



D20.2v0.2 OWL Lite- Reasoning with Rules

WSML Working Draft 23 November 2004

This version:

<http://www.wsmo.org/2004/d20/d20.2/v0.2/20041123/>

Latest version:

<http://www.wsmo.org/2004/d20/d20.2/v0.2/>

Previous version:

<http://www.wsmo.org/2004/d20/d20.2/v0.1/20041006/>

Authors:

Andreas Harth
Stefan Decker

Reviewer:

Jos de Bruijn

This document is also available in non-normative [PDF](#) version.

Copyright © 2004 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Abstract

This deliverable describes reasoning with ontologies using the TRIPLE language and reasoning system and elaborates on requirements for a reasoner for WSML. As more and more ontologies become available online, infrastructure is needed for querying, processing and transforming ontologies with general-purpose tools.

A reasoner allows for instance and schema inferencing, which enables advanced query answering services, schema matching and ontology alignment, data integration across heterogeneous data sources, and personalization of web sites.

We assume that the ontology language can be described using logic formulas, and the reasoner will use the formulas to draw inferences and supply answers to queries over a knowledge base consisting of ground and inferred facts. The reasoner, which is based on logic programming, is independent from the ontology language if the semantics of the ontology language can be described using rules.

The notion of semi-structured data, in our case RDF, serves as underlying data format for encoding the syntax of ontologies. OWL Lite- is a subset of the Web Ontology Language OWL which can be encoded in RDF. In TRIPLE, rules are the underlying basis for implementing and processing inferences. We show in detail how the semantics of the OWL Lite- ontology language can be expressed using logical rules. We describe the rules used to reason over OWL Lite- informally. A TRIPLE program that captures the semantics of OWL Lite- is part of this deliverable.

Further, we describe the general architecture of the TRIPLE reasoning system, including its Java API and the remote access interface built on the basis of HTTP. We show how TRIPLE converts formulas expressed in the TRIPLE language into conjunctive normal form logic programs; the logic programs are then processed by XSB, a logic programming and deductive database system.

Contents

- [1 Introduction](#)
- [2 Requirements for Reasoner](#)
 - [2.1 Language Layering](#)

- [2.2 Scalability](#)
 - [2.3 Implementation](#)
 - [3 Reasoning with Ontologies using Rules](#)
 - [3.1 Web Ontology Language OWL](#)
 - [3.2 TRIPLE and Rules](#)
 - [4 A Ruleset for OWL Lite-](#)
 - [4.1 OWL Lite- Constructs](#)
 - [4.2 TRIPLE Listing](#)
 - [4.3 TRIPLE Header and Model Definitions](#)
 - [4.4 RDF Schema Features](#)
 - [4.5 Property Characteristics](#)
 - [4.6 Equivalence](#)
 - [4.7 Property Restrictions](#)
 - [4.8 Example](#)
 - [5 TRIPLE Architecture and Implementation](#)
 - [5.1 Overview](#)
 - [5.2 Java API](#)
 - [5.3 TRIPLE Server](#)
 - [5.4 Parsing and Transformations](#)
 - [5.5 XSB Deductive Database](#)
 - [6 Conclusions](#)
 - [References](#)
 - [Acknowledgement](#)
 - [Disclaimer](#)
 - [Appendix](#)
 - [TRIPLE BNF Grammar](#)
 - [TRIPLE API Javadoc](#)
-

1 Introduction

"Reasoning for ontologies" can be illustrated using the following two quotes that define the concept of reasoning and the concept of ontologies. "*Reasoning is the process of drawing conclusions from facts.*" [[Wos et al., 1992](#)]. "*Ontologies are a conceptualization of a specification.*" [[Gruber, 1993](#)]. This deliverable is therefore concerned with drawing conclusions (using a computer) from formal descriptions.

Ontology specifications, together with a large number of instance data, are becoming available on the Semantic Web in ontology languages such as RDFS and OWL. Access to data encoded in a semi-structured format as opposed to unstructured documents already facilitates many information integration tasks. However, the full power of ontologies can be only leveraged when the reasoning infrastructure is in place to process the vast amount of semi-structured information made available online. Reasoning will be an important building block in the future Semantic Web infrastructure.

Automated reasoning has a broad range of application scenarios: advanced query answering services, schema matching and ontology alignment, data integration across heterogeneous data sources, and personalization of web sites.

Advanced Query Answering: Current query answering systems on the Web have limited expressivity, and often can process queries against ontologies only on a syntactic level, using ground facts. A reasoner allows for advanced query answering over ontologies and instance data on a semantic level using the richer expressiveness that rules and ontologies provide.

Schema Matching and Ontology Alignment: Ontologies are typically created in the distributed web environment, with the result that a number of ontologies with semantic and syntactic differences emerge describing the same domain. However, users should not need to be aware of subtle differences in schemas, but should be able to query instance data using their own conceptual schema. Therefore, capabilities for matching schemas and aligning ontologies have to be provided, and a reasoner offers a declarative way of achieving the integration of schemas.

Data Integration across Heterogeneous Data Sources: With more and more data sources becoming available, users have to combine data from multiple data sources. Automatic schema mapping needs to be carried out, and some software has to locate data on the network. A reasoner can perform data integration in a distributed environment over heterogeneous data sources, given appropriate descriptions of capabilities and contents of the data sources, freeing the user from the burden of manually invoking queries and

integrating the results of multiple data sources

Personalization of Web Sites: The results of query and integration operations have to be presented to the user in a meaningful way. A reasoner can use rules to deduce what elements of results should be presented to the user, or which parts have to be ranked higher based on the preferences specified by the user. Content can be customized to the user's needs; for example a learner can choose to have a short, simple presentation or a long and detailed one of the same underlying course material.

The structure of this deliverable is as follows: in Section 2, we discuss requirements for a reasoning system deployed on the Web; Section 3 describes the connection between rules and ontologies; in Section 4, we present a ruleset for a subset of OWL Light; Section 5 takes architectural and implementation considerations into account; we conclude in Section 6 by summarizing our work and hint at possible further areas for extensions, notably integration into the Web infrastructure.

2 Requirements for Reasoner

The use cases presented in the previous section impose requirements on a reasoning system, which we will discuss in detail in the following section.

2.1 Language Layering

Language layering is important to be able to process syntactically and semantically different ontologies using the same reasoning system.

Creating an abstraction of the ontology language schema by using a semi-structured data format has advantages over implementing a special schema for every supported ontology language. The stack of Semantic Web language is based on RDF as the underlying data format, and the Web Ontology Language OWL is layered on top of RDF.

Similarly, a reasoner should be able to process data in a semi-structured data format such as RDF, and layer more advanced functionality on top. Logic is a formalism to express a large number of constructs, and is therefore suitable for expressing functionality used in ontologies. Indeed, many ontology languages are defined using axioms, therefore using axioms to provide inferencing services is a natural way of programmatically dealing with ontologies.

2.2 Scalability

Scalability is of concern because the amount of data that a reasoner must process will be large.

The amount of data processed in any real world application is typically in the gigabyte range. Traditional databases can handle large amounts of data, but lack the advanced functionality of reasoning systems. Reasoners for the Semantic Web must be able to handle large amounts of data efficiently, possibly in a distributed way.

Web search engines can answer keyword searches covering a large part of the web in a matter of milliseconds, so users expect similar response times from other query answering systems as well. Achieving the response times of web search engines for reasoning tasks is not a viable goal for the short term, but represents a "gold standard" that is worth pursuing.

2.3 Implementation

Implementation issues are addressed here since the reasoner is expected to be reused in other parts of WSML. The WSML reasoner is an enabler for other Semantic Web related projects. We expect subprojects to make use of the reasoner, and that poses several requirements in terms of usability and documentation.

The key requirement is that reasoning services have to be easy to use in developers' own programs. To achieve ease of use requires either a defined interface to a remote reasoning service, or a software package that can be installed on a broad range of operating systems. A pure Java implementation can help to achieve extensibility and a simple setup procedure. Compliance to web standards can also be beneficial in that developers can readily use their existing knowledge to get up to speed with regard to the reasoning system.

Another requirement for ease of use is that documentation about the reasoning language is available. Alternatively, the use of standards can make a large range of documentation, tutorials, etc. available.

Part of the implementation effort includes setting up infrastructure for a source code repository, a web site, and using other means also to encourage contributions in the open source spirit and to coordinate with projects at other institutions that currently use TRIPLE as a reasoning engine.

3 Reasoning with Ontologies using Rules

In the following section, we will discuss the ontology language OWL [Patel-Schneider et al., 2004] which is based on Description Logics, and describe TRIPLE [Sintek & Decker, 2002], a reasoner for rules rooted in Horn logic. Understanding both OWL and TRIPLE is mandatory for the next chapter where we present a TRIPLE ruleset for reasoning with a sublanguage of OWL.

3.1 Web Ontology Language OWL

OWL, the Web Ontology Language, is inspired on Description Logics, and comes in three sublanguages with increasing expressiveness, OWL Lite, OWL DL, and OWL Full [McGuinness & van Harmelen, 2004]:

- OWL Lite is capable of supporting a classification hierarchy and simple constraints. Classes and properties can be defined as equivalent, which makes schema matching and ontology alignment possible.
- OWL DL is based on description logics and is the subset of OWL that has computational completeness and decidability, which means that all computations are guaranteed to be computable and will finish in finite time. OWL DL is only a minor extension to OWL Light, allowing cardinality restrictions greater than 1 and allowing nominals.
- OWL Full has the maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

OWL being an ontology language for the Semantic Web satisfies the language layering requirement. OWL is based on the Resource Description Format (RDF), which is a W3C recommendation and can be seen as a data format for semi-structured data [Abiteboul et al., 2000]. RDF in turn is built on top of XML. Mainly due to the technology layering of Semantic Web technologies, a large number of tools are available for processing XML and RDF.

OWL Full and OWL DL are not suited for reasoning over large and distributed ontologies, since there exist no efficient reasoning algorithms for these sublanguages. Even OWL Lite as the least expressive sublanguage has constructs, such as equality, disjunction and negation, complicating the development and implementation of efficient reasoning systems for that language considerably. Therefore, proposals have been made to identify the intersection of Description Logic with Logic Programs [Grosz et al., 2003]. The largest subset of OWL Lite that can be expressed in Datalog is called OWL Lite- [de Bruijn & Fensel, 2004], which existing reasoning systems can process efficiently. OWL Lite- can be translated to Datalog, which itself is a subset of Horn logic. For Horn logic, a number of efficient evaluation strategies are available.

3.2 TRIPLE and Rules

TRIPLE is an efficient reasoning system for rules. The term "TRIPLE" denotes both the language and the open source inferencing engine which allows for processing programs expressed in the language. In contrast to procedural programming languages such as C or Java, TRIPLE is a declarative language which shares some similarities with SQL or Prolog. TRIPLE can process programs that consist of facts and rules from which conclusions for answering queries can be drawn. TRIPLE programs can be translated to Horn logic programs using a process outlined in Chapter 5. The engine is integrated into the Semantic Web context by providing import facilities for RDF.

The data model for TRIPLE is that of semi-structured data, which can be seen conceptually as a direct labeled graph with cycles. RDF is a data model based on the notion of semi-structured data, but takes into account various requirements for data representation imposed by the Web (e.g., URIs and linkage between data residing on different servers, namespaces etc.) that were not considered in prior approaches. RDF is simple and therefore acceptable for a variety of user communities, and extensible in the sense that the user communities are able to adapt the language to their needs and layer more advanced functionality on top.

TRIPLE can import facts encoded in RDF and OWL ontologies, since OWL is layered on top of RDF. Rules can be used to perform operations on the data available in TRIPLEs knowledge base.

Both rules and facts can be grouped together using a "context", which allows for modularizing TRIPLE programs. Contexts, or models, are identified using URIs, and can have input parameters - variables or even

other contexts. Reasoning is then performed on the facts and rules in both the base model and the input model(s). Figure 1 shows how a model named `http://example.org/model1` is applied to a model called `owl_lite_` with parameter `Mdl`. The result is a model with the identifier `owl_lite_(http://example.org/model1)` that contains the statements from both source models together with inferred statements.

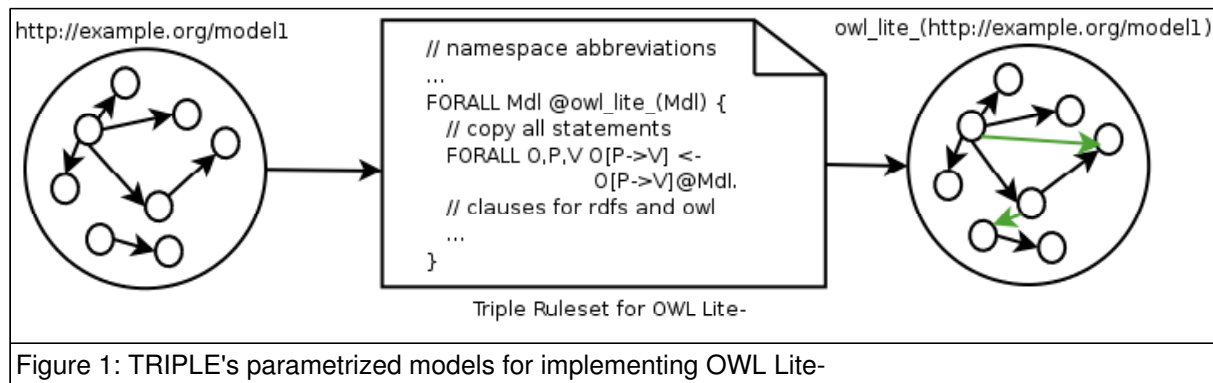


Figure 1: TRIPLE's parametrized models for implementing OWL Lite-

Since ontologies can be formally described using facts and rules, TRIPLE can reason with ontologies given that there is such a formal description available. Part of this deliverable is to develop a formal description for the ontology language OWL Lite- to enable instance and schema inferences using TRIPLE.

4 A Ruleset for OWL Lite-

In the previous section, we described the ontology language OWL, which is based on description logics, and presented the reasoning system TRIPLE, that uses evaluation strategies for Horn logics to draw conclusions from facts and rules. [Grosz et al., 2003] show that it is possible in principle to interoperate between OWL and rules. In the following, we present a number of TRIPLE rules that implement a sublanguage of OWL, namely OWL Lite-.

OWL Lite- has a number of restrictions to allow for efficient reasoning for ontologies. The constructs for class intersection and data types are not supported in OWL Lite- and therefore not listed. The abstract syntax construct "IntersectionOf" is not supported in OWL Lite-, but the RDF syntax construct `owl:intersectionof` is supported in complete class definitions. Notice that in complete class definitions, only named classes are allowed in the definition. Header information, versioning, and annotation properties are omitted in our listing since they are not part of the model theoretic semantics of OWL Lite-. However, one could import the OWL RDF description to allow a more complete coverage of the OWL standard.

On the other hand, the approach followed here allows one to use the expressive power of rules in combination with ontological definitions for vocabulary primitives. Rules allow for powerful clause constructs that are not expressible in OWL, for example complex view definitions.

We start with an overview of the constructs allowed in OWL Lite- following the presentation style of [McGuinness & van Harmelen, 2004]. We assume the well-known namespace abbreviations for `rdf`, `rdfs`, and `owl`. In subsequent sections, we present a listing with the TRIPLE rules and discuss each rule in detail. An example that demonstrates reasoning on with an OWL Lite- ontology concludes the chapter.

4.1 OWL Lite- Constructs

The list of OWL Lite- language constructs that is used in our ruleset is given below. In the TRIPLE listing, we lack a rule for complete class definitions (`owl:intersectionOf`) because our reasoner lacks functionality to handle blank nodes and `rdf:parseType="Collection"`, which are needed to encode complete class definitions in OWL/RDF serialization.

RDF Schema Features	Property Characteristics
<ul style="list-style-type: none"> • Class • <code>rdfs:subClassOf</code> • <code>rdf:subPropertyOf</code> • <code>rdfs:domain</code> • <code>rdfs:range</code> 	<ul style="list-style-type: none"> • ObjectProperty • TransitiveProperty • SymmetricProperty • inverseOf

Table 1: List of OWL Lite- language constructs

Equivalence	Property Restrictions
<ul style="list-style-type: none"> • equivalentProperty • equivalentClass 	<ul style="list-style-type: none"> • Restriction • onProperty • allValuesFrom

Table 1: List of OWL Lite- language constructs

OWL Lite- has classes (owl:Class), restrictions (owl:Restriction), properties (owl:ObjectProperty), and individuals that have properties and belong to a class specified using the rdf:type property.

For a detailed description of OWL language constructs we refer the interested reader to [\[McGuinness & van Harmelen, 2004\]](#).

4.2 TRIPLE Listing

Figure 2 shows the TRIPLE ruleset. Each clause is described in detail in the following sections. Readers are advised to view the TRIPLE listing in parallel with the descriptions to facilitate their understanding of the coordination between listing and explanations. Comments in the listing start with a double forward slash (//) and are not processed by the reasoning engine.

```
// namespace abbreviations
(1) rdf := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
(2) rdfs := 'http://www.w3.org/2000/01/rdf-schema#'.
(3) owl := 'http://www.w3.org/2002/07/owl#'.

(4) FORALL Mdl @owl_lite_(Mdl) {
// copy all facts from Mdl to current model
(5) FORALL X,Y,Z X[Y->Z] <- X[Y->Z]@Mdl.

// ----- rdfs class axioms -----
// rdfs:subClassOf
(6) FORALL O,V O[rdfs:subClassOf->V] <-
    EXISTS W (O[rdfs:subClassOf->W]
    AND W[rdfs:subClassOf->V]).
(7) FORALL O,T O[rdf:type->T] <-
    EXISTS S (S[rdfs:subClassOf->T]
    AND O[rdf:type->S]).

// ----- rdfs property axioms -----
// rdfs:subPropertyOf
(8) FORALL O,V O[rdfs:subPropertyOf->V] <-
    EXISTS W (O[rdfs:subPropertyOf->W]
    AND W[rdfs:subPropertyOf->V]).
(9) FORALL O,P,V O[P->V] <-
    EXISTS S (S[rdfs:subPropertyOf->P]
    AND O[S->V]).

// rdfs:domain DL-style
(10) FORALL O,T O[rdf:type->T] <-
    EXISTS P,V (O[P->V] AND P[rdfs:domain->T]).
// rdfs:range DL-style
(11) FORALL V,T V[rdf:type->T] <-
    EXISTS O,P (O[P->V] AND P[rdfs:range->T]).

// ----- owl property axioms -----
// owl:ObjectProperty has range and domain owl:Class
(12) owl:ObjectProperty[rdfs:domain->owl:Class].
(13) owl:ObjectProperty[rdfs:range->owl:Class].

// owl:TransitiveProperty
(14) FORALL O,P,V O[P->V] <-
    EXISTS W (O[P->W] AND W[P->V])
    AND P[rdf:type->owl:TransitiveProperty].

// owl:SymmetricProperty
(15) FORALL O,P,V V[P->O] <-
    O[P->V]
    AND P[rdf:type->owl:SymmetricProperty].

(16) FORALL P,V P[rdfs:domain->V] <-
    P[rdfs:type->owl:SymmetricProperty]
    AND P[rdfs:range->V].

(17) FORALL P,V P[rdfs:range->V] <-
    P[rdf:type->owl:SymmetricProperty]
    AND P[rdfs:domain->V].

// owl:inverseOf
```

Figure 2: The TRIPLE Ruleset

```

(18) owl:inverseOf[rdfs:domain->owl:ObjectProperty].
(19) owl:inverseOf[rdfs:range->owl:ObjectProperty].
(20) owl:inverseOf[rdf:type->owl:SymmetricProperty].
(21) FORALL O,P,V V[P->O] <-
      EXISTS Q (O[Q->V] AND P[owl:inverseOf->Q]).

// ----- equivalence -----

// owl:equivalentProperty
(22) owl:equivalentProperty[rdf:type->owl:SymmetricProperty].
(23) owl:equivalentProperty[rdf:type->owl:TransitiveProperty].
(24) FORALL O,P,V O[P->V] <-
      EXISTS Q (P[owl:equivalentProperty->Q] AND O[P->V]).
(25) FORALL O,Q,V O[Q->V] <-
      EXISTS P (P[owl:equivalentProperty->Q] AND O[P->V]).
(26) FORALL N,P,V N[P->V] <-
      EXISTS O (N[owl:equivalentProperty->O] AND O[P->V]).
(27) FORALL O,P,W O[P->W] <-
      EXISTS V (V[owl:equivalentProperty->W] AND O[P->V]).

// owl:equivalentClass
(28) owl:equivalentClass[rdf:type->owl:SymmetricProperty].
(29) owl:equivalentClass[rdf:type->owl:TransitiveProperty].
(30) FORALL O,T O[rdf:type->T] <-
      EXISTS S (T[owl:equivalentClass->S]
                AND O[rdf:type->S]).
(31) FORALL O,P,V O[P->V] <-
      EXISTS Q (P[owl:equivalentClass->Q] AND O[P->V]).
(32) FORALL O,Q,V O[Q->V] <-
      EXISTS P (P[owl:equivalentClass->Q] AND O[P->V]).
(33) FORALL N,P,V N[P->V] <-
      EXISTS O (N[owl:equivalentClass->O] AND O[P->V]).
(34) FORALL O,P,W O[P->W] <-
      EXISTS V (V[owl:equivalentClass->W] AND O[P->V]).

// ----- property restrictions -----

// owl:onProperty, owl:allValuesFrom
(35) FORALL O,R,P O[rdf:type->R] <-
      EXISTS V O[P->V]
            AND R[rdf:type->owl:Restriction]
            AND R[owl:onProperty->P]
            AND EXISTS C R[owl:allValuesFrom->C].
(36) }

```

Figure 2: The TRIPLE Ruleset

4.3 TRIPLE Header and Model Definitions

(1) defines the namespace for the abbreviation rdf as <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

(2) defines the namespace for the abbreviation rdfs as <http://www.w3.org/2000/01/rdf-schema#>.

(3) defines the namespace for the abbreviation owl as <http://www.w3.org/2002/07/owl#>.

(4) defines the model owl_lite_. The model owl_lite_ has parameter Mdl which contains the ground facts of the OWL Lite- ontology to which the rule set will be applied. owl_lite_ applied to the ground facts in Mdl results in a set of triples - the target model - that contains the ground facts and the inferred statements. The opening curly bracket denotes the start of a model block, which has its accompanying closing curly bracket at (36).

(5) copies all statements (ground facts) from the model specified as parameter Mdl to the target model. That is passed to the model owl_lite_ as a parameter

4.4 RDF Schema Features

rdfs:subClassOf: The rdfs:subClassOf property can be used to model class hierarchies. The rule specified in the following allows a reasoner to make assumptions about the class hierarchy and infer new type information for Individuals based on the hierarchy specified with rdfs:subClassOf.

(6) defines the transitivity of rdfs:subClassOf. Transitivity in the class hierarchy means that if Primates are a subclass of Mammals, and Humans are a subclass of Primates, then Humans are a subclass of Mammals. More formally: if O is a subclass of W, and W is a subclass of V, then O is a subclass of V.

(7) The type of an individual must take into account class hierarchies. That means if we have the above hierarchy, and there is an individual Joe that is of type Human, then a rule must infer that Joe is of both type Mammal and type Primate. Transitivity of rdfs:subClassOf is defined in rule (6), which takes care of expanding the class hierarchy. Therefore, the rule has only to infer that if S is a subclass of T, then O is of type T.

rdfs:subPropertyOf: rdfs:subPropertyOf is similar to rdfs:subClassOf, but enables in contrast to defining a hierarchy of subclasses the specification of a hierarchy of properties. A property hasCar can be a subproperty of hasMotorVehicle, which in turn can be a subproperty of hasVehicle. Exactly as rdfs:subClassOf, rdfs:subProperty is transitive, which means that if an individual has a property hasCar then the individual should also have the property hasMotorVehicle and hasVehicle with the same value.

(8) specifies the transitivity of rdfs:subPropertyOf, which means that if hasCar is subproperty of hasMotorVehicle, and hasMotorVehicle is subclass of hasVehicle, then hasCar is also subproperty of hasVehicle. More formally: if O is a subproperty of W, and W is a subproperty of V, then O is a subproperty of V.

(9) applies the property hierarchy to individuals that are part of the knowledge base, which, given our previous example, means that if there is an individual that has the property hasMotorVehicle it must also have the property hasVehicle with the same value. If S is a subproperty of P, and S is a property of O, then O has property P.

rdfs:domain: The domain of a property can be restricted to a class. If you want to say that only Humans can own Vehicles, then you can define the domain of a property to be of class Human.

(10) The description logic semantics requires in turn that if an individual has a certain property, the individual must be of the type of the properties' domain. For example, if you state that Joe hasVehicle Bike, then you can infer that Joe is of type Human. Rule (10) implements that behaviour: if O has property P, and P is of domain T, then O must be of type T.

rdfs:range: The range of a property can be specified similar to its domain. You can say that the object of the property hasVehicle has to be of type vehicle by restricting the range of a property to the class vehicle.

(11) rdfs:range is implemented with description logic semantics similar to rdfs:domain. That means if an individual appears as object of a property that has a range restriction, then the object has the type of the class specified in the range restriction. In our example, if you state that Joe hasVehicle Bike, then the rule to infer that Bike is of type vehicle is: if O has property P, and P is of range T, then O must be of type T.

4.5 Property Characteristics

owl:ObjectProperty: OWL allows an owl:ObjectProperty where its subject and object must be individuals, as opposed to owl:DatatypeProperty, which can have data types such as string or integer as objects.

(12) specifies that the domain of owl:ObjectProperty is owl:Class, which means that the subject of the property must be of type owl:Class. Rule (10) covers the inference that the subject of a triple with a property that has a given range is of a certain class. In our example, if hasVehicle is of type owl:ObjectProperty, then every subject of hasVehicle is of type owl:Class, which makes every subject of a statement with an owl:ObjectProperty to an individual.

(13) performs the same operation for the range as (12) does for the domain: the object of an owl:ObjectProperty must be of type owl:Class.

owl:TransitiveProperty: Transitivity of properties has been exhaustively covered in the discussion of rdfs:subClassOf and rdfs:subPropertyOf. We refer the interested reader to rules (6) and (8).

(14) is a rule that generalizes transitivity for properties of type owl:TransitiveProperty: if O has property P with value W, and W has property P with value V, then O must have property P with value V.

owl:SymmetricProperty: A symmetric property is true in both directions. Friendship is an example for a property that is usually symmetric: you are also your friend's friend.

(15) owl:SymmetricProperty defines a symmetric property with a rule that basically switches the subject and object of statements: if O has property P and value V, then V must have property P with value O.

(16) and (17) encode a restriction: since subject and object can be exchanged for properties of type owl:SymmetricProperty, the rdfs:domain and rdfs:range values for an owl:SymmetricProperty are the same.

owl:inverseOf: With owl:inverseOf, two properties can be declared inverse. For example, consider two inverse relations, hasChild and hasParent. The inference drawn in combination with a statement Walter hasChild Andreas is that Andreas hasParent Walter.

(18) and (19) define the domain and range of owl:inverseOf as owl:ObjectProperty, that means that only owl:ObjectProperty can be inverse to each other. In our example, hasChild and hasParent have to be of type owl:ObjectProperty, which is taken care of by the inference rules (10) and (11).

(20) states that owl:inverseOf is a owl:SymmetricProperty. That means that if hasChild is the owl:inverseOf hasParent, then hasParent must be the owl:inverseOf hasChild. Inference rules (15), (16), and (17) draw the necessary conclusions for symmetry.

(21) defines a rule that asserts a new fact for every inverse property. In our example, if we state that Andreas hasParent Walter, and hasChild is the owl:inverseOf hasParent, then Walter hasChild Andreas is asserted. The same thing with variables: if O has property Q and value V, and P is the inverse of Q, then V must have property P with value O.

4.6 Equivalence

owl:equivalentProperty: Assume two ontologies, say foaf and kw, and both ontologies specify different names for an equivalent property. For example, foaf contains a property foaf:name, and kw contains kw:FullName. Then, you may state that foaf:name and kw:FullName are equivalent properties, that is that every individual that has the property foaf:name also has the property kw:FullName with the same value and vice versa.

(22) Since equivalence is a symmetric property (if foaf:name is equivalent to kw:FullName, then kw:FullName is also equivalent to foaf:name), we state the symmetry here. Rules (15), (16), and (17) draw the conclusions required for symmetry.

(23) states that owl:equivalentProperty is of type owl:TransitiveProperty. The general rule that entails transitivity is (14).

(24) Because equivalence is reflexive, we include a rule that states reflexive behaviour. Formally, if there is a statement with subject O, predicate P, and object V, and P is the equivalentProperty of Q, then entail the statement with subject O, predicate P, and object V.

(25) entails statements with equivalent properties. If there is an individual with the property foaf:name, then that individual has also the property kw:FullName with the same value. Or simply put: If O has property W and value V, and P is equivalent property of W, then O must have property P and value V.

(26), (27) All occurrences of owl:equivalentProperties in either subject, predicate, or object positions have to be replaced. These two rules take care of the replacements of owl:equivalentProperties in subject and object positions of an RDF statement.

owl:equivalentClass:The construct owl:equivalentClass can be used to perform basic schema mapping functionality. Imagine two ontologies foaf and kw that both describe the concept Person. Then, you can state that the classes foaf:Person and kw:Person are equivalent. That means that every individual of the one class is also an individual of the other class and vice versa.

(28) Since equivalence is a symmetric property (if foaf:Person is equivalent to kw:Person, then kw:Person is also equivalent to foaf:Person), we state the symmetry in this clause. Rules (15), (16), and (17) entail the necessary statements for symmetry.

(29) states that owl:equivalentClass is of type owl:TransitiveProperty. The general rule that entails transitivity is (14).

(30) performs the required inferences on the type hierarchy, and infers that an individual of one class that is equivalent to the other class is also an individual of the other class. More formally: If T and S are equivalent classes, and O is of type S, then O must be of type T.

(31) Because equivalence is reflexive, we include a rule that states reflexive behaviour. Formally, if there is a statement with subject O, predicate P, and object V, and P is the equivalentClass of Q, then entail the statement with subject O, predicate P, and object V.

(32), (33), (34) All occurrences of owl:equivalentClass in either subject, predicate, or object positions have to be replaced. These three rules take care of the replacements of owl:equivalentClass in all positions of an RDF statement.

4.7 Property Restrictions

owl:onProperty, owl:allValuesFrom: The combination of the constructs owl:onProperty and owl:allValuesFrom poses a restriction to a property with respect to a class (local range restriction). The semantics is similar to rdfs:range, but takes into account not only the property but also the domain of the statement. For example, you can say that Stefan manages Wolf, and Wolf is of type Student implies that Stefan is of class Advisor.

(35) draws the necessary inference for the local range restriction. The rule is composed of a number of constraints; the restriction must be of class owl:Restriction, and must have owl:onProperty and a owl:allValuesFrom. An individual satisfying all conditions has to be an instance of the restriction class. Simply: A statement with O,P,V, and R is of type owl:Restriction, and R is on property P and R allows all values from C implies O is of type R.

4.8 Example

The following illustrates how the above reasoning rules can be applied to ontologies.

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:ex="http://example.org/ex#">

  <owl:Ontology rdf:about="" />

  <rdf:Description rdf:ID="andreas">
    <ex:worksAt rdf:resource="#swc" />
  </rdf:Description>

  <ex:Student rdf:ID="wolf">
    <ex:worksAt rdf:resource="#swc" />
    <ex:advisedBy rdf:resource="#stefan" />
  </ex:Student>

  <rdf:Description rdf:ID="stefan">
    <ex:leads rdf:resource="#swc" />
    <ex:manages rdf:resource="#andreas" />
  </rdf:Description>

  <owl:ObjectProperty rdf:ID="worksAt">
    <rdfs:domain rdf:resource="#Employee" />
    <rdfs:range rdf:resource="#Department" />
    <owl:inverseOf rdf:resource="#hosts" />
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="leads">
    <rdfs:subPropertyOf rdf:resource="#worksAt" />
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="manages">
    <rdfs:domain rdf:resource="#Manager" />
    <rdfs:range rdf:resource="#Student" />
    <owl:inverseOf rdf:resource="#advisedBy" />
    <owl:equivalentProperty rdf:resource="#advises" />
  </owl:ObjectProperty>

  <owl:Class rdf:ID="Advisor">
    <rdfs:subClassOf rdf:resource="#AdvisorRestriction" />
  </owl:Class>

  <owl:Restriction rdf:ID="AdvisorRestriction">
    <owl:onProperty rdf:resource="#manages" />
    <owl:allValuesFrom rdf:resource="#Student" />
  </owl:Restriction>
</rdf:RDF>

```

Figure 3: OWL Lite- Ontology Example

Performing reasoning on the OWL ontology in Figure 3 together with the OWL Lite- rules described, we get a number of new facts that were inferred during the reasoning step. Figure 4 lists the new statements entailed during OWL Lite- reasoning over the ontology in Figure 3.

```

ex:stefan rdf:type ex:Employee .
ex:stefan rdf:type ex:Manager .
ex:stefan rdf:type ex:Advisor .
ex:stefan ex:worksAt ex:swc .
ex:stefan ex:advises ex:andreas .
ex:stefan ex:advises ex:wolf .
ex:stefan ex:manages ex:wolf .
ex:wolf rdf:type ex:Employee .
ex:andreas rdf:type ex:Employee .
ex:andreas rdf:type ex:Student .
ex:andreas ex:advisedBy ex:stefan .

```

Figure 4: Entailed statements after OWL Lite- reasoning (control statements omitted)

```

ex:swc rdf:type ex:Department .
ex:swc ex:hosts ex:stefan .
ex:swc ex:hosts ex:wolf .
ex:swc ex:hosts ex:andreas .
ex:advises owl:equivalentProperty ex:manages .
ex:advisedBy rdf:type owl:ObjectProperty .
ex:advisedBy owl:inverseOf ex:manages .
ex:leads rdfs:subProperty ex:worksAt .
ex:hosts rdf:type owl:ObjectProperty .
ex:hosts owl:inverseOf ex:worksAt .

```

Figure 4: Entailed statements after OWL Lite- reasoning (control statements omitted)

5 TRIPLE Architecture and Implementation

The following sections give an architectural overview of TRIPLE and describe implementation details. The current version of TRIPLE can be downloaded from the TRIPLE homepage at <http://triple.semanticweb.org/>.

5.1 Overview

TRIPLE parses and translates facts and rules encoded in the TRIPLE language into Logic Programs (LP) that can be efficiently evaluated using the logic programming and deductive database system XSB. Facts encoded in RDF can be imported as well. Queries are transformed into LP queries; answers from XSB are translated back to facts encoded in RDF. Figure 5 shows a simplified view of TRIPLE's architecture.

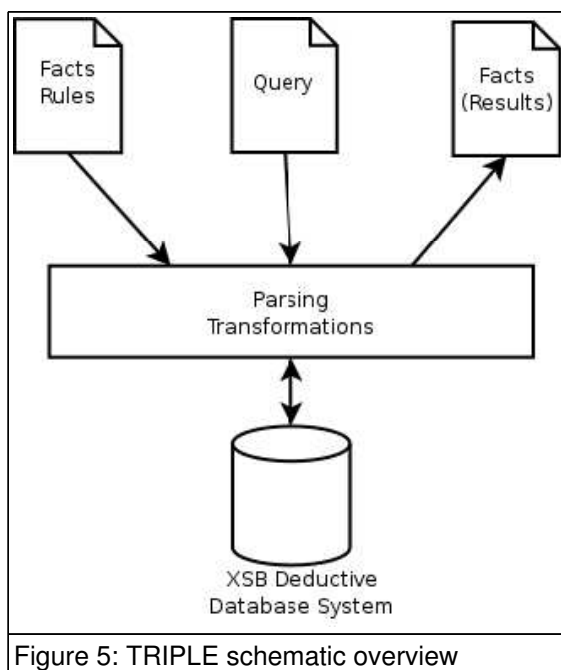


Figure 5: TRIPLE schematic overview

5.2 Java API

The basic operations that TRIPLE supports are add facts and rules, pose queries, and remove facts and rules from the knowledge base. Java programmes can access TRIPLE via a Java interface and integrate TRIPLE functionality. For a detailed description of the API we refer the reader to the [Appendix](#).

The Java interface offers the following methods.

- Adding operations can be performed on strings, files, or over the network on URIs. Facts and rules encoded in TRIPLE syntax can be added, as well as facts encoded in RDF.
- Queries are encoded in TRIPLE syntax, and return either variable bindings or a set of RDF statements.
- Individual facts and rules can be removed, as well as the content of models.

5.3 TRIPLE Server

In addition to the Java API, we have implemented a TRIPLE server that can be accessed via HTTP [[Berners-Lee et al., 1998](#)]. TRIPLE's basic operations such as adding facts and rules, posing queries, and removing facts and rules from the knowledge base are supported over HTTP. The interface is available

online at <http://triple.semanticweb.org:3030/> for the reader's own experiments.

Clients and APIs for HTTP are available for almost every programming language. Thus, programmers can use the online TRIPLE service for embedding reasoning functionality in own programs.

We suggest the following content-types that determine the format of the data that is exchanged between the TRIPLE server and a client.

- `x-application/triple`: Content type for files in TRIPLE syntax. Note that depending on the content of the files (i.e. if they contain a query), there might be some return value.
- `application/rdf+xml`: RDF/XML format as described in the RDF specification. `application/rdf+xml` should become an officially (IETF) registered content-type soon.
- `x-application/rdf+n3`: Content type for files using the Notation3 format [[Berners-Lee, 2001](#)].

Similar to the Java API, the following methods are supported for interacting with the online TRIPLE service:

- `/add` over HTTP GET, where the "uri" parameter specifies the URI of the file to be added. Additional parameters are "ns" and "localname" that specify namespace and localname respectively, in case RDF/XML is added. Type information is sent using the content-type header with either `x-application/triple`, `application/rdf+xml`, or `x-application/rdf+`. The method returns HTTP status 200 (OK) when successful.
- `/query` over HTTP GET, where the "uri" parameter denotes the URI of the query file. The results are returned in RDF/XML. Note that the query has to return (subject, predicate, object) variables with URIs to allow a serialization to RDF/XML. The return value of HTTP status is 200 (OK) when the query operation was successful.
- `/remove` over HTTP GET, where the "ns" and "localname" parameters specify the namespace and localname part of the model to be deleted. Returns HTTP status 200 (OK) when successful.

Please note that the remote interface as described here is an initial version and may change according to emerging standards or best-practice on how to remotely access reasoning systems and data stores.

5.4 Parsing and Transformations

The parsing and transformation steps encompass most of TRIPLES functionality. During this stage, arbitrary TRIPLE formulas are parsed and converted into normal form horn clauses which are a suitable format for evaluation by XSB.

Parsing. First, the parser reads the TRIPLE program into a parse tree, according to the TRIPLE grammar which can be found in the [Appendix](#). The main parser class is `LTermParser`. The parse tree consists of a number of clauses which are encoded in `LTerm` objects, together with the model in which they appear, which is stored in the `Context` object. The parser works in a streaming fashion and emits `LTerm` objects, which are processed by callback objects that are invoked on a clause-by-clause basis. Callbacks perform the translation steps described in the following sections. Possible callback objects are `ExternalPrologCallback` which calls `XSB` or `PrintCallback` for debugging purposes.

Parse Tree to TRIPLE Language. Next, `TripleTransform` takes in `LTerms` of clauses together with the model also represented as an `LTerm` object, and translates them to `language.Clause` objects. There are also `Formula`, `HornAtoms`, and `Terms` in the `language.*` package, but those are not utilized in the code.

TRIPLE Language to General Logic Programming. In the next step, the `language.Clause` objects are translated to `glp.GLPRule` objects, using the `toGLP` method on `language.Clause`. All `language.*` classes have the `toGLP` method that serializes the objects into `glp.*` classes.

General Logic Programming to Logic Programming. As a final step, the `glp.GLPRule` objects are then transformed using the Lloyd-Topor transformation in the `GLPLloydToporTransform` class. Lloyd-Topor transformations are a set of rewrite rules to translate a general logic program [[Lloyd, 1987](#)] into conjunctive normal form [[Decker, 2002](#)]. The result of the transformation is an `ArrayList` of `lp.ProgramClause` objects, together with the variable mappings stored in an `ArrayList`. Both objects are finally passed to `XSB` for evaluation.

Results Transformation. Queries are translated to LP analog to facts and queries. `XSB` returns the results of queries in `ArrayLists` of variable bindings, embodied in `ResultSet` objects. If the queries return three variables that can be interpreted as (subject predicate object) statements in RDF, it is possible to convert the result into an RDF representation. Using RDF as the result format for queries can be a first step in allowing distributed reasoning.

5.5 XSB Deductive Database

XSB is a Logic Programming and Deductive Database system (Tabled Prolog) for Unix and Windows, available under an open source license. XSB is used as a backend engine that performs reasoning on logic programs that are emitted after TRIPLE performed the transformation steps.

TRIPLE is written in Java, and uses XSB (written in C) as an evaluation engine. The Java Native Interface (JNI) connects the Java layer of TRIPLE with the XSB evaluation engine. We use a Java Virtual Machine (JVM) running the Java layer, and another JVM running the XSB and JNI code that communicate via RMI. Running both parts in one JVM results in random crashes of TRIPLE probably due to XSB memory management.

6 Conclusions

In this document, we have described how to use a rule-based system for reasoning with ontologies. We have presented requirements for a reasoning system on the Web. A reasoning system used in the context of the web poses different requirements than a centralized reasoner. We identified language layering, scalability, and a solid implementation as the main requirements to successfully apply a reasoner on the Web. We briefly described OWL and how OWL related to rules-based systems. Next, we described what language constructs can be expressed using Horn rules, and defined a rule set that encodes the semantics of OWL Lite-. Reasoners with the ability to process Horn rules can perform ontology-based reasoning by importing the rule set we have presented.

References

- [Abiteboul et al., 2000]** Serge Abiteboul, Peter Bruneman, and Dan Suciu. "Data on the Web". Morgan Kaufmann Publishers, San Francisco, California, 2000.
- [Berners-Lee et al., 1998]** Tim Berners-Lee, Roy Fielding, and L. Masinter. "Uniform Resource Identifiers (URI): Generic Syntax". Technical Report RfC 2396, IETF, 1998.
- [Berners-Lee, 2001]** Tim Berners-Lee. "Notation 3, an RDF Language for the Semantic Web". W3C Design Issues, <http://www.w3.org/DesignIssues/Notation3.html>
- [de Bruijn & Fensel, 2004]** Jos de Bruijn and Dieter Fensel. "WSML Deliverable D20 v0.1 OWL Light-". WSML Working Draft May 31, 2004.
- [Decker, 2002]** Stefan Decker. "Semantic Web Methods for Knowledge Management". Dissertation University of Karlsruhe , 2002.
- [Grosz et al., 2003]** Benjamin Grosz, Ian Horrocks, Raphael Volz, Stefan Decker. "Description Logic Programs: Combining Logic Programs with Description Logic". World Wide Web Conference, Budapest , May 20-24, 2003.
- [Gruber, 1993]** T. R. Gruber. "A translation approach to portable ontologies". Knowledge Acquisition, 5(2): 199-220, 1993.
- [Lloyd, 1987]** J. W. Lloyd. "Foundations of Logic Programming". Second, Extended Edition. Springer Verlag, 1987.
- [McGuinness & van Harmelen, 2004]** Deborah L. McGuinness and Frank van Harmelen (eds.). "OWL Web ontology Language Overview". W3C Recommendation 10 February 2004.
- [Patel-Schneider et al., 2004]** Peter F. Patel-Schneider, Patrick Hayes, Ian Horrocks. "OWL Web Ontology Language Semantics and Abstract Syntax". W3C Recommendation 10 February 2004.
- [Sintek & Decker, 2002]** Michael Sintek, Stefan Decker. "TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web". International Semantic Web Conference (ISWC), Sardinia , June 2002.
- [Wos et al., 1992]** Larry Wos, Ross Overbeek, Ewing Lusk, Jim Boyle. "Automated reasoning: Introduction and Applications" McGraw Hill 1992.

Acknowledgement

The work is funded by the European Commission under the projects [DIP](#), [Knowledge Web](#), [SEKT](#), [SWWS](#), and [Esperanto](#); by [Science Foundation Ireland](#) under the DERI-Lion project; and by the Vienna city government under the [CoOperate](#) program.

The editors would like to thank to all the [members of the WSML working group](#) for their advice and input into this document.

Thanks to Wolf Winkler for suggesting the ontology example and help in creating the diff between the ontologies.

Disclaimer

The opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Science Foundation Ireland, NUI, Galway or the Hewlett-Packard Company.

Appendix

TRIPLE BNF Grammar

parse	::=	((ClauseSeq)? <EOF>)
ClauseBlock	::=	(((ForallQuantifier)? <AT> StructTerm)? SimpleClauseBlock)
SimpleClauseBlock	::=	("{" ClauseSeq "}")
ClauseSeq	::=	(Clause (ClauseSeq)?)
Clause	::=	(ClauseBlock ((ForallQuantifier)? Term <EOC>))
ForallQuantifier	::=	(<FORALL> IdTermSeq)
Term	::=	Op1200Term
Op1200Term	::=	((((Unop1200)? Op1100Term) (Binop1200 Op1200Term)?)
Op1100Term	::=	(Op1000Term (Binop1100 Op1100Term)?)
Op1000Term	::=	(Op900Term (Binop1000 Op1000Term)?)
Op900Term	::=	(((Unop900 Op900Term) (Quantop900 IdTermSeq Op900Term) Binop900Term)
Binop900Term	::=	(Op700Term (Binop900 Binop900Term)?)
Op700Term	::=	(Op680Term (Binop700 Op700Term)?)
Op680Term	::=	(Op661Term (Binop680 Op680Term)?)
Op661Term	::=	(Op500Term (Binop661 Op661Term)?)
Op500Term	::=	((((Unop500)? Op400Term) (Binop500 Op500Term)?)
Op400Term	::=	(StructTerm (Binop400 Op400Term)?)
StructTerm	::=	(UnitTerm (ArgList SBArgList)*)
UnitTerm	::=	(PathExpression Integer Double "(" Term ")" "<" Term ">")
PathExpression	::=	(IdTerm) ("." IdTerm)*
IdTerm	::=	((Variable Symbol) (<COLON> IdTerm)?)
Variable	::=	("?" <SYMBOL>)
Symbol	::=	(<SYMBOL> <Q_SYMBOL> <DQ_SYMBOL>)
Integer	::=	(<INTEGER_LITERAL>)
Double	::=	(<FLOATING_POINT_LITERAL>)
ArgList	::=	("(" TermSeq ")")
SBArgList	::=	("[" TermSeq "]")
TermSeq	::=	(Op900Term ((<COMMA> <SEMICOLON>) Op900Term)*)
IdTermSeq	::=	(IdTerm (<COMMA> IdTerm)*)

Unop1200	::=	(<IMPLIEDBY>)
Binop1200	::=	(<IMPLIEDBY> <EQUIV> <ASSIGN>)
Binop1100	::=	(<SEMICOLON> <OR>)
Binop1000	::=	(<COMMA> <AND>)
Quantop900	::=	(<FORALL> <EXISTS>)
Unop900	::=	(<NOT> <NEG> <EXTERNAL>)
Binop900	::=	(<IMPLIES>)
Binop700	::=	(<GEQ> <EQUALS> <IS> <STRUCTEQUALS> <LESSER> <GREATER> <LEQ>)
Binop680	::=	(<AT>)
Binop661	::=	(<DOT>)
Unop500	::=	(<PLUS> <MINUS>)
Binop500	::=	(<PLUS> <MINUS>)
Binop400	::=	(<TIMES> <BY> <INTERSECT> <UNION> <DIFF>)

TRIPLE API Javadoc

org.semanticweb.triple.application Interface TripleService

All Known Implementing Classes:

TripleServiceImpl

public interface **TripleService**

TRIPLE Service API

Author:Stefan Decker, Andreas Harth

Method Detail

initTriple

```
public void initTriple()
    throws XSBException
```

Init. Loads XSB engine.
XSBException

closeTriple

```
public void closeTriple()
    throws XSBException
```

Cleanup. Does nothing right now.
XSBException

addTripleFile

```
public void addTripleFile(java.lang.String fileName)
    throws java.io.FileNotFoundException,
           ParseException,
           XSBException
```

Compiles the TRIPLE definitions contained in the file and adds the content incrementally to the internal Knowledge Base.

java.io.FileNotFoundException
ParseException

addTripleString

```
public void addTripleString (java.lang.String rules)
    throws ParseException,
           XSBException
```

Compiles the TRIPLE definitions contained in the string and adds the content incrementally to the internal Knowledge Base.

ParseException
XSBException

addTripleStream

```
public void addTripleStream (java.io.InputStream ruleStream)
    throws ParseException,
           XSBException
```

Compiles the TRIPLE definitions provided by the InputStream and adds the content incrementally to the internal Knowledge Base.

ParseException
XSBException

addRDFFile

```
public void addRDFFile (java.lang.String filename,
    java.lang.String ns,
    java.lang.String localname)
    throws java.io.FileNotFoundException,
           org.xml.sax.SAXException,
           java.io.IOException,
           XSBException
```

Compiles the RDF definitions provided by the file and adds the content incrementally to the RDF model whose URI consists of the provided namespace and localname.

java.io.FileNotFoundException
org.xml.sax.SAXException
java.io.IOException
XSBException

addRDFFile

```
public void addRDFFile (java.lang.String filename,
    java.lang.String ns,
    java.lang.String localname,
    java.lang.String appendix)
    throws java.io.FileNotFoundException,
           org.xml.sax.SAXException,
           java.io.IOException,
           XSBException
```

java.io.FileNotFoundException
org.xml.sax.SAXException
java.io.IOException
XSBException

removeRDFFile

```
public void removeRDFFile (java.lang.String ns,
    java.lang.String localname,
    java.lang.String appendix)
    throws java.io.FileNotFoundException,
           org.xml.sax.SAXException,
           java.io.IOException,
           XSBException
```

java.io.FileNotFoundException
org.xml.sax.SAXException
java.io.IOException

removeN3File

```
public void removeN3File (java.lang.String ns,  
                          java.lang.String localname,  
                          java.lang.String appendix)  
    throws java.io.FileNotFoundException,  
           org.xml.sax.SAXException,  
           java.io.IOException,  
           XSBEException
```

java.io.FileNotFoundException
org.xml.sax.SAXException
java.io.IOException
XSBEException

addRDFString

```
public void addRDFString (java.lang.String rdfString,  
                          java.lang.String nameSpace,  
                          java.lang.String localName)  
    throws org.xml.sax.SAXException,  
           java.io.IOException,  
           XSBEException
```

Compiles the RDF definitions provided by the String and adds the content incrementally to the RDF model whose URI is constructed out of the provided namespace and localname.

org.xml.sax.SAXException
java.io.IOException
XSBEException

addRDFString

```
public void addRDFString (java.lang.String rdfString,  
                          java.lang.String nameSpace,  
                          java.lang.String localName,  
                          java.lang.String appendix)  
    throws org.xml.sax.SAXException,  
           java.io.IOException,  
           XSBEException
```

org.xml.sax.SAXException
java.io.IOException
XSBEException

addRDFStream

```
public void addRDFStream (java.io.InputStream rdfStream,  
                          java.lang.String nameSpace,  
                          java.lang.String localName)  
    throws org.xml.sax.SAXException,  
           java.io.IOException,  
           XSBEException
```

Compiles the RDF definitions provided by the InputStream and adds the content temporarily to the internal Knowledge Base into the RDF model with an URI composed of the provided namespace and localname parameters.

org.xml.sax.SAXException
java.io.IOException
XSBEException

removeModel

```
public void removeModel (java.lang.String nameSpace,  
                          java.lang.String localName)  
    throws XSBEException
```

Deletes the content of the RDF model whose ID is given by the namespace and the localname.
XSBEException

askTripleQueryFile

```
public org.semanticweb.triple.application.ResultSet askTripleQueryFile (java.lang.String filename)
                                                    throws ParseException,
                                                    XSBException,
                                                    java.io.IOException
```

Ask a query that's stored in a file.

ParseException
XSBException
java.io.IOException

askTripleQueryString

```
public org.semanticweb.triple.application.ResultSet askTripleQueryString (java.lang.String queryString)
                                                    throws ParseException,
                                                    XSBException,
                                                    java.io.IOException
```

Ask a query that's stored in a string.

ParseException
XSBException
java.io.IOException

askTripleQueryStream

```
public org.semanticweb.triple.application.ResultSet askTripleQueryStream (java.io.InputStream queryStream)
                                                    throws ParseException,
                                                    XSBException,
                                                    java.io.IOException
```

Ask a query that's in a stream.

ParseException
XSBException
java.io.IOException

addN3File

```
public void addN3File (java.lang.String filename)
                  throws java.io.FileNotFoundException,
                  java.io.IOException,
                  XSBException
```

Compiles the N3 definitions provided by the file and adds the content incrementally to XSB.

java.io.FileNotFoundException
java.io.IOException
XSBException

addN3String

```
public void addN3String (java.lang.String n3String)
                  throws java.io.FileNotFoundException,
                  java.io.IOException,
                  XSBException
```

Compiles the N3 definitions provided by the file and adds the content incrementally to XSB.

java.io.FileNotFoundException
java.io.IOException
XSBException

addN3Stream

```
public void addN3Stream (java.io.InputStream n3Stream)
                  throws java.io.IOException,
                  XSBException
```

Compiles the RDF definitions provided by the InputStream and adds the content temporarily to the

internal Knowledge Base into the RDF model with an URI composed of the provided namespace and localname parameters.

java.io.IOException
XSException

getDetails

```
public java.lang.String getDetails(java.lang.String ns,  
                                     java.lang.String localname,  
                                     java.lang.String appendix)  
    throws java.io.IOException
```

Returns the details of an RDF file.
java.io.IOException

