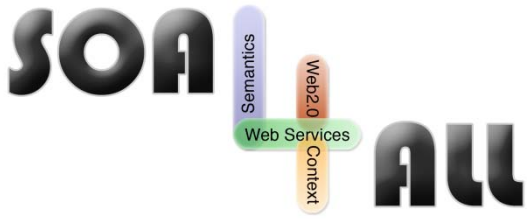

Project Number: **215219**
Project Acronym: **SOA4All**
Project Title: **Service Oriented Architectures for All**
Instrument: **Integrated Project**
Thematic Priority: **Information and Communication Technologies**

D3.2.6 Second Prototype
Rule Reasoner for WSML-Rule v2.0

Activity N: 2	Fundamental and Integration Activities	
Work Package: 3	Service Annotation and Reasoning	
Due Date:	31/08/2010	
Submission Date:	31/08/2010	
Start Date of Project:	01/03/2008	
Duration of Project:	36 Months	
Organisation Responsible of Deliverable:	UIBK	
Revision:	1.0	
Author(s):	Daniel Winkler Matthias Pressnig	UIBK UIBK
Reviewers:	Barry Norton Maria Maleshkova	UKARL OU

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**x**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	22/07/2010	First draft	Daniel Winkler
0.2	13/08/2010	Finalized version for peer review	Daniel Winkler
0.3	17/08/2010	Corrections after peer review	Daniel Winkler
1.0	25/08/2010	Final version	Daniel Winkler

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

Acronym	Definition
D	Deliverable
EC	European Commission
WP	Work Package
DL	Description Logic
LP	Logic Programming
OWL	Web Ontology Language
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology
KB	Knowledge Base
RB	Rule Base
RIF	Rule Interchange Format
RIF BLD	RIF Basic Logic Dialect
RIF DTB	Datatypes and Built-Ins
W3C	World Wide Web Consortium
DT	Datatype
DV	Data Value
DLP	Description Logic Programs
XSD	XML Schema Definition

Executive summary

In order to automate tasks such as discovery and composition, Semantic Web Services must be described in a well-defined formal language. The Web Services Modeling Language (WSML) [10] is based on the conceptual model of the Web Service Modeling Ontology [9] (WSMO) and as such can be used for modeling all aspects of Web services and associated ontologies. WSML is actually a family of several language variants, each of which is based upon a different logical formalism. The family of languages are unified under one syntactic umbrella, with a concrete syntax for modeling ontologies, web services, etc.

WSML2Reasoner is a reasoning framework that allows querying for implicit knowledge over explicit modeled WSML knowledge bases, which is of interest to various components of the SOA4All Service Delivery Platform such as the Service Location (WP5) and Service Construction (WP6) components.

This deliverable, along with others, describes the second prototype rule reasoner for WSML-Rule v2.0, in particular improvements and reconsiderations to the reasoner extensions, namely equivalences as discussed in deliverable D3.1.4 [3]. However, the main contribution described in this deliverable concerns the discussion and extension of the algorithms described in deliverable D3.2.5 [7], as well as a performance evaluation of the rule reasoner IRIS. The evaluation compares the actual time consumption with the theoretical complexity of Datalog, by creating artificial rule bases that are exponentially or linearly increased in size.

1. Introduction

The Web Service Modeling Language WSML is a formal language for the specification of ontologies and different aspects of Web services, based on the conceptual model of WSMO [9]. Several different WSML language variants exist, which are based upon different logical formalisms. The main formalisms exploited for this purpose are Description Logics (DL, [17]), Logic Programming (LP, [12]), and the intersection of these two families of logics, namely "Description Logic Programs" [16], which form the basis of WSML-DL, WSML-Flight/Rule and WSML-Core, respectively. Furthermore, WSML has been influenced by F-Logic [18] and frame-based representation systems.

Rule-based reasoning is of interest to various components of the SOA4All Service Delivery Platform: developers of the different Service Location (WP5) and Service Construction components (WP6), for which reasoning is basic infrastructure in the process of service discovery and composition. Furthermore, all use-cases (WP7, WP8 and WP9) have certain direct or indirect dependencies on the reasoning component.

Reasoning for WSML-Flight v2.0 and WSML-Rule v2.0 can be achieved in the same way as for WSML-Core v2.0 by performing the conversion steps that transform a WSML ontology to the corresponding Datalog program. In order to support the added expressivity in WSML-Flight v2.0 and WSML-Rule v2.0 the underlying Datalog reasoner needs to provide support for the required features. In particular, support for the Rule Interchange Format (RIF) built-in data types, predicates and functions [21] and for instance equivalence; i.e., *equality in rule heads* [20]. WSML-Flight v2.0 is the less expressive of the two LP-based WSML variants. Compared to WSML-Core v2.0, it adds features such as meta-modeling, constraints and non-monotonic (stratified) negation. WSML-Flight v2.0 is semantically equivalent to Datalog with equality and integrity constraints. WSML-Rule v2.0 is an extension of WSML-Flight v2.0. It adds features from Logic Programming, such as the use of function symbols, unsafe rules and unstratified negation [3].

Reasoning for these two WSML variants is realized by converting a WSML ontology to the corresponding Datalog program (with the discussed extensions) and then perform reasoning on this Datalog program using a Datalog reasoner. Therefore, the Datalog reasoner has to cover the features required by WSML-Flight v2.0 and WSML-Rule v2.0. In this deliverable, we focus on describing the necessary adaptations done to the Datalog reasoner IRIS[1].

The implementation presented in this deliverable follows the concepts and specifications that were released with deliverable D3.2.1 [4] with respect to reasoning for WSML-Rule v2.0. This document belongs to a set of related deliverables, which discuss the second prototype implementations of several WSML v2.0 variants, namely:

- D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0
- **D3.2.6 Second Prototype Rule Reasoner for WSML-Rule v2.0 (including Reasoner Framework Report [23])**
- D3.2.7 Second Prototype for Description Logic Reasoner for WSML-DL v2.0 (including Reasoner Framework Report [23])

[1] http://www.iris-reasoner.org/

1.1 Purpose and Scope

This document is a progress report on the software implementation for the second prototype rule reasoner for WSML-Flight v2.0 and WSML-Rule v2.0. The objective of the report is to provide information about the use and features of the final SOA4All prototype of the rule-based reasoning infrastructure. In particular, it explains the extensions and updates required for the instance equivalence feature, in the final implementation of the Datalog reasoner IRIS and the WSML2Reasoner reasoning framework. Previous releases exhibited several limitations and incorrect behavior in certain circumstances and some unexpected modifications were required to the reasoning algorithms.

The target audience of this report are mainly developers who wish to integrate the WSML reasoning framework into their components to model Web services and ontologies, and others who want to understand some of the issues regarding processing information represented using this formalism.

1.2 Structure of the Document

Section 2 of the deliverable discusses the actual implementation and its changes with respect to the specification. The main implementation of the prototype, as well as the algorithms extending conventional semi-naive Datalog evaluation are described in Section 3. Section 4 describes how to install and use the IRIS reasoner used for WSML Flight 2.0 and WSML-Rule v2.0 reasoning; for a description of the reasoner framework references to the reasoning framework report are given. An evaluation of the reasoning component is provided in Section 5. Section 6 concludes with a short summary of the deliverable.

2. Reflection of the Specification

For compatibility reasons, the rule-based reasoner supports instance equality, which allows the inference that two distinct identifiers refer to the same real world object, e.g. that ‘Dr. Gordon Freeman’ and ‘gordonFreeman’ are one and the same thing. To accomplish this, the IRIS prototype implementation and the WSML2Reasoner framework have been modified to add new transformations and reasoning behavior. The WSML-Core reasoner deliverable D3.2.5 [7] discussed, in addition to the accomplished evaluation strategies, an advanced strategy that was expected to be implemented for the second prototype of the WSML-Rule reasoner, considered in this deliverable. Section 3.3 reviews the proposal and justifies why the implementation of the strategy was not carried out.

3. Software Description

This section discusses the architecture of the logic programming part of the reasoning framework WSML2Reasoner, as well as the Datalog-based reasoner implementation IRIS and related issues that have been tackled since the first prototype implementation was released with deliverable D3.2.3 in month M18 [6]. The current implementation of the final SOA4All reasoner prototype follows in principle the guidelines specified in [4].

The Datalog reasoner IRIS is written in the Java programming language and is integrated in the reasoning framework WSML2Reasoner. The rule reasoner can be run with different underlying Datalog engines. In order to align with the existing reasoner framework, the reasoner release provides implementations for the following interfaces:

- `DatalogBasedWSMLReasoner`: An implementation of this interface takes care of axiomatization, normalization and generation of Datalog rules of WSML expressions. The Datalog rules are represented by a generic object model. These rules are then passed on to an external reasoning engine represented by a concrete implementation of a `DatalogReasonerFacade`.
- `DatalogReasonerFacade`: An implementation of this interface converts the generic Datalog object model into the representation required by the underlying Datalog reasoning engine. The prototype reasoner provides such a facade for the Datalog reasoner IRIS.

In the following, the most prominent changes to the WSML-Rule v2.0 reasoner are discussed and revised. The first two subsections are related to the W3C standards RIF and XML Schema Datatypes and the efforts that have been undertaken for aligning the implementations to them. Section 3.3 will then discuss the issues related to equality in rule conclusion, a feature that has been added to the language variants due to the alignment of WSML to the RIF BLD standard.

3.1 RIF Datatypes and Built-in Predicates

The Rule Interchange Format (RIF) is a W3C working group that develops standards for exchanging rules in the context of modern rule systems and the World Wide Web.[2] RIF enables the semantic and syntactic description of rule systems, which can be further used to exchange axiomatic knowledge between systems. RIF includes a framework for defining logic dialects, several concrete dialects, data type definitions and built-in predicates and functions.

The Datalog reasoner IRIS has been updated to support most RIF built-in datatypes, predicates and functions that have been identified as being relevant for WSML [2]; currently all but the list built-ins are implemented. The new supported built-in functions and predicates are listed in [21] and implemented in the WSML2Reasoner framework, as well as in IRIS. From a high level perspective they include:

- Predicates for all datatypes: Predicates that are not restricted to certain datatypes.
- Guard Predicates for Datatypes: Predicates to check if a term is of a specified data type.
- Negative Guard Predicates for Datatypes: Predicates to check if a term is not of a specified data type.
- Datatype Conversion and Casting: Various functions to convert from one data type to

[2] RIF working group, http://www.w3.org/2005/rules/wiki/RIF_Working_Group

another.

- Numeric Functions and Predicates: Various functions and predicates for operations on numeric datatypes; e.g., subtract, divide, less-than.
- Functions and Predicates on Boolean Values: Various functions and predicates for operations on the boolean datatype.
- Functions and Predicates on Strings: Various functions and predicates on the string datatype; e.g., concatenation, substring, starts-with.
- Functions and Predicates on Dates, Times, and Durations: Various functions to extract elements from the complex data types date, time and duration.
- Functions and Predicates on rdf:XMLLiterals: self-defined; e.g., equality.
- Functions and Predicates on rdf:PlainLiteral: self-defined; e.g., language tag extraction.
- Functions and Predicates on RIF Lists: Various functions and predicates on the RIF list datatype; e.g., contains, make-list.

Note that, as stated above, the last point *Functions and Predicates on RIF Lists* is not implemented, neither for WSML2Reasoner nor for the IRIS reasoner. To this end, there are 63 RIF functions and 46 predicates implemented and supported by the presented release.

The RIF DTB specification references to the document *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes* which is not yet a W3C recommendation. Section 3.2 discusses the working draft and the proposed datatypes separately; the XML Schema datatypes are almost fully implemented in the reasoning framework and the reasoner implementation.

3.2 W3C XML Schema Datatypes

Although deliverable D3.1.4 [3] does not mention built-in datatypes, the WSML specification [10] does by defining datatype constructors based on XML Schema datatypes and the RDF XMLLiteral. For the second version of WSML, the set of supported datatypes has been extended from 17 in [10] to 47, capturing all W3C XSD datatypes and RDF-based datatypes referenced in RIF DTB. For this reason, the IRIS reasoner and the reasoner framework were updated to be fully compliant to the current W3C working draft (version 1.1 part 2) of the XML Schema Definition Language [13].

For a full list of changes to datatype definitions consult Appendix I of [13]. The framework and the reasoner implementation are conformant with the standard, with some minimal deviations only:

1. One issue is related to the notions of *equality* and *identity* for the float and double datatypes, Appendix I.1 of [13] serves as definition for the distinction: “The (numeric) equality of values is now distinguished from the identity of the values themselves; this allows float and double to treat positive and negative zero as distinct values, but nevertheless to treat them as equal for purposes of bounds checking. This allows a better alignment with the expectations of users working with IEEE floating-point binary numbers”. The current implementation does not support distinction between equality and identity, thus this difference is not reflected in reasoning. For the OWL 2 test cases, this causes the implementation fail when it comes to checking whether a data property is functional or not (as discussed in deliverable *D3.2.7 [8]*). This specification is however of minor practical relevance, thus the reasoner has not been adapted.
2. The RIF DTB standard [21] does not include all datatypes defined by XSD [13] that are implemented in the WSML2reasoner framework. However, the XSD standard defines the following datatypes which are not implemented in the reasoner framework and that are not part of RIF DTB:

- IDREFS
- ENTITIES
- NMTOKENS

All other XSD datatypes are implemented according to the specification.

Note that the data value constructors of WSML do not reflect the lexical representation of XSD completely. Whereas XSD defines the lexical space of datatypes using an extended *Backus Naur Format grammar,*[3] WSML defines datatype constructors based on XML Schema datatypes and separates the datatype space in primitive and complex datatypes. Primitive datatypes encompass string, integer and decimal, which have a direct correspondence to the according XML Schema datatypes. However, all other datatypes are complex and are created of one or more WSML primitive datatypes. As an example, the XML Schema data value

```
"2001-10-26T21:32:52Z"^^xsd:dateTime
```

corresponds to the WSML data value constructor

```
_dateTime(2001,10,26,21,32,52,0,0,0).
```

Problematic is the definition of the WSML constructors for durations and time zone (which is a special case of duration) in [10], which has no parameter in the constructor for defining the sign of the duration; i.e., it has to be defined in one of the duration values (e.g., year, hour). Additionally, the specification does not define how negative durations should be encoded. Examples for such durations are:

1. `"2001-10-26T21:32:52-02:00"^^xsd:dateTime`
2. `"2001-10-26T21:32:52-00:30"^^xsd:dateTime`
3. `"-PT35.89S"^^xsd:dayTimeDuration`

For the case of the above values, the duration part is always negative, but in some cases the most significant value (e.g., hour for the time zone part of `xsd:dateTime` or day for `xsd:dayTimeDuration`) is zero, which means that this value cannot be used to attach a sign to the duration.

For reasons of readability, stability and conformance to the XSD standard, the constructor definition was changed such that an additional integer value to determine sign of duration was added. In specific, the sign is defined as follows: `-1` if the duration is negative, `0` if the duration is zero, and `+1` if the duration is positive. Thus, the corresponding WSML constructors for the above examples are as follows:

1. `_dateTime(2001,10,26,21,32,52,-1,2,0)`
2. `_dateTime(2001,10,26,21,32,52,-1,0,30)`
3. `_dayTimeDuration(-1,0,0,0,35.89)`

Annex B contains a complete list of all datatypes supported by the reasoning framework with the respective WSML constructors and datatype shortcut syntax.

[3] and in most cases also a regular expression using the regular expression language defined in the document http://www.w3.org/TR/xmlschema11-2/#regexs

3.3 Equality in Rule Conclusion

RIF BLD [20] introduces instance equivalence, also known as equality in rule heads. In WSML this permits to declare that different instance identifiers (IRIs) refer to the same object. In Datalog equality in rule heads allows for the declaration of equivalence between constant terms, such as strings or integers, too. Equality in rule heads has been integrated into the Datalog reasoner IRIS. Two approaches have been implemented to realize this feature, a rewriting technique and integrated support for equivalence in rule heads, see *D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0* [7].

D3.2.5 furthermore discusses a formal algorithm for extending the semi-naive evaluation strategy defined in [12]. The problem is that "equality in the rule conclusion" breaks the semi-naive evaluation strategy, since tuples that were not able to be joined in the first place are not considered any more for later iterations, which is wrong as some asserted equality might enable them to be joined.

The easiest approach is to fall back to the naive evaluation in case of any equality assertion. A more complex solution would be to extend the semi-naive evaluation by some post-processing; i.e., using the EQUAL relation as intermediate relation for joins: P x EQUAL x Q. These two approaches were described in D3.2.5, Sections 4.1 and 4.4.

After investigating the problem thoroughly, the algorithm was defined slightly different to optimize the evaluation. The EQUAL relation is used in a semi-naive manner as pre-processing step to extend the relations before each join iteration. This is basically the same as the above discussed join with the intermediate EQUAL relation, but was thought to be more efficient since it keeps the results of equality joins in the according relations; i.e., joins need to be computed only once. That optimization would be useful if a certain argument of a relation is considered for more than one join. The algorithm extension compared to [7] is as follows:

For some rule

$p(X,Y) :- r(X,Z,U_1,...,U_n)$ and $s(Z,Y,V_1,...,V_m)$

do (semi-naive) pre-processing

$r(X,Y,U_1,...,U_n) :- r(X,Z,U_1,...,U_n)$ and EQUAL(Z,Y).

$s(X,Y,V_1,...,V_m) :- s(Z,Y,V_1,...,V_m)$ and EQUAL(Z,X).

Note that the extension of the relation is only done on those arguments that are considered for joining.

The advantage is that:

- The semi-naive evaluation is used for evaluation;
- The semi-naive evaluation is used for extending the relations;
- The relations get extended only on those arguments that are considered for joining during the evaluation.

The disadvantage is that:

- A tuple gets doubled in a relation for every asserted equality of an individual that is considered for a join (unless the tuple already exists).

The implementation could be realized by using the union-find algorithm on disjoint sets representing equivalence classes [22]. More specifically, every data value is represented by an equivalence class containing all entities that are equal to it. Thus, the disadvantage of tuple doubling occurs solely on a theoretical basis and could be eradicated by the bespoken implementation.

This theoretically improved approach was presented and discussed with different leading experts in the field of reasoning, in particular logic programming: Dr. Axel Polleres, Dr. Jos de Brujn, Prof. Dr. Jürgen Angele, and Prof. Michael Kifer (notably one of the editors of the RIF standard). Two important claims that could be extracted from the Email conversions are:

- **Prof. Dr. Jürgen Angele** agreed on the theoretical value of the proposal, denied however its practical applicability. In fact, equality in rule conclusions is not and will not be supported in any of the Ontoprise products (including Ontobroker) "as it will break performance"[4].
- **Prof. Michael Kifer** pointed out that equality is hard to implement and changes the computational complexity of the evaluation. It was tagged in RIF BLD as a "feature at risk". Eventually, it was added to the recommendation as "many people felt it is needed although it is not expected that many system will implement it in full. The idea is that some consensus might emerge as to what is really needed for Semweb applications and then a subdialect of BLD will be defined appropriately. At this point there is not enough info to decide what this might be"[5].

These two strong opinions on rule-head equality clearly lowered the importance or even the use of the feature at hand. Consequently, it was decided to stick with the non-optimized implementation discussed in D3.2.5 and trade the advanced implementation for better integration and conformance with built-ins and datatypes as they were presented in Sections 3.1 and 3.2.

[4] Email conversation with Prof. Dr. Jürgen Angele, 31/05/2020

[5] Email conversation with Prof. Michael Kifer, 04/06/2010

4. Installation and Configuration

A detailed description of the WSML2Reasoner framework is given in the framework report on WSML2Reasoner [23], shared between the WP3 second prototype deliverables. This section concentrates on the description of the underlying Datalog engine IRIS and discusses how it can be used as standalone reasoner. A short guide on how to install and configure the IRIS reasoner is followed by a short example that outlines the actual use of the framework.

4.1 Installation

IRIS is an open-source Datalog reasoner that can evaluate safe or unsafe Datalog extended with function symbols, XML schema datatypes, built-in predicates and (locally) stratified or well-founded negation as failure.

It is delivered in three java "jar" files. One contains the API, another contains the parser and the last contains the actual reasoning engine implementation including two applications that provide a user interface to the IRIS engine. These applications are useful for experimenting with Datalog and various evaluation options. IRIS is licensed under the GNU lesser GPL and hosted by Sourceforge[6]. More detailed information is available on the IRIS home page[7].

Additionally, to ease the integration of the framework including all dependencies, IRIS is developed as Apache Maven project and distributed via the STI maven repository (http://maven.sti2.at/archiva/repository/external/). To get releases and snapshots of IRIS and dependent components, the following repositories have to be added to the project object model (POM) files:

```
<repositories>
  <repository>
    <id>sti2-archiva-external</id>
    <url>http://maven.sti2.at/archiva/repository/external</url>
  </repository>
  <repository>
    <id>sti2-archiva-snapshots</id>
    <url>http://maven.sti2.at/archiva/repository/snapshots</url>
  </repository>
</repositories>
```

However, the standard SOA4All project setup should have the SOA4All NEXUS repository[8] hosted by TIE in its configuration, which mirrors both STI repositories, thus they do not need to be added explicitly. The repositories that should be used in the configuration for mirroring are:

- `http://coconut.tie.nl:8080/nexus-webapp-1.3.1/content/groups/public/`
- `http://coconut.tie.nl:8080/nexus-webapp-1.3.1/content/groups/public-snapshots/`

[6] http://sourceforge.net/projects/iris-reasoner

[7] http://www.iris-reasoner.org/

[8] http://coconut.tie.nl:8080/nexus-webapp-1.3.1

The software was released on 23/07/2010 in its latest version `0.7.1`. Ongoing work, e.g. bug fixes are released on weekly basis, the corresponding version is `0.7.2-SNAPSHOT`. The reasoner can be added as dependency by adding `at.sti2.iris:iris-impl` as dependency to the POM file:

```
<dependency>
      <groupId>at.sti2.iris</groupId>
      <artifactId>iris-impl</artifactId>
      <version>0.7.2-SNAPSHOT</version>
</dependency>
```

4.2 Configuration

IRIS can be configured at the point where a knowledge base is created. All configuration parameters are collected together in a single configuration class that is passed to the knowledge base factory, thus allowing a highly flexible combination of standard and user-provided components. The configuration class contains these categories of parameters:

- **Factories** for evaluation strategies, rule compilers, rule evaluators, relations and indexes.
- **Termination parameters** for termination conditions (time out, maximum tuples, maximum complexity).
- **Numerical behavior** significant bits of floating point precision for comparison, divide by zero behavior.
- **External data sources** collection of external data source objects.
- **Optimizers** collections of program optimizers, rule optimizers and a rule reordering optimizer.
- **Stratifiers** collection of rule stratifiers.
- **Rule-safety processor** for detecting unsafe rules or making unsafe rules safe.

4.3 Datalog Reasoning

IRIS evaluates queries over a knowledge base. The knowledge base consists of facts (ground atomic formula) and rules. The combination of facts, rules and queries is known as a logic program and forms the input to a reasoning (query-answering) task.

The creation of the knowledge base is achieved in one of two ways:

- Create the java objects representing the components of the knowledge base using the API.
- Parse an entire Datalog program written in human-readable form (Datalog) using the parser.

For each query submitted to the knowledge base, IRIS will return the variable bindings; i.e., the set of all tuples that can be found or inferred from the knowledge base that satisfy the query.

4.3.1 Creating Objects with the Java API

Rules, facts, queries and their components are created using factories. The most important ones are described below[9]:

[9] all contained in the `org.deri.iris.api.factory` package

- `IProgramFactory` creates programs with or without initial values.
- `IBasicFactory` creates tuples, atoms, literals, rules and queries.
- `ITermFactory` creates variables, strings and constructed terms.
- `IConcreteFactory` creates all datatype terms.
- `IBuiltinsFactory` creates built-in atoms provided by IRIS.

The `Factory` class holds static final instances of all the factories, so they can be easily imported (e.g., import static `org.deri.iris.factory.Factory.CONCRETE`). For a more complete list of methods, input parameters and return values it is recommended to read the JavaDoc[10].

4.3.2 Creating Objects Using the Parser

Instead of creating the java objects by hand, the `org.deri.iris.compiler.Parser` can be used to parse a Datalog program. The grammar used by the parser is described in the grammar guide [11].

4.3.3 Evaluating a Program

After the components of a logic program have been created, either step by step using the API factories or using the parser, a knowledge base can be created and queries evaluated by following these steps:

1. **Choose a configuration:** a default configuration object can be obtained from the `KnowledgeBaseFactory` class. Modify this object to change the `KnowledgeBase` behavior.
2. **Instantiate a `KnowledgeBase`:** by passing the configuration object, starting facts and rules to the `KnowledgeBaseFactory.createKnowledgeBase()` method.
3. **Execute queries:** after initialization queries can be executed against the `KnowledgeBase` by calling `execute()`. Two variations of this method are available. The first one just accepts a query and the second accepts a query and an array for variable bindings. This second method can be useful if the query is complex and the order of variables is not obvious.

[10] http://www.iris-reasoner.org/snapshot/javadoc/

4.4 Example

Listing 1 outlines an example for an IRIS evaluation where the default configuration is used. The method `loadRuleBase()` emulates the loading of the rule base; e.g., from a file. Annex A shows such an example rule base with one rule and one query. The result to the query is `28` since the rule infers that `gordanFreeman` and `gf` are equal.

Listing 1: IRIS example parsing and query execution

```
// load the rule base to String
String program = loadRuleBase();
// create parser instance
Parser parser = new Parser();
// parse the datalog program
parser.parse( program );
// extract facts and rules
Map<IPredicate,IRelation> facts = parser.getFacts();
List<IRule> rules = parser.getRules();

// create knowledge base from facts and rules
IKnowledgeBase knowledgeBase = KnowledgeBaseFactory.
  createKnowledgeBase( facts, rules );

// iterate over queries in program String
for( IQuery query : parser.getQueries() )
{
  // execute the query
  IRelation results = knowledgeBase.execute( query );
  // print results to console
  System.out.println(results.toString());
}
```

5. Evaluation

The evaluation of the WSML-Rule v2.0 reasoner is done on implementation specific level. This means that the implementation of IRIS is tested rather than the entire reasoning framework, which allows to evaluate the performance of the reasoning engine more specifically. Although the evaluation is not performed on WSML knowledge bases, the evaluation respects the expressivity of the WSML LP variants. This means that knowledge bases are created such that the used features match the expressivity of specific language variants. The evaluation focus is on comparison of the actual performance with the theoretical complexity results discussed in [26], [30]. However, section 5.1 summarizes the usage of the reasoning framework in the project and references deliverables that evaluate the reasoner performance where possible.

5.1 Performance in Application Scenarios

For the evaluation of the reasoning framework in terms of use case scenarios, we refer to the deliverables of the according work packages. Table 1 gives an overview of Ontology-based reasoning in the SOA4All project.

Table 1: Reasoner usage

Component	Work package	WSML variant
Semantic Discovery	WP5	Core
Rule-based Ranking	WP5	Flight/Rule
Fuzzy Ranking	WP5	DL
Design Time Composition	WP6	Flight/Rule
Process Optimization	WP6	DL

Deliverable D5.3.2 Second Service Discovery Prototype [27] has undertaken a thorough performance evaluation of reasoner performance in Web service discovery. The performance test aims at the evaluation of the entire process of semantic service discovery, which however produces representative results for the reasoner evaluation since “semantic matchmaking highly depends on the performance of the reasoner” [27]. The evaluation was carried out by using the Semantic Web for Research Community (SWRC) ontology [28] and creating artificially 5,000 to 30,000 rich semantic service descriptions. The time to answer small, medium, large sized queries ranges from 2.8s, 4.2s, 5.0s with 5,000 service descriptions to 17s, 23s, 33s with 30,000 descriptions, respectively.

Deliverable D6.4.2 Advanced Prototype for Service Composition and Adaptation Environment [29] discusses the usage of the reasoning framework in their applications. However, the deliverable describes the integration of the framework as partly done and refers to upcoming months, which will focus on the implementation of full support for “WSMO-Lite descriptions and use common SOA4All reasoner facilities”. According to ATOS this has been accomplished at current point in time for the Design Time Composer. Process Optimization developed by UNIMAN still uses their in-house reasoner solution Fact++, but will change to the SOA4All reasoning framework for the next review (M36). The upcoming deliverable D6.5.4 Evaluation of Service Construction will serve as evaluation of the reasoner in terms of performance and applicability for WP6.

5.2 Performance Test Suite

For the performance evaluation of IRIS a test harness was set up. Each WSML variant is evaluated by a generic rule base that takes a predicate `max` as input relation for the size of the rule base. The rule base is generated dynamically by the rule engine based on this value.

The rule

```
p(?n) :- p(?x), max(?max), ?x + 1 = ?n, ?n <= ?max.
```

illustrates an easy example how the reasoner is used to dynamically create an input relation for the evaluation. The input value is increased either linearly or exponentially; the resulting query times are used for evaluation. Note that every relation size is evaluated by running ten iterations to be resistant to test outliers. The results are depicted by smoothed[11] colored graphs in the following diagrams. Every diagram also shows a function that serves as upper bound for the reasoning time, visualized by a black solid graph.

All test results were produced by running *10* iterations on a system with

- Intel ® Core™ i7-620M 2x 2.66GHz,
- Ubuntu Linux 10.04, 64bit,
- 4 Gbyte DDR2 RAM,
- Sun Java SE Development Kit (JDK) 6 Update 20 (64 bit),
- Java Runtime Options: -Xmx3g.

5.3 Performance Evaluation Results

For the evaluation of the WSML-Core variant, an input relation p is created as described with 2^1 to 2^{11} tuples. The idea of the program (see Annex C, Listing 2) is to use two unary relations and create a binary relation from the cross product. Since both input relations are of equal size, the cross product results in a relation of size square compared to the input relation, for input `i = 1 ... 11` this means $(2^i)^2 = 2^{i \times 2}$ as size for the output relation. The rule to be evaluated is:

```
q(?x,?y) :- p(?x), p(?y).
```

The example is chosen to be representative for Core performance evaluation, since only unary and binary relations are used. The intention is to test the basic join operation, fundamental to evaluating Datalog. The rule base setup gives a good estimation for the amount of tuples that can be reasoned over and the corresponding time that is needed for the rather easy computation of a natural join. This data can be used as reference in the following benchmarks when more expressive constructs are evaluated.

Figure 1 illustrates the evaluation results of IRIS with the bespoken rule base. The function `f(x)` was created manually to serve as upper bound estimation for the time consumption. The offset of 30 is used to compensate setup times, for bigger input relations this offset has only minor impact. Crucial about the estimation is the exponent, namely 2.8, which is responsible for the slope. Since it is fixed, the evaluation shows that the behavior is polynomial, thus behaving according to the theoretical complexity of general recursive Datalog (PTIME) [25].

[11] Smoothness is achieved by interpolating smoothly between successive points.

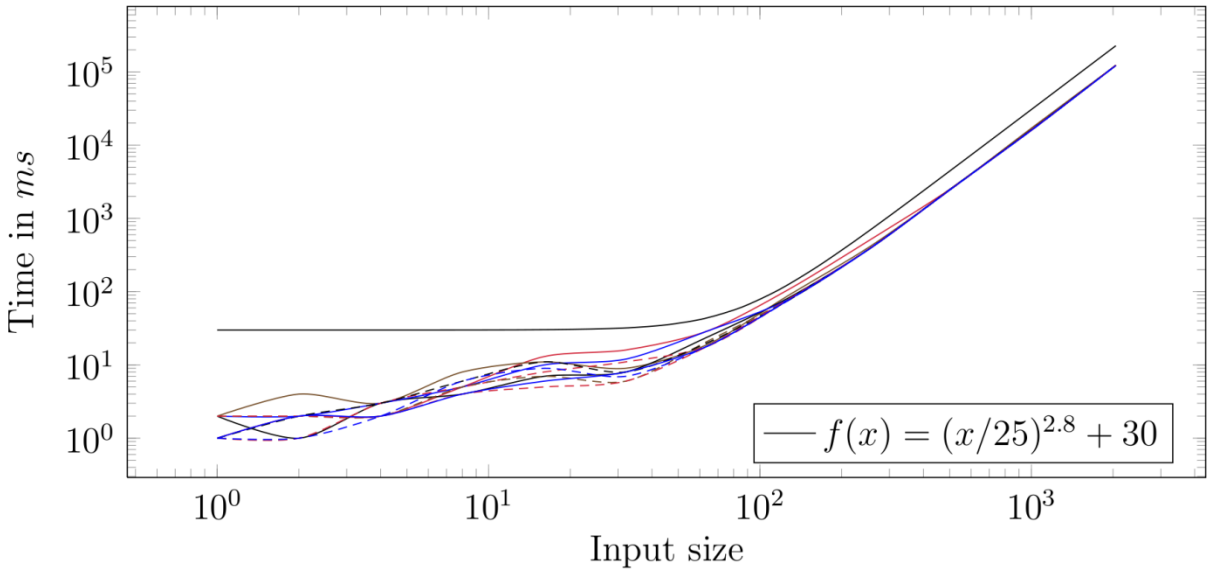

Figure 1: Evaluation of cross product

WSML-Flight extends WSML-Core with several features, one of which is stratified default negation [3]. The rule base (see Annex C, Listing 3) is generated as discussed, but for this evaluation two input relations `p` and `q` are created. `q` is created with a offset of `max/4`, e.g. for `max = 100` relations `p(1 .. 100)` and `q(26 .. 125)` are created. Each are used to create a cross product `p2` and `q2`, respectively. The query predicate `q2_minus_p2` is computed by joining `q2` with `not p2`.

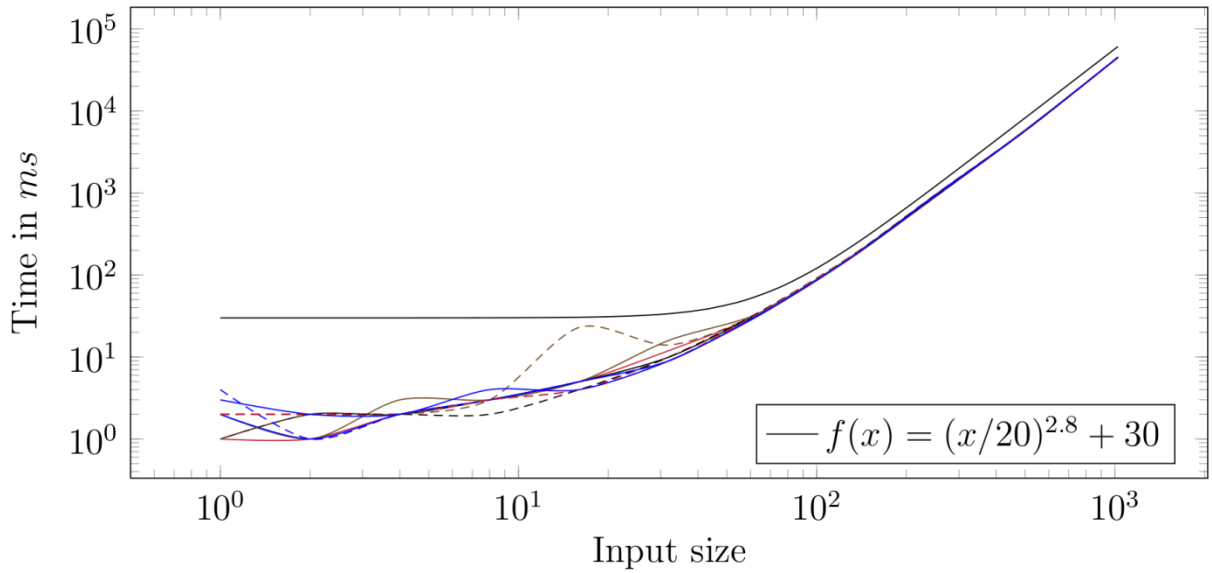

Figure 2: Evaluation of cross product with default negation

The computation is more expensive than the previous discussed since the cross product is computed twice, additionally a join between the resulting relations has to be computed. This join brings unstratified default negation into the evaluation, which means that a fact is considered to be false if it is not existent in the relation (also known as closed world assumption). This is also reflected in Figure 2 since the factor of x shrinks from

approximately 25 to 20. The exponent remains 2.8, which can be interpreted such that this feature does not influence the performance of the computation significantly.

Another feature is relations of arbitrary arity, which break the compatibility with the DL based paradigm, but allow to model knowledge more flexible. The following benchmark is borrowed from [11], where it is used to compare IRIS to other rule engines in a benchmark. The benchmark shows that IRIS outperforms the competitors in time, but due to the Java implementation IRIS has a higher memory consumption such that it cannot reason over an input relation bigger than 17. For consistency reasons, the test has been repeated in this paper to capture also the changes that have been applied to the engine in the evaluation.

The test setup is chosen differently: the input size seems to be merely small, which however is a result of the rule base definition (see Annex C, Listing 4), which creates very big relations from a small input; e.g.,

```
ra(?A,?B,?C,?D,?E) :- p(?A),p(?B),p(?C),p(?D),p(?E).
```

This means that from an input relation `p` of size `i`, the rule creates a relation `ra` of size i^5. The same procedure is applied for a relation `rb`, which both are used to create a relation `r` by applying the rule

```
r(?A,?B,?C,?D,?E) :- ra(?A,?B,?C,?D,?E),rb(?A,?B,?C,?D,?E).
```

`r` is used in the following to reconcile all artificially generated tuples in a predicate `q` with rules

```
q(?A) :- r(?A,?B,?C,?D,?E).
```

for each argument of `r`. The rule base has a relatively small input (`p`) and output (`q`), but creates big relations during computation (`r, ra, rb`).

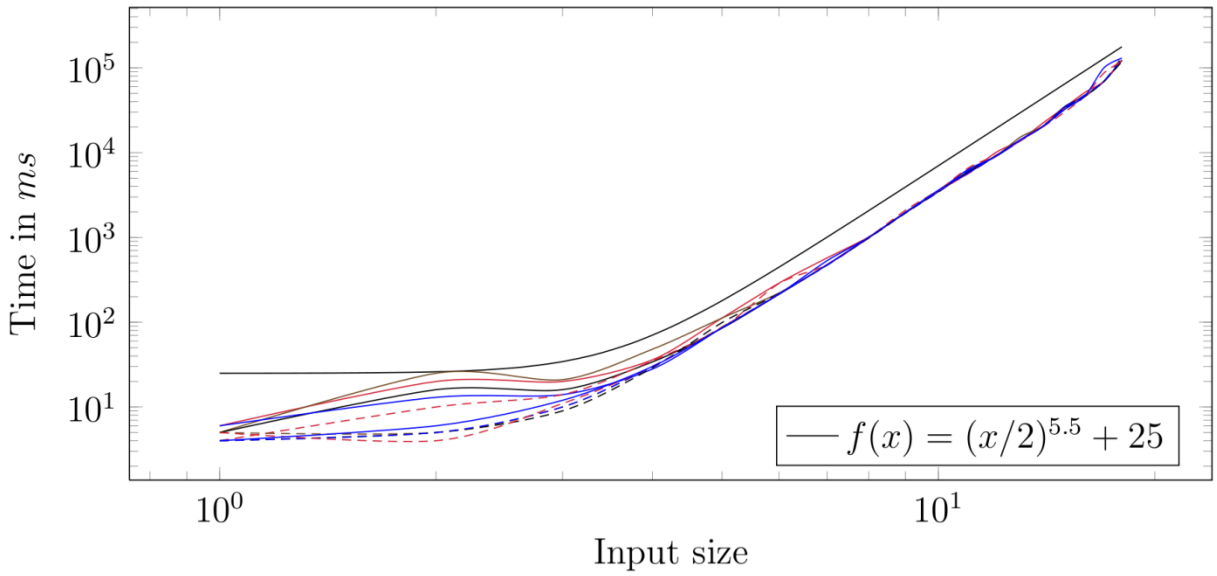

Figure 3: Evaluation of join with relations of size 5

Figure 3 illustrates the evaluation result of the discussed rule base. The maximum size of the input value is `18`, which means that two relations of size 18^5 are computed and used for joining. This fact clearly influences the slope of the graph, increasing it from approx. `3` to `5.5`, additionally the factor is decreased by the factor 10. Nevertheless, the behavior retains polynomial although the performance if much worse if only the input relation is used as reference (if the actual size of computed terms would be used, the performance would be in the same order, cf. $2*18^5$ vs 2^{22}).

For the case of WSML-Rule, the evaluation tests the reasoning strategy for well-founded semantics [26]. Well-founded semantics allow modeling of unstratified knowledge bases; i.e., the rule conclusion may depend negatively on itself. The evaluation example (see Annex C, Listing 5) is adapted from [26], in a way such that it is generated dynamically depending on a parameter serving as maximum size. The idea is to build a directed binary tree where all nodes are enumerated, starting at the root (0) and ending at the last leaf (max). Additionally the unstratified rules

```
even(?x) :- ?x - 1 = ?p, not even(?p).
jump(?x) :- even(?x), ?x - 2 = ?p, not jump(?p).
```

are used to determine whether or not a leaf is even and has the property to "jump" back. This is the case for every second even node `n1`, such that an additional edge to the node `#(n1)/2` is added to the graph (those edges create cycles thus the original tree becomes a directed graph). The program simulates a game where the player who is not able to perform a move looses, thus the player that does the last move wins. Every edge in the tree corresponds to a move, such that the rule

```
win(?x) :- move(?x,?y), not win(?y).
```

defines the predicate `win`, depending negatively on itself.

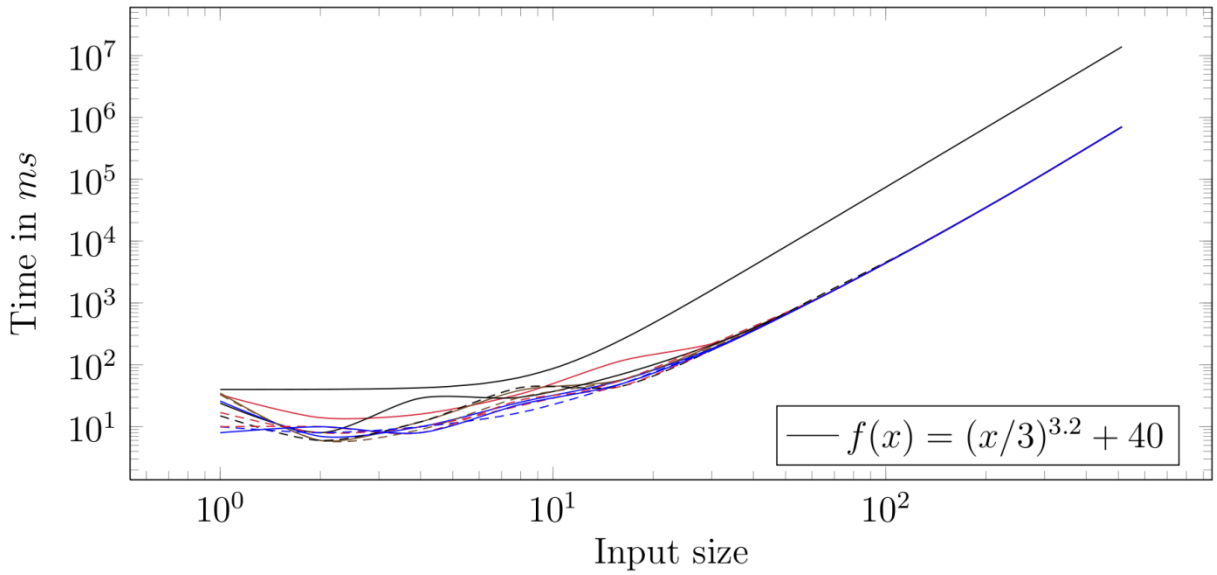

Figure 4: Evaluation of well-founded semantics evaluation strategy

As visualized in Figure 4, the performance of the well-founded semantics and the corresponding evaluation strategy is comparatively weak to the stratified bottom-up evaluation strategy evaluated so far. Furthermore, compared to the previous evaluations, the time consumption was not needed for querying but consumed as initialization time. The graph for the query time is not shown since the results for arbitrary tested size were either 0 or 1 milliseconds; i.e., the entire rule base is computed at the initialization of the rule base. Thus, if unstratified negation can be avoided it should not be used, such that (locally) stratified bottom-up evaluation strategies can be used for query answering. However, even though the performance is not as good as for less expressive variants, the evaluation shows that query answering can still be performed in polynomial time as discussed in [26].

6. Conclusions

This deliverable summarizes the efforts that have been put into the development of the WSML2Reasoner framework and the LP reasoner IRIS. The idea of WSML is to create a language for the Web Service Modeling Ontology (WSMO), and as such can be used for modeling all aspects of Web services and associated ontologies. Notably, WSML tries to avoid reinventing the wheel, such that conformance to existing Web standards is accomplished wherever applicable. For the case of data representation, the most prominent Web standard is XML, having a distinct standard for datatypes, which is almost fully implemented in both, the reasoner framework and the underlying reasoner. For the Logic Programming paradigm, one important upcoming standard is the Rule Interchange Format (RIF), which defines semantic profiles on top of a standardized syntax. The WSML variants as well as the implementations have been extended accordingly, to capture this semantics. In course of the project, RIF4J[12] has been developed, which serves as Java object model for RIF rule bases. RIF4J also supports serialization of RIF BLD rule bases as WSML logical expressions, which allows for reasoning with the WSML2Reasoner framework over any RIF BLD rule base.

This document serves as description for the second version of the reasoner. It discusses all implemented features and partially justifications for deviations from the standard. A key point is the evaluation of the reasoner. Since IRIS serves as reasoner for the WSML-Core, Flight and Rule variant, the performance evaluation is done on an implementation specific level. This allows for an easier comparison with other rule engines or future improvements of the reasoning engine by excluding the (static) syntactic transformations performed by WSML2Reasoner. The evaluation has shown that reasoning for all WSML variants is performed according to the theoretical complexity results in polynomial time. Apart from the theoretic evaluation, pointers to work package deliverables utilizing the reasoner framework are given.

[12] http://sourceforge.net/projects/rif4j/

7. References

[1] Unel, G., Keller, U., Fisher, F., Bishop, B., “D3.1.1 Defining the features of the WSML-Quark language”, SOA4All Deliverable, 2009.

[2] Unel, G., Keller, U., Fischer, F. and Bishop, B., “D3.1.2 Defining the Features of the WSML-Core v2.0 Language”, SOA4All Deliverable, 2009.

[3] Toma, I., Bishop, B., Fischer, F. D3.1.4 “D3.1.4 Defining the features of the WSML-Rule v2.0 language”, SOA4All Deliverable, 2009.

[4] Fischer, F., Bishop, B., “D3.2.1 Framework and APIs for integrated reasoning support”, SOA4All Deliverable, 2009.

[5] Pressnig, M., SOA4All deliverable “D3.2.2 First Prototype Reasoner for WSML-Core v2.0”, SOA4All Deliverable, 2009.

[6] Marte, A., “D3.2.3 First Prototype Rule Reasoner for WSML-Rule v2.0”, SOA4All Deliverable, 2009.

[7] Winkler, D. “D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0”, SOA4All Deliverable, 2010.

[8] Winkler, D. “D3.2.7 Second Prototype for Description Logic Reasoner for WSML-DL v2.0”, SOA4All Deliverable, 2010.

[9] Roman, D., Lausen, H. and Keller, U., “Web Service Modeling Ontology (WSMO)” WSMO Working Draft, 2004.

[10] The WSML Working Group, “D16.1v1.0 WSML Language Reference”, WSML Final Draft, 2008.

[11] B Bishop, F Fischer, “IRIS - Integrated Rule Inference System”, International Workshop on Advancing Reasoning on the Web, 2008.

[12] Ullman, J. D., "Principles of Database and Knowledge base Systems”, vol. I. Chapter 3 (Logic as a Data Model), 1988.

[13] Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C. M., Thompson, H. S. “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, W3C Working Draft, 3 December 2009.

[14] Bao, J., Hawke, S., Motik, B., Patel-Schneider, P. F., Polleres, A., "rdf:PlainLiteral: A Datatype for RDF Plain Literals”, W3C Recommendation, 27 October 2009.

[15] The IRIS Datalog reasoner website: http://www.iris-reasoner.org/

[16] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. 2003. Description logic programs: combining logic programs with description logic. In Proceedings of the 12th international Conference on World Wide Web (Budapest, Hungary, May 20 - 24, 2003). WWW '03. ACM, New York, NY, 48-57. DOI= http://doi.acm.org/10.1145/775152.775160

[17] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P.F. (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, 2003.

[18] Kifer, M., Lausen, G., and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. J. ACM 42, 4 (Jul. 1995).

[19] Toma, I., Bishop, B., Fischer, F, SOA4All deliverable “D3.1.4 Defining the features of the WSML-Rule v2.0 language”, 2009.

[20] Boley H., Kifer M., “RIF Basic Logic Dialect”, W3C Recommendation, 2010.

[21] Polleres A., Boley H., Kifer M., “RIF Datatypes and Built-Ins 1.0”, W3C Recommendation, 2010.

[22] Gabow H., Tarjan R., “A linear-time algorithm for a special case of disjoint set union”, Journal of Computer and System Sciences, 1985.

[23] Winkler, D., Pressnig M., “Reasoning Framework Report Installation and Configuration”, SOA4All Deliverable Attachment, 2010.

[24] Klyne G., Carroll J., McBride B., “Resource Description Framework (RDF): Concepts and Abstract Syntax”, W3C Recommendation, 2004.

[25] Calvanese D., Giacomo G., Lembo D., Lenzerini M., Rosati R., “Data Complexity of Query Answering in Description Logics”, Proceedings of the 10th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2006), pp. 260-270, Lake District, 2-5 June 2006.

[26] Van Gelder A. ; Ross K. A. ; Schlipf J. S., “The well-founded semantics for general logic programs”, Journal of the Association for Computing Machinery, 1991.

[27] Junghans M., Agarwal S., “D5.3.2 Second Service Discovery Prototype”, SOA4All deliverable, 2010.

[28] Sure Y., Bloehdorn S., Haase P., Hartmann J., and Oberle D., “The SWRC Ontology - Semantic Web for Research Communities” in Proc. of the 12th Portuguese Conference on Artificial Intelligence – Progress in Artificial Intelligence (EPIA 2005), ser. LNCS, vol. 3803. Springer, December 2005, pp. 218–231.

[29] Gorroñogoitia Y., Radzimksi M., Lecue F., Villa M., di Matteo G., “D6.4.2 Advanced Prototype for Service Composition and Adaptation Environment”, SOA4All deliverable, 2010.

[30] Immerman N., “Relational queries computable in polynomial time”, Information and control, 1986.

Annex A. Example rule base

Listing 2: Example rule base for inferring equality in the rule conclusion

```
// facts
hasName('gf', 'Gordon Freeman').
hasAge('gf', 28).
hasName('gordenFreeman', 'Gordon Freeman').

// rule with head equality
?x = ?y :- hasName(?x, ?name) and hasName(?y, ?name).

// query, age of 28 should be inferred
?- hasAge('gordenFreeman', ?age).
```

Annex B. Datatype constructors

Annex B lists all available datatype constructors for XSD and RDF datatypes. The table contains the WSML primitive datatype constructors for string, integer and decimal, corresponding to the respective XSD datatypes. All other datatypes, called complex datatypes, are created from one or more primitive datatypes. For the restriction on the data values have a look at the respective specification, XSD [13], RDF PlainLiteral [14] or RDF XMLLiteral [24].

Table 2: WSML Datatypes

Datatype	Syntax	Datatype constructor shortcut syntax
XSD Primitive Datatypes		
`string*`	`xsd#string("any-character*")`	`_string`
`boolean`	`xsd#boolean(string_boolean)`	`_boolean`
`decimal*`	`xsd#decimal("'-'?numeric+.numeric+")`	`_decimal`
`float`	`xsd#float(string_float)`	`_float`
`double`	`xsd#double(string_double)`	`_double`
`duration`	`xsd#duration(integer_sign, integer_year, integer_month, integer_day, integer_hour, integer_minute, decimal_second)` `xsd#duration(integer_sign, integer_year, integer_month, integer_day, integer_hour, integer_minute, integer_second)`	`_duration`
`dateTime`	`xsd#dateTime(integer_year, integer_month, integer_day, integer_hour, integer_minute, decimal_second, integer_timezone-sign, integer_timezone-hour, integer_timezone-minute)` `xsd#dateTime(integer_year, integer_month, integer_day, integer_hour,`	`_datetime`

	integer_minute, decimal_second)	
time	xsd#time(integer_hour, integer_minute, decimal_second, integer_timezone-sign, integer_timezone-hour, integer_timezone-minute) xsd#time(integer_hour, integer_minute, decimal_second)	_time
date	xsd#date(integer_year, integer_month, integer_day, integer_timezone-hour, integer_timezone-minute) xsd#date(integer_year, integer_month, integer_day)	_date
gYearMonth	xsd#gYearMonth(integer_year, integer_month)	_gyearmonth
gYear	xsd#gYear(integer_year)	_gyear
gMonthDay	xsd#gMonthDay(integer_month, integer_day)	_gmonthday
gDay	xsd#gDay(integer_day)	_gday
gMonth	xsd#gMonth(integer_month)	_gmonth
hexBinary	xsd#hexBinary(string_hexadecimal-encoding)	_hexbinary
base64Binary	xsd#base64Binary(string_base64)	_base64binary
anyURI	xsd#anyURI(string_anyURI)	_anyuri
QName	xsd#QName(string_namespace, string_localpart)	_qname
NOTATION	xsd#NOTATION(string_namespace, string_localpart)	_notation
XSD Other Built-in Datatypes		
normalizedString	xsd#normalizedString(string_normalizedString)	_normalizedstring
token	xsd#token(string_token)	_token

language	xsd#language(string_language)	_language
NMTOKEN	xsd#NMTOKEN(string_NMTOKEN)	_nmtoken
Name	xsd#Name(string_Name)	_name
NCName	xsd#NCName(string_NCNAME)	_ncname
ID	xsd#ID(string_ID)	_id
IDREF	xsd#IDREF(string_IDREF)	_idref
ENTITY	xsd#ENTITY(string_ENTITY)	_entity
integer*	xsd#integer("'-'?numeric+")	_integer
nonPositiveInteger	xsd#nonPositiveInteger(string_nonPositiveInteger)	_nonpositiveinteger
negativeInteger	xsd#negativeInteger(string_negativeInteger)	_negativeinteger
long	xsd#long(string_long)	_long
int	xsd#int(string_int)	_int
short	xsd#short(string_short)	_short
byte	xsd#byte(string_byte)	_byte
nonNegativeInteger	xsd#nonNegativeInteger(string_nonNegativeInteger)	_nonnegativeinteger
unsignedLong	xsd#unsignedLong(string_unsignedLong)	_unsignedlong
unsignedInt	xsd#unsignedInt(string_unsignedInt)	_unsignedint
unsignedShort	xsd#unsignedShort(string_unsignedShort)	_unsignedshort

unsignedByte	xsd#unsignedByte(string_unsignedByte)	_unsignedbyte
positiveInteger	xsd#positiveInteger(string_positiveInteger)	_positiveinteger
yearMonthDuration	xsd#yearMonthDuration(integer_sign, integer_year, integer_month)	_yearmonthduration
dayTimeDuration	xsd#dayTimeDuration(integer_sign, integer_day, integer_hour, integer_minute, decimal_second) xsd#dayTimeDuration(integer_sign, integer_day, integer_hour, integer_minute, integer_second)	_daytimeduration
dateTimeStamp	xsd#dateTimeStamp(integer_year, integer_month, integer_day, integer_hour, integer_minute, decimal_second, integer_timezone-sign, integer_timezone-hour, integer_timezone-minute)	_datetimestamp
RDF Datatypes		
rdf#XMLLiteral	rdf#XMLLiteral(string_literal, string_lang)	_xmlliteral
rdf#PlainLiteral	rdf#PlainLiteral(string_literal, string_lang)	_plainliteral

* primitive datatype

Annex C. Evaluation rule bases

Listing 3: Evaluation rule base for WSML-Core expressivity, testing join performance

```
p(1).
p(?n) :- p(?x), max(?max), ?x + 1 = ?n, ?n <= ?max.
q(?x, ?y) :- p(?x), p(?y).
?- q(?x, ?y).
```

Listing 4: Evaluation rule base for WSML-Flight expressivity, testing negation as failure join performance

```
p(1).
p(?n) :- p(?x), max(?max), ?x + 1 = ?n, ?n <= ?max.
p2(?x, ?y) :- p(?x), p(?y).

diff(?diff) :- max(?max), ?max / 4 = ?diff.
q(?n) :- diff(?diff), ADD(?diff, 1, ?n).
q(?n) :- q(?x), max(?max), diff(?diff),
         ?max + ?diff = ?maxplus, ?x + 1 = ?n, ?n <= ?maxplus.
q2(?x, ?y) :- q(?x), q(?y).

p2_minus_q2( ?x, ?y) :- q2(?x, ?y), not p2(?x, ?y).

?- p2_minus_q2( ?x, ?y).
```

Listing 5: Evaluation rule base for WSML-Flight expressivity, testing relations with arity bigger two join performance

```
p(1).
p(?n) :- p(?x), max(?max), ?x + 1 = ?n, ?n <= ?max.

ra(?A,?B,?C,?D,?E) :- p(?A),p(?B),p(?C),p(?D),p(?E).
rb(?A,?B,?C,?D,?E) :- p(?A),p(?B),p(?C),p(?D),p(?E).
r(?A,?B,?C,?D,?E)  :- ra(?A,?B,?C,?D,?E),rb(?A,?B,?C,?D,?E).

q(?A) :- r(?A,?B,?C,?D,?E).
q(?B) :- r(?A,?B,?C,?D,?E).
q(?C) :- r(?A,?B,?C,?D,?E).
q(?D) :- r(?A,?B,?C,?D,?E).
q(?E) :- r(?A,?B,?C,?D,?E).

?- q(?X).
```

Listing 6: Evaluation rule base for WSML-Rule expressivity, testing well-founded semantics performance (unstratified program)

```
move(0,1).
move(0,2).

even(0).
even(?x) :- ?x - 1 = ?p, not even(?p).

jump(0).
jump(?x) :- even(?x), ?x - 2 = ?p, not jump(?p).

move(?from, ?to) :- move(?x, ?from), ?from * 2 = ?t, ?t + 1 = ?to, ?to <
?max, max(?max).
move(?from, ?to) :- move(?x, ?from), ?from * 2 = ?t, ?t + 2 = ?to, ?to <
?max, max(?max).
move(?from, ?to) :- move(?x, ?from), jump(?from), ?from / 2 = ?td,
                    TO_INTEGER(?td, ?to).

win(?x) :- move(?x,?y), not win(?y).

?- win(?x).
```