a set of atomic terms which can be concepts, relations or individuals. Such an OWL ontology enables new knowledge to be inferred by using reasoners such as Pellet (Sirin et al., 2007), FaCT++ (Tsarkov and Horrocks, 2006). In addition, an application domain can be described by several ontologies which are related in some way. In this context, a reconciliation between such ontologies can yield new pieces of knowledge which establish correspondences between entities of two ontologies. A set of such correspondences is called an *alignment*. (Borgida and Serafini, 2003) proposed a formalism, namely DDL (Distributed Description Logics), to represent a system of ontologies with alignments. DRAGO reasoner [2] resulting from this work allows to check consistency of such system and provides other applications related to alignment manipulations, e.g., alignment debugging and minimization.

The most of these reasoners have been integrated within ontology editors such as Protg, Swoop, etc. However, they are only usable through interfaces (API). These API are more and more numerous. They are often specific to reasoners (e.g. KAON2 API). The proliferation of APIs leads to two issues. On the one hand, a change of environment can imply to modify source codes of an application and recompile them in order to be able to use a given reasoner. On the other hand, there exist several reasoners supporting different semantics such as standard Description Logics (DL), Distributed Description Logics (DDL) (Borgida and Serafini, 2003), Integrated Distributed Description Logics (IDDL) (Zimmermann and Le Duc, 2008), etc.. This fact implies that specific APIs offer the same services, but with different syntaxes. Developers must make efforts to learn APIs on which their application depends.

To address this issue, a W3C working group proposes a new protocol, namely, OWLlink (Liebig et al., 2009). The main goal is to facilitate (i) the specification of reasoners with knowledge bases associated, (ii) the axiom specification, and (iii) the manner of asking inferred results. This protocol seems to be par-

[1] http://www.w3.org/TR/owl-features/
[2] http://drago.itc.it/download.html

Le Duc C., Lamolle M. and Zimmermann A. (2010).
AN API FOR DISTRIBUTED REASONING ON NETWORKED ONTOLOGIES WITH ALIGNMENTS.
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development*, pages 295-304
DOI: 10.5220/0003101002950304

ticularly interesting since it is extensible. This enables us to add functionalities adequate to different kinds of current and future reasoners.

The main functionalities of the proposed API are creation and checking consistency of a network of OWL ontologies with alignments. In addition, it allows users to select a semantics associated to a given ontology network (e.g. DL, DDL, IDDL), and thus reasoner corresponding to the selected semantics is determined as well (Horridge et al., 2007). In this paper, we focus on the IDDL reasoner[3] which implements the algorithm presented in (Zimmermann and Le Duc, 2008). This reasoner supports distributed reasoning services that allow us :

- to deal with ontologies and ontology alignments (i.e. an ontology network constituted of local OWL ontologies and their alignments),
- to check local and global consistency of a network of OWL ontologies with alignments.

Roughly speaking, the IDDL reasoner performs the following tasks : (i) checking consistency of alignments by using a global reasoner (e.g. Pellet), (ii) propagating knowledge from alignments to each local ontology and asking consistency of the local ontology with respect to the propagated knowledge, and (iii) collecting the consistency result from each local ontology and deciding consistency of the whole system.

According to this schema, checking consistency of each local ontology with respect to the knowledge propagated from alignments can be carried out independently by local reasoners. By consequent, the computation for deciding consistency of the whole system is performed in a distributed way.

In the following, we present the main features of different reasoners in Section 2. We introduce in Section 3 the new protocol OWLlink which facilitates communication between reasoners. We propose in Section 4 different ways to use the IDDL reasoner by our own Java API based on OWLlink. Section 5 provides details on an optimized implementation of the algorithm presented in (Zimmermann and Le Duc, 2008). Section 6 presents how to integrate the IDDL reasoner within the NeOn Toolkit [4]. We conclude and present future work in Section 7.

2 EXISTING REASONERS

Reasoning on ontologies is essentially aimed to infer new knowledge. Reasoners can be also used to check ontology consistency and to classify terms (classes or relations) in an ontology.

2.1 Reasoners for Simple Ontology

Amongst existing reasoners, which have a Java interface and use OWL API, Pellet and FaCT++ are the most common. Therefore, we claim that an API used in distributed reasoners must be compatible with them. We choose Pellet to experiment the integration of new reasoners in a transparent way to end-user. The tested interface with Pellet remains identical to other open reasoners. In the context of distributed reasoning supported by the IDDL reasoner, Pellet is used to deal with reasoning at global level with alignments.

The OWL API has a standard interface *org.semanticweb.owl.inference.OWLReasoner* with several methods such as *isConsistent()*, *isSatisfiable(OWLClass)*, *isEquivalent(OWLClass, OWLClass)*, etc. Pellet provides an implementation of these methods. In terms of interoperability with existing reasoners, Pellet can be used with Jena as follows:

```
import org.mindswap.pellet.jena.PelletReasonerFactory;
import com.hp.hpl.jena.rdf.model.InfModel;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.reasoner.Reasoner;
// creating Pellet reasoner
Reasoner reasoner =
        PelletReasonerFactory.theInstance().create();
// creating empty model
Model emptyModel = ModelFactory.createDefaultModel();
// creating inference engine
// using Pellet reasoner
InfModel model = ModelFactory.createInfModel(reasoner,
                                        emptyModel);
```

This interoperability ability makes Pellet easier to be integrated within systems based on RDF ontologies.

2.2 Reasoners for Networked Ontologies

Another reasoners support distributed reasoning on ontologies and alignments such as DRAGO[5]. The peer-to-peer DRAGO system uses a procedure based on distributed tableau algorithm for reasoning with several ontologies interconnected by semantic mappings. This distributed reasoner is built on the top of standard tableau reasoner as Pellet or FaCT++. The algorithm for distributed reasoning as described in (Serafini and Tamilin, 2005) builds a distributed

[3] http://gforge.inria.fr/projects/iddl

[4] http://neon-toolkit.org/wiki

[5] http://drago.itc.it/download.html

tableau and consults the consistency of other local ontologies. If there is a local ontology participating to bridge rules (mappings) the algorithm consults another distributed reasoner at the peer where the local ontology is located. This process may be propagated in the whole network.

The major disadvantage of this algorithm is that it requires a tableau-based reasoning for every peer. Therefore, the heterogeneity of reasoning mechanisms is impossible.

Moreover, DRAGO adopts the Pellet API by adding *"D"* to method signatures to dedicate distributed reasoning (e.g., *isDconsistent()*, *isDSatisfiable()*). However, this interface does not satisfy one of our criteria (see Section 4) which says that the name of services (e.g. consistency, satisfiability) definable in two different semantics must not be different.

ContentMap[6] is a tool supporting the integration of independently designed ontologies by using mappings. This tool uses the standard reasoners to generate semantic consequences of integrated ontologies. It can help users to detect potential errors and to evaluate mappings. This work is comparable to that of (Meilicke et al., 2009), who proposed a method to help human experts to check automatically alignments created. They used DDL to formalize mappings as bridge rules, which exploit Drago to reason with alignments.

3 OWLlink PROTOCOL

If ontologies are expressed in OWL, it is necessary also to use standard communication to make easier interactions between reasoners and different services, mainly in distributed ontology context. OWLlink (Liebig et al., 2009), successor of DIG protocol, is a Java implementation-neutral communication interface for OWL (Figure 1). It follows the new W3C recommendations about extensible communication protocols for systems based on OWL2 (Motik et al., 2009), which allows one to add new functionalities required by specific applications. OWLlink overcomes drawbacks of DIG protocol (Bechhofer et al., 2003) (Dickinson, 2004) since it relies on DIG2 propositions (Turhan et al., 2006). OWLlink consists of two parts: (i) structural protocol specification, and (ii) link mechanism to transport protocols.

[6]http://krono.act.uji.es/people/Ernesto/contentmap

3.1 OWLlink Specification

OWLlink manages client/server communication (with HTTP protocol) by message through session (Wessel and Luther, 2009). Each message of request type (`requestMessage()`) on ontology corresponds a response message (`requestResponse()`). These two message types embed respectively a set of objects `Request` and `Response`. Associated with these messages, a mechanism of error management informs the client about errors arising from syntactic problems (`SyntaxError`), semantic problems (`SemanticError`) or ontology management (`KBError`). Clients can get also information about server itself (`GetDescription()` and `Configuration()`).

Moreover, OWLlink server can manage simultaneously several ontologies by their IRI or by ontology name. It can check axioms about classes, properties, or individuals. To do that, client must make a request (`tell`) embedding axioms to be checked.

In terms of interoperability, OWLlink provides a mechanism enabling to encapsulate existing OWL-based reasoners (Pellet, FaCT++, etc.). This facilitates system design which needs to deal with several ontologies associated with local reasoners. This ability with client/server communication gives fundamental elements on which a API dedicated to distributed reasoning can be elaborated.

3.2 Basic Primitives

OWLlink provides a set of basic primitives[7] to access ontologies to obtain information about entities, status, schema and individual. For instance, the primitive `isKBDeclaredConsistent()` checks ontology consistency. Regarding the schema of ontologies, the primitive `isClassSatisfiable()` and `isClassSubsumedBy()` allow to ask whether a class is satisfiable or a class is subsumed by another one. Finally, knowledge about individuals can be obtained by using `isInstanceOf()`, `areIndividualsRelated()` or `getEquivalentIndividuals()`, etc.

4 IDDL REASONER INTERFACE

Zimmermann (Zimmermann, 2007) has introduced IDDL (Integrated Distributed Description Logics) as

[7]http://owllink-owlapi.sourceforge.net/documentation.html

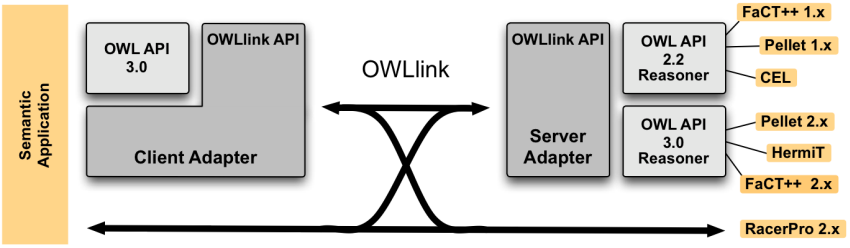

Figure 1: OWLlink Architecture (source http://owllink-owlapi.sourceforge.net/).

a new formalism enabling to represent a set of ontologies with their alignments, i.e. networked ontologies interconnected by alignments. Such a network consists of ontologies expressed by OWL, and alignments considered as a set of subsumption or disjunction relations between ontology entities (concept/role/individual). This formalism is adapted particularly to reason with OWL ontology alignment automatically generated by tools as Alignment Server[8]. The differences between IDDL and other formalisms are:

1. In IDDL, alignments are considered as knowledge pieces independent from ontology knowledge. As a result, when knowledge is propagated from alignments to local ontologies, new semantic consequences can be entailed.
2. IDDL does not make any expressiveness hypothesis over used formalisms in local ontology apart from decidability. This enables the heterogeneity of reasoning mechanisms and formalisms that are used in local ontologies. For instance, a local reasoner uses a tableaux-based algorithm while another reasoner can decide consistency with help of an automata-based algorithm.
3. IDDL supports a real distributed reasoning, i.e. all reasoning on local ontologies can be carried out independently.

Like the most of reasoners, IDDL gives two main services: consistency checking and entailment. IDDL reasoner provides the two following interfaces:

1. "standalone" interface compatible with classical reasoners (Pellet, FaCT++). The IDDL reasoner consists of main class `IDDLReasoner`, which implements standard interface `org.semanticweb.owl.inference.OWLReasoner`,
2. interface for distributed reasoning which is based on `OWLlinkReasoner` presented in section 3.

It is important to note that the services available in the IDDL API must have the same signature that should not depend on which reasoner associated with a semantics is used. Moreover, from users' point of view, this feature facilitates changes from a reasoner to another, and corresponds to the fact that we can talk about (the) consistency of a system including several logics in condition that the notion of consistency is generally definable for each one. This is the case for a system consisting of DL (OWL), DDL, IDDL, etc.. Besides, distributed reasoners depend on APIs for manipulating correspondences between ontologies. For this purpose, the IDDL reasoner currently uses the Alignment API (Euzenat, 2004). It may arise an issue if users replace a distributed reasoner by another and they use different Alignment APIs.

4.1 Classical Reasoning with a Standalone Interface

Regarding classical reasoning, the IDDL reasoner merges all local ontologies and their alignments, then applies reasoning on the unique ontology. To do so, the IDDL reasoner provides the following primitives (where `<domain>=fr.inrialpes.exmo.iddl`):
`<domain>.IDDLReasoner.isConsistent()` returns *true* if and only if the resulting ontology is consistent,
`<domain>.IDDLReasoner.isEntailed(OWLAxiom)` returns *true* if and only if the axiom can be entailed from the resulting ontology,
`<domain>.IDDLReasoner.isEntailed(Alignment)` returns *true* if and only if each axiom belonging to *Alignment* can be entailed from the resulting ontology.

From this point of view, the `OWLlink` interface (presented in section 3) can be used to encapsulate the IDDL reasoner in the same way as classical reasoner (i.e. OWLReasoner). In this case, the IDDL reasoner with the OWLlink protocol is comparable to a local IDDL system with regard to the global IDDL system.

[8]http://aserv.inrialpes.fr/

4.2 Distributed Reasoning with `OWLlinkReasoner`

In distributed reasoning context, the IDDL reasoner consists of global reasoner using also Pellet. Moreover, the IDDL reasoner needs to know consistency of each local extended ontology (Zimmermann and Le Duc, 2008) with the help of local associated reasoners. The local reasoners can be Pellet, FaCT++, DRAGO, IDDL itself, etc., and they can be located in different sites. `OWLlinkReasoner` interface provides necessary elements that the IDDL reasoner needs to ensure communication between local and global reasoners. Indeed, in order to define the method:

```
IDDLReasoner.addOntology(OWLOntology onto1,
                         URL localReasoner);
```

the following code can be used:

```
URL reasonerURL = new URL( ... );
OWLlinkReasoner reasoner =
   new OWLlinkHTTPXMLReasoner(
       OWLOntologyManager manager,
       URL reasonerURL);
CreateKB createKB = new CreateKB();
KB kb = reasoner.answer(createKB);
IRI kbIRI = kb.getKB();
// configuration : a set of global axioms
Tell tell = new Tell(kbIRI,
     OWLAxioms[] configuration);
OK ok = reasoner.answer(tell);
...
```

Roughly speaking, a configuration is a set of axioms propagated from alignments to local ontologies. This notion will be clarified in Section 5.

This code enables to associate a local ontology (`createKB`) to a local reasoner (`reasoner` which is responsible for deciding and transmitting the result (consistency) to the global reasoner (`reasoner.answer(tell)`) whether the local ontology is consistent with respect to the configuration (`new Tell(kbIRI, OWLAxioms[] configuration)`). The following code shows how to use the IDDL reasoner.

```
import fr.inrialpes.exmo.iddl.IDDLreasoner;
import fr.inrialpes.exmo.iddl.types.Semantics;
import org.semanticweb.owl.model.OWLAxiom;
import org.semanticweb.owl.align.Alignment;
 ...
IDDLReasoner reasoner = new IDDLReasoner();
reasoner.addAlignment(URI onto1);
reasoner.addAlignment(URI align1);
reasoner.setSemantics(Semantics.DL);
//check consistency of DL system
reasoner.isConsistent();
//check entailment of an OWL axiom
reasoner.isEntailed(OWLAxiom ax);
//check entailment of an alignment
reasoner.isEntailed(Alignment al);
 ...
reasoner.addOntology(onto1);
//associate a local reasoner to an ontology.
reasoner.addOntology(URI onto1,
                     URL localReasoner1);
reasoner.addAlignment(URI align1);
reasoner.setSemantics(Semantics.IDDL);
//check consistency of IDDL system
reasoner.isConsistent();
```

5 OPTIMIZATION AND IMPLEMENTATION

This section briefly presents the principle of the algorithm for distributed reasoning and its implementation in the IDDL reasoner. This version does not allow disjointness correspondences to occur in alignments. This restriction does not lead to a serious drawback of the expressiveness since alignments generated by the majority of matching algorithms do not often include disjointness correspondences.

5.1 Algorithm and Optimization

The distributed algorithm implemented in the IDDL reasoner was presented in (Zimmermann and Le Duc, 2008). However, the vocabulary used in (Zimmermann and Le Duc, 2008) differs from the one used here. According to the metamodel, wherever *ontology* is used in (Zimmermann and Le Duc, 2008), we can replace it by *module* without loss of correctness. Where *alignment* is used in (Zimmermann and Le Duc, 2008), we use *mapping* here. Finally, a *correspondence* in (Zimmermann and Le Duc, 2008) is the equivalent of a *mapping assertion* here.

5.1.1 Preliminary Assumption

The IDDL reasoner works by having a module reasoner communicate with imported modules' reasoners. The imported modules reasoners are supposed to be encapsulated so that their implementation is unknown but they can be used via an interface. Consequently, for each imported module m_i, we assume that there exists an oracle F_i which takes a set of DL axioms A_i as arguments and returns a boolean equal to Consistency$(m_i \cup A_i)$.

Definition 1 (Reasoning oracle)**.** *Let O be an ontology defined in a logic L. A* reasoning oracle *is a boolean function $F : \mathcal{P}L \rightarrow$* Bool *which returns*

$F(A) = \text{Consistency}_L(O \cup A)$, *for all sets of axioms* $A \in \mathcal{PL}$.

The term ontology in Defintion 1 is to be taken in a general sense. It can be a module or even a distributed system, as long as the associated reasoner can interpret DL axioms and offers correct and complete reasoning capabilities. In practice, such oracles will be implemented as an interface which encapsulates a reasoner like Pellet, Fact++, or a module reasoner.

A module reasoner must call the oracles associated with the imported modules with well chosen axioms in order to determine consistency. The choice of axioms will be explained below. In addition, the module reasoner have access to the mappings that may exist between imported modules. From the importing module's point of view, these mappings are treated like local axioms. Therefore, we can consider that the mappings are equivalent to an ontology (called *the alignment ontology* in (Zimmermann and Le Duc, 2008)).

Definition 2 (Alignment ontology)**.** *Let* $\mathbf{A}$ *be a set of mappings. The alignment ontology is an ontology* $\widehat{\mathbf{A}}$ *such that:*

- *for each mapping assertion* $i:C \xleftrightarrow{\sqsubseteq} j:D$ *with* C *and* D *local concepts,*
  - $\widehat{i:C}$ *and* $\widehat{i:D}$*;*
  - $\widehat{i:C} \sqsubseteq \widehat{i:D} \in \widehat{\mathbf{A}}$*;*
- *for each mapping assertion* $i:P \xleftrightarrow{\sqsubseteq} j:Q$ *with* P *and* Q *local roles,*
  - $\widehat{i:P}$ *and* $\widehat{i:Q}$*;*
  - $\widehat{i:P} \sqsubseteq \widehat{i:Q} \in \widehat{\mathbf{A}}$.

In order to check the global consistency of the module, we also assume that there is a reasoning oracle F_A associated to $\widehat{\mathbf{A}}$. The algorithm consists in questioning all the reasoning oracles with well chosen axioms that are detailed just below.

5.1.2 Algorithm Overview

In (Zimmermann and Le Duc, 2008), it is formally proven that consistency checking of an IDDL system with only subsumption mapping assertions can be reduced to determining the emptiness and non emptiness of specific concepts. More precisely, we define the notion of configuration, which serves to explicitly separate concepts between empty concepts and not empty concepts, among a given set of concepts. It can be represented by a subset of the given set of concepts, which contains the asserted non-empty concepts.

Definition 3 (Configuration)**.** *Let* $\mathscr{C}$ *be a set of concepts. A* configuration Ω *over* $\mathscr{C}$ *is a subset of* $\mathscr{C}$.

In principle, a configuration Ω implicitly assert that for all $C \in \Omega$, $C \sqsubseteq \bot$ and for all $C \notin \Omega$, $C(a)$ for some individual a. A similar notion of *role configuration* is also defined in (Zimmermann and Le Duc, 2008), but for the sake of simplicity we will only present it for concepts.

The algorithm then consists in selecting a configuration over the set of all concepts of the alignment ontology. The axioms associated with the configuration are then sent to the oracles to verify the consistency of the resulting ontologies. If they all return *true* (i.e., they are all consistency with these additional axioms) then the modular ontology is consistent. Otherwise, another configuration must be chosen. If all configurations have been tested negatively, the modular ontology is inconsistent, according to the proof in (Zimmermann and Le Duc, 2008). Since there is a finite number of configurations, this algorithm is correct and complete.

The sets of axioms that must be used to query the oracles are defined according to a configuration Ω as follows.
Let A (resp. $A_1, \ldots, A_n$) be the sets of axioms associated with the oracles of the alignment ontology (resp. with the oracle of modules $m_1, \ldots, m_n$).
For all imported modules m_i,

- for all concepts $i:C \in \Omega$, $\widehat{i:C}(a) \in A$ and $C(a_C) \in A_i$ where a is a fixed individual and a_C is a new individual in m_i.
- for all concepts $i:C \notin \Omega$, $\widehat{i:C} \sqsubseteq \bot \in A$ and $C \sqsubseteq \bot \in A_i$.

5.1.3 Optimization

We can identify, from the algorithm as described above, some situations that may lead to blow-up complexity.

1. The algorithm will answer negatively if and only if it has tested all possible configurations. So, every time a module is inconsistent, the reasoner must call all the oracles 2^n times, where n is the number of concepts in the alignment ontology. This situation is hardly acceptable in a practical reasoner.

   However, optimizations can be carried out to improve this situation. In particular, backtrack algorithms can be applied to this procedure. Indeed, for each concept C appearing in a mapping, it must be decided whether it is empty or not. There are cases where it can be deduced that C is empty (resp. not empty) immediately. In this case, it not necessary to test configurations where C is not empty (resp. empty). This can be visualized in Figure 2.

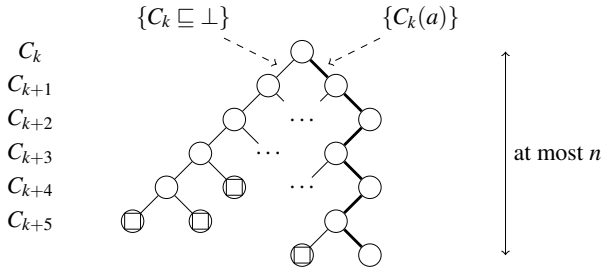

Figure 2: At each node, the left branch indicates that the concept C_k is asserted as an empty concept ($C_k \sqsubseteq \bot$), while the right branch indicates a non empty concept ($C_k(a)$). The thick path indicates a possible configuration for the distributed system.

2. Theoretically, there are 2^n possible global configurations where n is the number of terms occuring in alignments. Each configuration Ω is of the form $\Omega = (X_1, \cdots, X_n)$ where $X_i = C(a)$ asserts the non-emptyness of the concept C or $X_i = (C \sqsubseteq \bot)$ asserts the emptyness of the concept C. We denote $\bar{X}_i = C(a)$ if $X_i = (C \sqsubseteq \bot)$ and $\bar{X}_i = (C \sqsubseteq \bot)$ if $X_i = C(a)$. It holds that $O_i \cup \Omega$ is not consistent if there is $X_i \in \Omega$ such that $O_i \cup \{X_i\}$ is not consistent.

   This observation allows the global reasoner to reduce dramatically the number of checks on consistency of local ontologies since the global reasoner can reuse the inconsistency result of $O_i \cup \{X_i\}$ for all $X_i \in \Omega$ and for all configuration Ω.

6 INTEGRATION OF THE IDDL REASONER WITHIN NEONTOOLKIT

In this section we describe the principle of the integration of the IDDL reasoner within the NeOn Toolkit. This integration is performed by developing a plug-in, namely IDDL reasoner plug-in, which plays an interface role between the IDDL reasoner and the NeOn-Toolkit plug-ins, e.g. the module API, Ontology Navigator, Alignment Plugin, etc..

The NeOn toolkit is an environment for managing and manipulating networked ontologies developed. It was developed within the NeOn project as a plug-in for managing ontologies under Eclipse and extends previous products such as KAON2. The NeOn toolkit features run time and design time ontology alignment support. It can be extended through a plug-in mechanism, so it can be customized to the users needs. As a development environment for ontology management, the NeOn Toolkit supports the W3C recommendations OWL and RDF as well as F-Logic for processing rules. With the support of the integrated mapping-tool, named OntoMap, heterogeneous data sources, e.g., databases, file systems, UML diagrams, can be connected to ontologies quickly and easily.

The IDDL reasoner API for ontology modules uses the module API to access to **alignments** and **imported modules** of an ontology module. In other terms, the IDDL reasoner plug-in is developed such that it can get access to the alignments and imported modules from an ontology module and pass them to the IDDL reasoner with help of the IDDL reasoner API.

More precisely, from the NeOn toolkit environment the IDDL reasoner plug-in gets URIs of the imported modules and the alignments from an ontology module. In the most of cases where alignments are not available from the ontology module, the plug-in can fetch available alignments from an alignment server and use them as alignments. This feature allows users to use alignments permanently stored on servers and select the most suitable alignments for an intended purpose.

The IDDL reasoner plug-in relies on the IDDL reasoner, and the core module API which provides basic operations to manipulate ontology modules and alignments. By using the core module API, the IDDL reasoner plug-in can get necessary inputs from an ontology module. In the case where the alignments obtained from the ontology module in question are not appropriate, the plug-in can connect to Alignment Server[9] to fetch alignments available. For this purpose, the plug-in offers to users an interface allowing to visualize and select alignments. Figure 3 is a screenshot of the plugin integrated within Neon Toolkit.

From a determined input, the IDDL reasoner plug-in can obtain an answer for consistency from the IDDL reasoner. In the case where the answer is negative the plug-in can obtain an explanation indicating configurations and/or correspondences which are responsible for that inconsistency.

[9] http://aserv.inrialpes.fr/

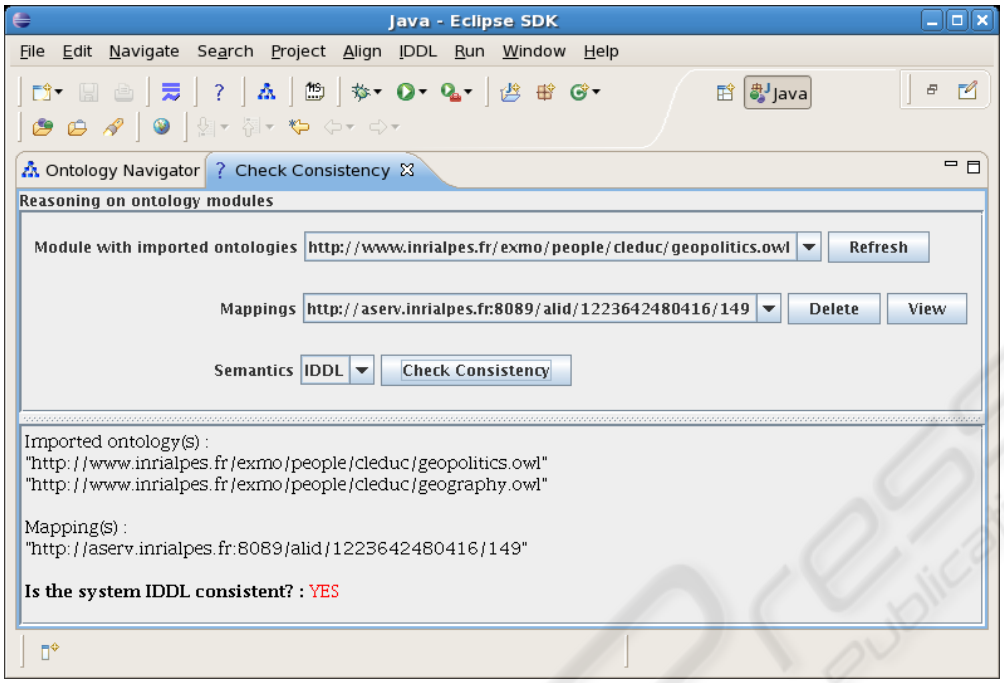

Figure 3: An IDDL consistent system.

The answer time of the IDDL reasoner depends on the following elements which are taken into account in the optimized algorithm design.

1. If there are more unsatisfiable or non-empty concepts occurring in correspondences, the answer time is shorter,
2. If there are more equivalent concepts or properties occurring in correspondences, the reasoner answers faster,
3. If ontology module is inconsistent, the reasoner has to check likely all configurations. Therefore, the answer time would be long.

In Example 4, we have two imported ontologies : **Geopolitics** and **Geography** with a mapping between them. The axioms of the ontologies and mapping are expressed in a description logic and they can be directly coded in OWL. We consider the following cases:

1. If these imported modules are merged with the correspondences of the mapping, we obtain an OWL ontology which is not consistent. The reason is that the mapping allows one to deduce that two classes "EuropeanRegion" and "SouthAmericanRegion" are not disjoint, which contradicts the disjointness axiom in **Geography**.
2. However, in the context of ontology modules the IDDL reasoner can check consistency of the module and answer that the module is consistent (Figure 3).
3. If we now add to the following mapping $1 : \mathit{Guyana} \overset{\in}{\leftrightarrow} 2 : \mathit{SouthAmericanRegion} \sqcap \mathit{EuropeanRegion}$, the IDDL reasoner answers that the module is no longer consistent.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we argued that it is necessary to introduce a generic API for reasoners to facilitate environment changes without upgrading the current API version. Moreover, we suggested to use the same signature of the reasoning services independently from reasoners associated with a given semantics.

Then, we proposed a design and implementation of such an API which meets these two criteria. Finally, we optimized and implemented in this API a distributed algorithm for checking consistency of networked ontologies with alignments based on the IDDL semantics.

Comparing to other approaches, e.g. the distributed reasoning based on DLL, the IDDL distributed reasoning is more modular and heterogeneous, i.e., what global reasoner needs about a local ontology is whether it is consistent. Thus, local ontologies can use different logics in condition that they

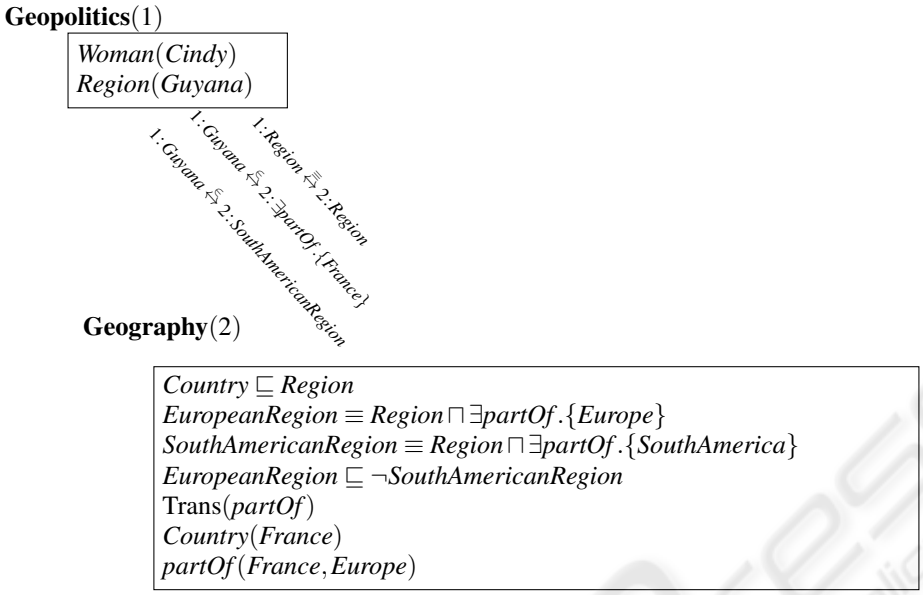

Figure 4: An example of an ontology module with mappings.

remain to be decidable. This feature of IDDL provides possibility to use different algorithms for reasoning on local ontologies.

The current version of the IDDL reasoner provides only explanations for inconsistencies which are caused by correspondences propagated from mappings to local ontologies. A future version of the IDDL reasoner should take advantages of explanations from local reasoners to give more details about how propagated correspondences impact on a local ontology.

As mentioned at the beginning of the present section, the current version of the IDDL reasoner does not allow disjointness correspondences to occur in alignments. This limitation prevents us from supporting axiom entailment since it is equivalent to inconsistency of an IDDL system including disjointness correspondences. For instance, the current IDDL reasoner does not know whether $\langle \mathbf{O}, \mathbf{A} \rangle \models i{:}C \sqsubseteq j{:}D$ where $\mathbf{O}$, $\mathbf{A}$ are the sets of imported ontologies and mappings from an ontology module.

In a future version, we plan to extend the reasoner such that it takes into account only disjointness correspondences translated from entailment but not those initially included in alignments. Allowing disjointness correspondences in this controlled way may not lead to a complexity blow-up.

REFERENCES

Bechhofer, S., Moller, R., and Crowther, P. (2003). The DIG Description Logic Interface. In Calvanese, D., editor, *Proceedings of the International Workshop on Description Logics (DL03), Volume 81 of CEUR., Rome, Italy*.

Borgida, A. and Serafini, L. (2003). Distributed description logics : Assimilating information from peer sources. *Journal Of Data Semantics*, 1(1):153–184.

Dickinson, I. (2004). Implementation experience with the DIG 1.1 specification. Technical report, Hewlett-Packard.

Euzenat, J. (2004). An API for Ontology Alignment. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, volume 3298 of *LNCS*, pages 698–712. Springer.

Horridge, M., Bechhofer, S., and Noppens, O. (2007). Igniting the OWL 1.1 Touch Paper: The OWL API. In *3rd OWL Experienced and Directions Workshop, OWLED 2007, Innsbruck, Austria*.

Liebig, T., Luther, M., and Noppens, O. (2009). The owllink protocol. In *OWLED*.

Meilicke, C., Stuckenschmidt, H., and Tamilin, A. (2009). Reasoning support for mapping revision. In *Journal of Logic and Computation*.

Motik, B., Patel-Schneider, P., and Parsia, B. (2009). OWL2 Web Ontology Language: Structural Specification and Functional-Style Syntax. Technical report, World Wide Web Consortium.

Serafini, L. and Tamilin, A. (2005). DRAGO: Distributed reasoning architecture for the semantic web. In Lecture Notes in in computer science, S., editor, *Proceed-*

ing of the 2nd European Semantic Web Conference, pages 361–376.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: a pratical owl-dl reasoner. *Journal of Web Semantics*, 5(2):51–53.

Tsarkov, D. and Horrocks, I. (2006). FaCT++ Description Logic Reasoner: System Description. In Lecture Notes in Artificial Intelligence, S., editor, *Proceeding of the International Joint Conference on Automated Reasoning*.

Turhan, A.-Y., Bechhofer, S., Kaplunova, A., Liebig, T., Luther, M., Noppens, O., Patel-Schneider, P., Suntisrivaraporn, B., and Weithner, T. (2006). DIG 2.0 : Towards a Flexible Interface for Description Logic Reasoners. In *Proceedings of the OWL Experiences and Directions Workshop (OWLED06) at the ISWC06, Athens, Georgia, USA*.

Wessel, M. and Luther, M. (2009). OWLlink: HTTP/Functional Binding, Version 1.0. Technical report, W3C.

Zimmermann, A. (2007). Integrated distributed description logics. In *Proceedings of the 20th International Workshop on Description Logics (DL 2007)*, pages 507–514.

Zimmermann, A. and Le Duc, C. (2008). Reasoning with a network of aligned ontologies. In *Proceedings of the 2nd International Conference on Web Reasoning and system rules, LNCS 5341*, pages 43–75.