



Project IST 026850 SUPER

Semantics Utilized for Process management within and between Enterprises

Deliverable 6.9

Process Ontology and Query Reasoner – Final Implementation

Leading Partner: LFUI

Contributing Partner: CEFRIEL

Security Classification: Public (PU)

September, 2008

Version 1.2

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Project Details

IST Project Number	026850
Acronym	SUPER
Project Title	Semantics Utilised for Process management within and between EnteRprises
Project URL	http://www.ip-super.org
EU Project Officer	Werner Janusch

Authors (Partner)	Barry Bishop (LFUI)		
Deliverable Owner (Partner)	Barry Bishop (LFUI)	E-mail	barry.bishop@sti2.at
		Phone	+43 512 507 6488

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Versioning and Contribution History

Version	Description	Comments
0.1	First version.	Barry Bishop (LFUI)
0.2	First draft for peer review.	Barry Bishop (LFUI)
1.0	Final version for M24 interim review	Barry Bishop (LFUI)
1.1	Second draft for M30 final review	Barry Bishop (LFUI)
1.2	Final version for M30	Barry Bishop (LFUI)

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Table of Contents

Executive Summary	1
1 Alignment with SUPER	2
1.1 Architecture Alignment	2
1.2 Methodology Alignment	3
1.3 Modelling Stack Alignment	4
2 Overview of Reasoning Components	5
3 IRIS	6
3.1 Evaluation Strategies	6
3.2 Stratification	6
3.3 Local Stratification	7
3.3.1 Motivation	8
3.4 External Data Sources	8
3.5 Non-query answering functions	9
3.5.1 Query containment	9
4 WSML2Reasoner Framework	10
4.1 External Data Sources	11
4.2 Recent Reasoner API Changes	12
4.2.1 Interfaces	12
4.2.2 Note Regarding Ontology Registration with Imported Ontologies	12
4.3 Query Containment	13
4.4 Outstanding Issues	14
5 WSML-Flight-A Query Extensions Module	15
5.1 Overview of query extensions	15
5.2 Implementation	15
6 Licensing	17
7 Future Work	18
7.1 Rule Interchange Format	18
8 References	19
9 Appendix – Local Stratification Algorithm in IRIS	21

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Table of Figures

Figure 1 SUPER architecture	2
Figure 2: The Module Design of the BPL	3
Figure 3: SUPER SBPM Lifecycle	3
Figure 4: SUPER Modeling Stack	4
Figure 5. WSML2Reasoner Framework Architecture	10

List of Tables

Table 1 Licensing Schema of third party libraries included in the Process Ontology Reasoner	17
---	----

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Executive Summary

The final implementation of the Process Ontology Reasoning is described.

This Process Ontology Reasoning consists of the underlying datalog reasoner (IRIS – Integrated Rule Inference System) and the WSML adapter layer (WSML2Reasoner). Both of these components are developed by STI Innsbruck and in combination fully support the WSML-Flight language variant.

Additional features have been introduced in to WSML2Reasoner that support the WSML-Flight-A query language extensions (defined in Deliverable 1.4), which (among other things) allows the use of aggregate functions in WSML queries.

WSML2Reasoner and IRIS have been modified to allow reasoning with data sets not stored within the reasoner itself, so called 'external data sources'.

Lastly, a non-query-answering function has been added for testing query containment.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

1 Alignment with SUPER

1.1 Architecture Alignment

The SUPER architecture showed in Figure 1 depicts the core elements in SUPER. In this document we are concerned with the Process Ontology Reasoner which is a part of the Business Process Library, the bottom-right block. The Business Process Library (BPL), described in Deliverable 3.1 “Business Process Library Design and First Prototype”, is used for storing semantically enriched process artefacts. Process artefacts are identified as process models described using semantic Business Process Modelling Ontology (BPMO), Business Process Execution Language (sBPEL) or Behavioural Reasoning Ontology (BRO) as developed in Deliverable 1.7/1.5. The Business Process Library has to be able to cope with these ontological descriptions when storing and retrieving process models, and in particular support efficient querying and reasoning capabilities based on the ontology formalism that is used. This is precisely the function of WSMML2Reasoner/IRIS as the Process Ontology Reasoner.

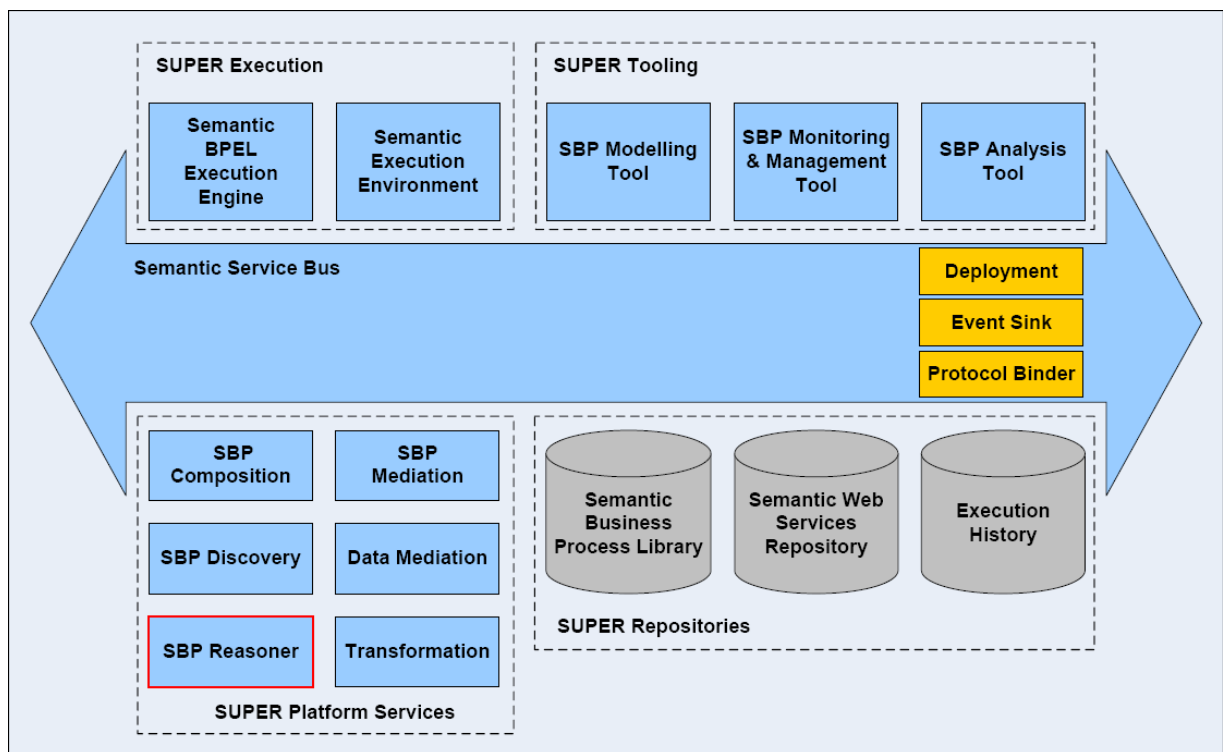


Figure 1 SUPER architecture

The Process Ontology Reasoner is a service on the semantic service bus as shown in Figure 2. It is an implementation of the Query API (a part of the upper level API) and a component responsible for the query processing in the Business Process Library.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

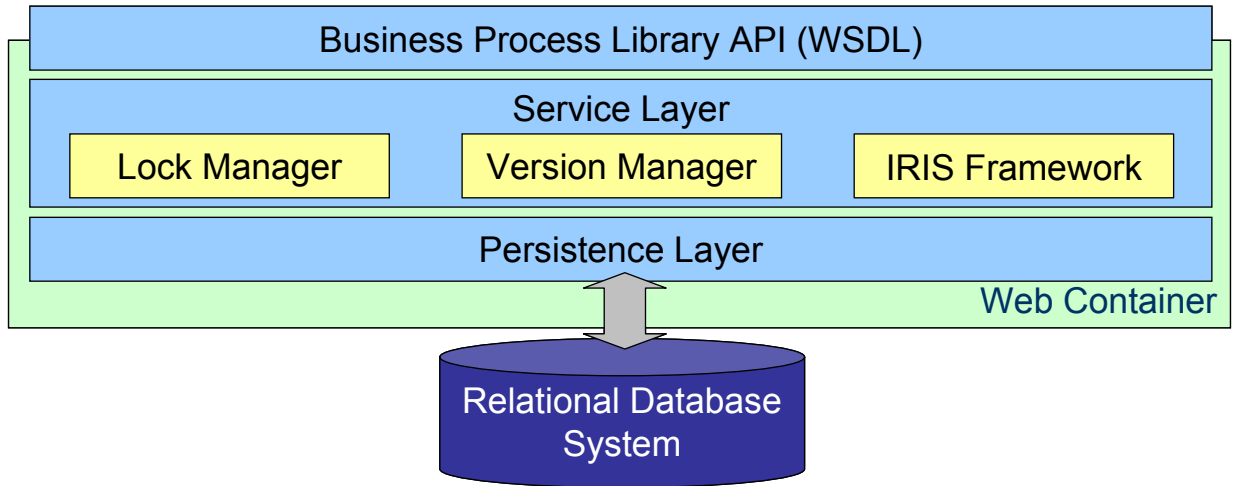


Figure 2: The Module Design of the BPL

The process ontology reasoner is used in:

- Data Mediation - During design time it allows the user to define ontology mappings on the schema level, while during runtime these mappings are executed on the ontology instance level.
- Composition – When a set of partially matching web services have been selected, select and combine particular services that can be composed to encompass the goal.
- Discovery – To match a goal’s required capabilities to web services’ provided capabilities.

1.2 Methodology Alignment

Figure 3 represents a symbolic depiction of the SUPER methodology for SBPM. As depicted in the picture, the SBPM lifecycle is based on the Ontological Foundation and aims to support the Strategic Semantic Business Process Management.

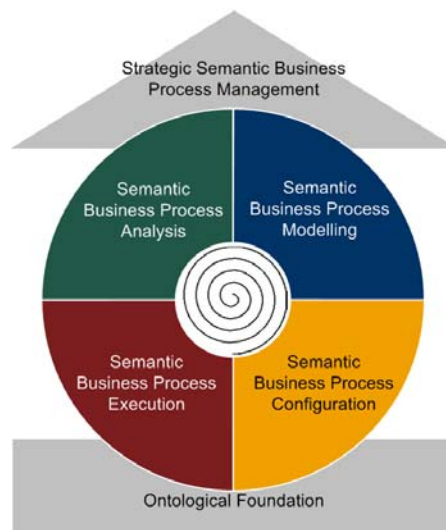


Figure 3: SUPER SBPM Lifecycle

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Reasoning with IRIS is used in SUPER in the Semantic Business Configuration phase to query the Business Process Library using ontological concepts. Also in the Semantic Business Process Analysis phase reasoning could be employed to discover information about executed processes.

1.3 Modelling Stack Alignment

As shown in Figure 4, the SUPER Modelling Stack is structured in five layers from business domain analysis to the concrete implementation. IRIS is a component which enables reasoning with WSML-Flight, the underlying language used to represent ontologies in IRIS. Thus, it is placed on the last layer, being a concrete implementation.

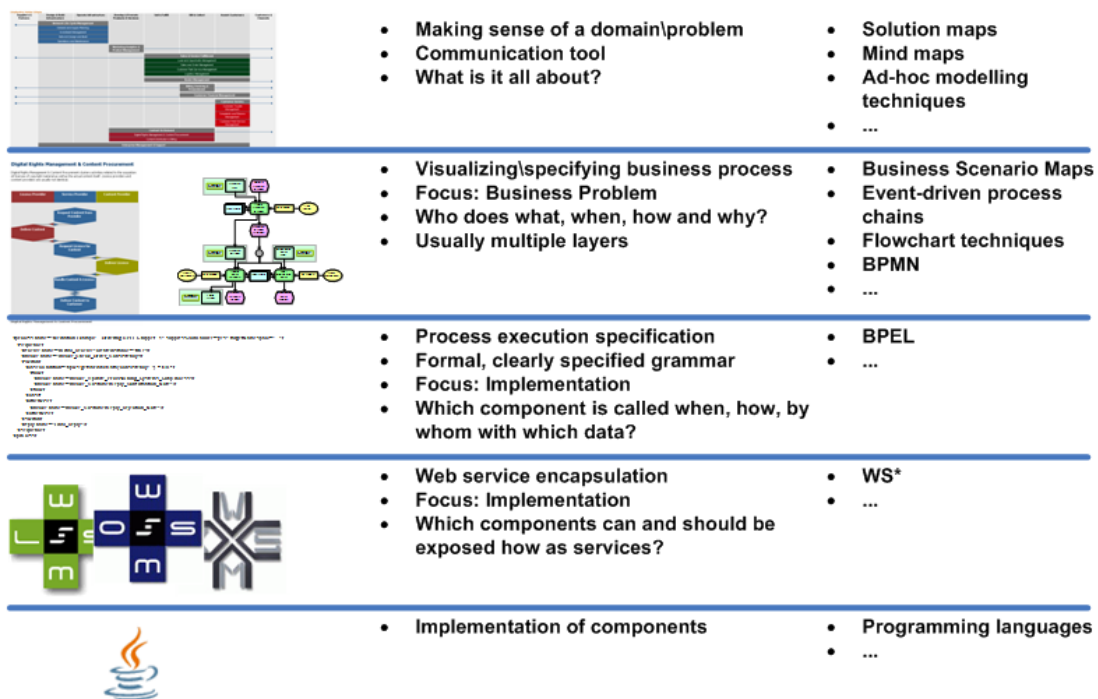


Figure 4: SUPER Modeling Stack

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

2 Overview of Reasoning Components

Semantic Business Process Management (SBPM) utilizes semantic technologies to achieve more automation throughout the BPM lifecycle. An integral part of the SBPM infrastructure is a Semantic Business Process Repository (SBPR), which is used for storage and management of business process modelling artefacts. SBPM business process models are based on process ontologies, and WSML-Flight as a variant of WSML is used for describing these ontologies.

Reasoning is achieved using a layering consisting of (from the bottom up):

- IRIS – Datalog reasoner (download from <http://www.iris-reasoner.org/download>)
- WSML2Reasoner – WSML to Datalog translator and WSML-Flight-A Query Extensions (download from <http://tools.sti-innsbruck.at/wsml2reasoner/download>)

‘WSML-Flight-A’ (developed in Deliverable 1.4) is a query language that is an extension to the WSML-Flight logical expression syntax. It permits SQL-like constructs to be used in queries including the use of aggregate functions like SUM(), MAX(), COUNT() etc.

In the next sections, the architecture of IRIS and the WSML2Reasoner framework is presented.

Lastly, licensing information and future work is covered.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

3 IRIS

IRIS is an inference engine that supports query answering for WSML-Core, WSML-Flight and WSML-Rule. In essence, it is an engine for Datalog with stratified or well-founded negation. The system implements a variety of deductive database algorithms and evaluation techniques. IRIS allows a range of data types to be used in semantic descriptions according to the XML Schema specification [6] and offers a number of built-in predicates.

IRIS has recently been modified to use hash joins, instead merge-sort joins. This technique combined with a pre-compile step for rules has seen a large increase in performance. IRIS has further been enhanced beyond the requirements of WSML-Flight and can now serve as the underlying reasoning engine for WSML-Rule by virtue of its support for unsafe rules, well-founded negation and function symbols.

3.1 Evaluation Strategies

IRIS uses only bottom-up (forward-chaining) evaluation strategies, specifically ‘naive’ and semi-naive’ as described in [13] with the option to use the magic-sets [2][3] rule re-writing technique. ‘Magic Sets’ optimisation can rival top-down, resolution based methods for efficiency.

The version of the reasoner delivered as part of Deliverable 6.3 was not able to evaluate conjunctive queries when using the magic sets evaluation strategy. This restriction has now been removed.

3.2 Stratification

We describe the following construct:

$$p(X) \text{ :- } q(X), \text{ not } r(X)$$

as meaning, that the relation associated with predicate ‘p’ contains all those values from ‘q’ that are not in ‘r’. In other words, the set difference of ‘q’ and ‘r’.

Traditional forward chaining methods for evaluating logic programs involve simply using the values of tuples from relations associated with predicates in rule bodies and applying them to the program’s rules to generate more tuples for the relations associated with the predicates in rule heads.

Without negation such techniques are guaranteed to be monotone. However, in the presence of negation, rules that generate tuples for a predicate that is used in negated sub-goals of other rules, must be ‘fully’ evaluated before evaluation of the dependant rules begins.

Consider what would happen if we have the following:

$$p(X) \text{ :- } q(X), \text{ not } r(X) \tag{1}$$

$$r(X) \text{ :- } t(X) \tag{2}$$

$$q(a)$$

$$q(b)$$

$$t(a)$$

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

If the known facts are applied to rule (1) first, the following new facts are generated:

$$p(a)$$

$$p(b)$$

Then applying the known facts to rule (2) produces the following:

$$r(a)$$

However, the existence of fact $r(a)$ should have precluded the deduction of fact $p(a)$ in rule (1).

In order to ensure that rule evaluation is monotone, rules must be evaluated in a specific order. To achieve this, rules are allocated to strata where rules of a given stratum are evaluated completely before beginning evaluation of the next higher stratum.

For any general rule:

$$p :- L_1 \dots L_m, N_1 \dots N_p$$

where $L_1 \dots L_m$ are positive literals and $N_1 \dots N_p$ are negative literals, monotone evaluation using forward chaining techniques can only be assured if the rule 'p' be allocated to a stratum that is at least as high as each of its positive literals and at least one higher than each of its negative literals.

Such a scheme would therefore require that rule (2) above is evaluated before rule (1).

When all the rules in a knowledge base can be allocated to strata with no inconsistencies, the knowledge base is said to be stratified.

This approach therefore precludes the evaluation of any logic program containing a rule that has a negative dependency upon itself.

3.3 Local Stratification

There are genuine reasoning activities that can lead to the creation of logic programs containing rules that do contain a negative dependency to themselves, but can still be evaluated in a meaningful way, because of the presence of constants in the rules that separate the domains of tuples used as input to the rule and tuples produced by the rule.

Consider:

$$p('a', X) :- q(X), \text{ not } p('b', X) \quad (3)$$

This rule can produce tuples ('a',?) for the relation associated with predicate 'p' from tuples ('b',?) also associated with the relation for 'p'. However, no special treatment is required, because nothing produced by the rule can be used as input to the rule, because of the presence of constants 'a' and 'b'.

A more complicated scenario is as follows:

$$p('a', X) :- r(X), \text{ not } q('b', X) \quad (4)$$

$$q(X, Y) :- p(X, Y) \quad (5)$$

Here rule (5) can produce tuples for input to rule (4) and vice versa.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

The algorithm used by IRIS to evaluate such logic programs is explained in section 9.

3.3.1 Motivation

The ability to evaluate locally stratified logic programs is important, because of the way that WSML is modelled for reasoning purposes, where `memberOf`, `subConceptOf`, `hasValue`, etc are treated as predicates (i.e. meta-predicates). This allows for very powerful query answering, such as:

- what is the value of this attribute for the instance?
- what instances have attributes with these values?
- what instances have an attribute named...?

Such modelling easily leads to locally stratified logic programs. For example, the following axiom:

```
axiom isSingle definedBy
    ?x[family_status hasValue single] :-
        ?x memberOf Human and naf ( ?x[married_to hasValue ?y] )
```

is modeled in datalog as:

```
_has_values(?x, family_status, single) :-
    _member_of(?x, Human), ¬ _has_values(?x, married_to, ?y)
```

using the meta-predicate `'_has_values'`. However, notice the negative self dependency between the head and body predicate `'_has_values'`.

This rule-form can and will occur frequently and this is precisely the reason that the reasoner must properly evaluate local stratified logic programs.

3.4 External Data Sources

Functionality to enable the storage and retrieval of facts in a relational database was added to the IRIS reasoner as part of Deliverable 6.3

As part of this deliverable the concept of external data sources is extended to the general case to allow reasoning over data stored in any format.

In order to use an external data source, a user must create a Java class that implements the external data source interface. During evaluation this object must answer requests from the reasoner to deliver tuples belonging to a specific predicate.

The Java interface that must be implemented is `org.deri.iris.facts.IDataSource` and is given here:

```
public interface IDataSource {
    /**
     * Retrieves some tuples for a given predicate from the data source.
     * The terms in 'from' and 'to' set the lower and upper bounds for the terms in the
     * corresponding columns of the tuples, which should be added to the
```

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

```

* tuple collection. <code>>null</code> in the <code>from</code> or
* <code>to</code> list, stands for the smallest, respectively
* biggest possible term for this column.
* @param p the predicate for which to retrieve the tuples (because one
* data source might hold tuples for multiple predicates)
* @param from the lower bound for the tuples which should be added to
* the relation (<code>>null</code> is equivalent to a tuple containing
* only <code>>null</code>s)
* @param to the upper bound for the tuples which should be added to
* the relation (<code>>null</code> is equivalent to a tuple containing
* only <code>>null</code>s)
* @param r the relation where to add the tuples
*/
public void get( IPredicate p, ITuple from, ITuple to, IRelation r );
}

```

During the initialisation of the IRIS reasoner, any number of data sources can be registered using the ‘Configuration’ object that is passed to the method `KnowledgeBaseFactory.createKnowledgeBase()`.

The details of how this behaviour is implemented in the `WSML2Reasoner` component is described in a later section.

3.5 Non-query answering functions

3.5.1 Query containment

Query containment is the process of asking: Will the results of query `q1` contain the results of query `q2` for every possible model of a knowledgebase?

In other words, is it possible to prove that the results of query `q2` are always a subset of query `q1`, independent of what facts are actually stored in a knowledgebase?

This process must therefore be carried out by examining the rule set only. Two stages in this task are envisioned:

1. Pure Datalog – no negation and no built-in predicates
2. Datalog with negation and built-in predicates

The first step is complete and involves using the Chase/Frozen fact algorithm. This algorithm is presented in [19].

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

4 WSML2Reasoner Framework

The WSML2Reasoner framework translates from WSML to Datalog using a combination of various validation, normalisation and transformation functions.

The WSML2Reasoner framework is a flexible and highly modular architecture for easy integration of external reasoning components [8]. It has been implemented in Java and uses the WSMO4J library, which provides an API for the programmatic manipulation of a WSML object model. The reasoning framework performs various syntactical transformations to convert an original ontology in WSML syntax into a semantically equivalent Datalog program. WSML reasoning tasks are then realised by executing queries over the Datalog knowledge base.

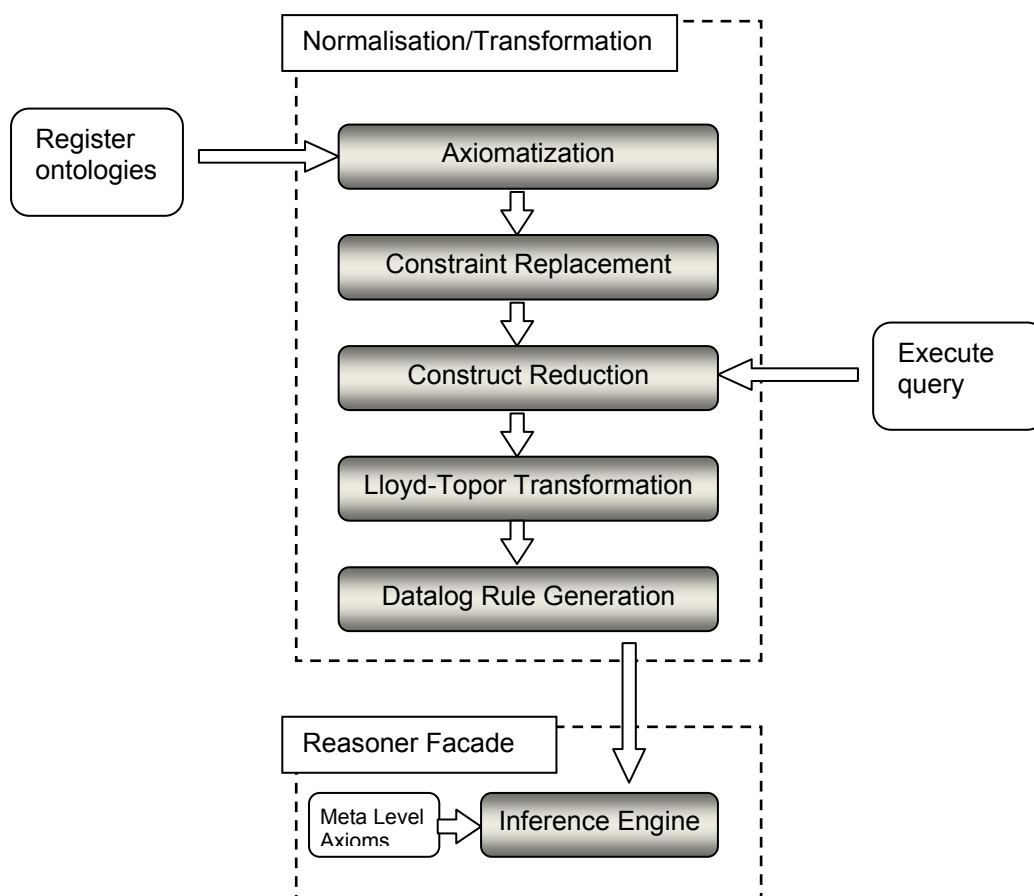


Figure 5. WSML2Reasoner Framework Architecture

Figure 5 shows the internal architecture and gives an overview of the syntactical transformations handled in the framework. The core component of the framework is an exchangeable Datalog inference engine wrapped by a reasoner facade which embeds it in the framework infrastructure. This facade mediates between the generic Datalog program produced in the transformations and the external engine's tool-specific Datalog implementation and built-in predicates.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

4.1 External Data Sources

External data sources allow reasoning with instance data that is not declared directly in an ontology. Instead, instance data can be read from an external source, such as a relational database, although in fact any storage mechanism can be used.

The user of `wsml2reasoner` must provide an adaptor class that retrieves instance data when requested by the reasoner during query answering. There is a unit test that serves as a good example to follow in the `wsml2reasoner` project here: `test/engine.iris.DataSourceTest`.

Here is a summary of what to do:

1) Create a Java class that implements `org.wsml.reasoner.api.ExternalDataSource`

2) Implement the two methods:

```
public Set<HasValue> hasValue(IRI id, IRI name, Term value);
public Set<MemberOf> memberOf(IRI id, IRI concept);
```

3) Create some configuration options for the reasoner and include the

```
new data source, e.g.
Map<String, Object> config = new HashMap<String, Object>();
config.put(IrisFacade.EXTERNAL_DATA_SOURCE,
    Collections.singletonMap("urn:dogsworld",
        Collections.singleton(new MayDataSource())));
```

4) Instantiate the reasoner:

```
config.put(WSMLReasonerFactory.PARAM_BUILT_IN_REASONER, reasoner);
config.put(WSMLReasonerFactory.PARAM_EVAL_METHOD, evalMethod);
config.put(WSMLReasonerFactory.PARAM_ALLOW_IMPORTS, allowImports);
WSMLReasoner wsmlReasoner = DefaultWSMLReasonerFactory.getFactory()
    createWSMLFlightReasoner( config );
```

From now on the adaptor object must answer requests from the reasoner by returning has-value and member-of information.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

4.2 Recent Reasoner API Changes

In previous versions of the reasoner, separate interfaces were provided for the different WSML language variants. However, it has been decided that the reasoning interface should reflect rather the reasoning paradigm. Consequently, consortium members integrating the reasoner must make a few minor changes to their software source code in order to integrate the latest version of WSML2Reasoner. These changes are detailed as follows:

4.2.1 Interfaces

WSMLReasoner	The super interface of all other reasoning interfaces. Methods common to all reasoning activities are here, e.g. for registration and de-registration of ontologies
DLReasoner	Typical description logic methods for terminological and assertional reasoning when working with WSML-DL. Replaces WSMLDLReasoner
LPReasoner	'Logic Programming' interface that is used for WSML-Flight and WSML-Rule. This interface provides the general purpose query methods that find variable bindings for a logical expression query. Replaces WSMLFlightReasoner and WSMLRuleReasoner

Software developers integrating the WSML2Reasoner will need to make the substitutions above as well as changing how they instantiate a reasoner via the WSMLReasonerFactory. This interface has also been modified to reflect the changes to the reasoners themselves. The new methods for getting a reasoner as follows:

```
public LPReasoner createCoreReasoner(Map<String, Object> params);
public DLReasoner createDLReasoner(Map<String, Object> params);
public LPReasoner createFlightReasoner(Map<String, Object> params);
```

4.2.2 Note Regarding Ontology Registration with Imported Ontologies

There has been some confusion by various consortium members regarding how the reasoner behaves when registering an ontology that imports other ontologies. Here is the behavior:

- All ontologies required for a reasoning activity must be registered with the reasoner in a single call to `registerOntologies(Set<Ontology> ontologies)`
- If ontology A imports ontology B and B contains schema or instance data required for a reasoning activity, then ontology B must be explicitly loaded (using the `wsmo4j` API) and

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

passed to the ontology registration method. WSML2Reasoner will not automatically load imported ontologies.

- Consequently, an ontology IRI is no longer needed for query methods on LPReasoner and these redundant method arguments have been dropped. A query is executed against whatever has been registered with the reasoner.

4.3 Query Containment

A new method on the WSMLReasoner interface exposes query containment:

```
/**
 * This method checks for query containment, i.e. it checks
 * for whether one query is contained within another query. The
 * query containment is checked using the 'Frozen Facts' algorithm
 * (This algorithm is presented in Ramakrishnan, R., Y. Sagiv,
 * J. D. Ullman and M. Y. Vardi (1989). Proof-Tree Transformation
 * Theorems and their Applications. 8th ACM Symposium on Principles
 * of Database Systems, pp. 172 - 181, Philadelphia) within the
 * reasoning engine IRIS.
 * The query containment check can only be performed over positive
 * queries that do not contain built-ins and disjunctions.
 * Example:
 * In the following Query1 is contained within Query2:
 * Program: vehicle(?x) :- car(?x).
 * Query1: car(?x).
 * Query2: vehicle(?x).
 *
 * @param query1 the query that may be contained within query2.
 * @param query2 the query that may contain query1.
 * @param ontologyID the original logical ontology URI
 * @return true if query1 is contained within query2, false otherwise.
 */
public boolean checkQueryContainment(LogicalExpression query1,
    LogicalExpression query2, IRI ontologyID);
```

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

4.4 Outstanding Issues

The last six months has seen an intensive period of refactoring, testing and improving of the reasoner. However, there are still remaining software issues that will be tackled during the weeks following the submission of this deliverable.

WSML relations	WSML allows relations of different arity to share the same name. These relations should be completely unrelated, however WSMO4J does not properly support this. Hence strange behavior can result if developers overload relation names.
Query extensions module	This will be fully integrated in to the LPReasoner interface. There is also a problem at the moment in that IRIs cannot be specified in the WSML logical expression of the WHERE clause. This will be addressed shortly.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

5 WSML-Flight-A Query Extensions Module

The WSML-Flight-A query extensions specification from Deliverable 1.4 provides query writers with enhanced functionality, similar to that of SQL.

This language extension is not part of the WSML specification.

5.1 Overview of query extensions

The primary motivation for the creation of the query language extensions, is the ability to use aggregate functions on WSML query result sets. Functionality identical to SQL's 'GROUP BY' with aggregate functions COUNT(), MIN(), MAX(), SUM(), AVG() is provided.

Furthermore, the ability to do pattern matching within selection criteria using 'LIKE' is also provided.

A simplified specification for a WSML-Flight-A query is:

```

SELECT [aggregate_function( ] X1 [ ) ] [ LIKE pattern1 ] , . . .
FROM <ontology>
WHERE <wsml_query>
[ORDER BY Y1 [DESC] , . . . , Yk [DESC] ]
[GROUP BY Z1 , . . . , Zm ]

```

Where <ontology> is the IRI that identifies the WSML document containing the ontology (and possibly other imported ontologies) and <wsml_query> contains the WSML query in standard WSML Flight syntax.

5.2 Implementation

For this first prototype version, it was decided to make use of existing database technology and simply do post-processing of a standard WSML-Flight query.

In essence, the current implementation uses standard WSML2Reasoner query execution functions to execute the WSML query from the 'WHERE' clause and then afterwards applies the pattern matching/aggregation functions.

To do this, the results of the WSML query are inserted in to a temporary table of an in memory relational database. Then a standard SQL query is executed on this table using the corresponding SQL SELECT statement constructs from the original query.

The steps: Parsing the WSML-Flight-A query, executing the WSML query, storing the temporary results, re-forming a standard SQL query and executing this are all straightforward.

The only problems encountered are those regarding datatypes:

- conversion from WSMO4J types to SQL types has been difficult
- defining behaviour of aggregate functions with mixed datatypes

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

The new Java source code to enable this functionality has been added to the WSML2Reasoner project in package:

`src/org.wsml.reasoner.ext.sql`

To execute WSML-Flight-A queries, first create an instance of WSMLQuery and then call its `executeQuery()` method.

A swing user interface is provided to allow experimentation with query forms and for testing generally. The user interface entry point is:

`src/org.wsml.reasoner.ext.sql.gui.DemoW`

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

6 Licensing

The license scheme adopted for the Process Ontology Reasoner is GNU LESSER GENERAL PUBLIC LICENSE (LGPL) version 2.1. The Process Ontology Reasoner consists of two software components (i.e. IRIS and WSML2Reasoner framework) which are both developed under the LGPL licence. The following table list all third-party libraries used in the implementation of the Process Ontology Reasoner as well as their corresponding licensing schemes.

Library Name	License
Apache Derby	ASL 2.0
JGraphT	LGPL 2.1
JUnit	CPL 1.0
Log4j	ASL 2.0
WSMO4J	LGPL 2.1

Table 1 Licensing Schema of third party libraries included in the Process Ontology Reasoner

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

7 Future Work

The IRIS reasoner is moving towards a more pluggable framework that will allow many kinds of extensions including: evaluation strategies, storage management, program optimisations, rule optimisations, stratification algorithms, etc.

The aggregation functions might one day be added to the datalog query syntax.

Work is also underway to implement the top-down resolution-based evaluation strategy SLDNF and there are plans for implementing OLDT soon afterwards for the top-down evaluation of well-founded models.

7.1 Rule Interchange Format

One of the major objectives related to the usage of IRIS is the implementation of support for the Rule Interchange Format (RIF)[15]. This will enable IRIS to handle rules from diverse rule systems and will make WSML rule sets interchangeable with rule sets written in other languages that are also supported by RIF.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

8 References

- [1] S. Abiteboul, R. Hull, and V. Vianu: *Foundations of Databases*. Addison Wesley, 1995.
- [2] Catriel Beeri, Raghu Ramakrishnan: *On the Power of Magic*. Journal of Logic Programming, Volume 10, Numbers 1/2/3&4, January 1991, 255-299. bzw. Kurzversion: Catriel Beeri, Raghu Ramakrishnan: On the Power of Magic. PODS 1987: 269-283.
- [3] Bancilhon, F., Maier, D., Sagiv Y. and Ullman, J. (1986) *Magic sets and other strange ways to implement logic programs*. In Proc. 5th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, Cambridge, pp. 1--15. ACM Press, New York
- [4] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, D. Fensel. *The Web Service Modeling Language WSMML*. WSMML Deliverable D16.1v0.2, 2005. Available from <http://www.wsmo.org/TR/d16/d16.1/v0.2/>.
- [5] J. de Bruijn, T. Eiter, A. Polleres, H. Tompits (2007). Embedding nonground logic programs into autoepistemic logic for knowledge-base combination, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, AAAI, Hyderabad, India. URL: <http://www.debruijn.net/publications/fo-ael-ijcai07.pdf>
- [6] P.V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation 28 October 2004.
- [7] M. van Emden, and R. Kowalski. *The semantics of predicate logic as a programming language*. Journal of the Assoc. for Comp. Mach. 23 (1976),733-742.
- [8] Stephan Grimm, [Uwe Keller](#), [Holger Lausen](#), Gabor Nagypal. *A Reasoning Framework for Rule-Based WSMML*, Proceedings of the 4th European Semantic Web Conference (ESWC), 2007
- [9] S. Heymans, C. Feier, J. de Bruijn, Stefan Zöller: *Process Ontology Query Language*. Super Deliverable D1.4, 2007
- [10] Grosz, B. N., Horrocks, I., Volz, R. and Decker, S. (2003). *Description logic programs: Combining logic programs with description logic*, *Proc. Intl. Conf. on the World Wide Web (WWW-2003)*, Budapest, Hungary.
- [11] Richard J. Lipton and Jeffrey F. Naughton. *Query size estimation by adaptive sampling (extended abstract)*. In PODS '90: Proceedings of the ninth ACM SIGACTSIGMOD-SIGART symposium on Principles of database systems, pages 40–46, New York, NY, USA, 1990. ACM Press.
- [12] Joshua Spiegel and Neoklis Polyzotis. *Graph-based synopses for relational selectivity estimation*. In SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 205–216, New York, NY, USA, 2006. ACM Press.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

- [13] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1989.
- [14] Maria-Esther Vidal Edna Ruckhaus and Eduardo Ruiz. *Query evaluation and optimization in the semantic web*. In Proceedings of the ICLP'06 Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006), Washington, USA, August 16 2006.
- [15] Rule Interchange Format, W3C Working Group <http://www.w3.org/2005/rules/wg/charter>
- [16] The IRIS Reasoner, <http://iris-reasoner.org/>
- [17] The WSML reasoner, <http://tools.deri.org/wsm12reasoner/>
- [18] The WSML reasoner online demo, <http://tools.deri.org/wsm1/rule-reasoner/v0.1/>
- [19] Ramakrishnan, R., Y. Sagiv, J. D. Ullman and M. Y. Vardi (1989). Proof-Tree Transformation Theorems and their Applications. 8th ACM Symposium on Principles of Database Systems, pp. 172 - 181, Philadelphia

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

9 Appendix – Local Stratification Algorithm in IRIS

If the Datalog program cannot be stratified, IRIS adopts a ‘brute force’ approach to identify if the program is locally stratified.

The technique involves adorning the rule head with information that indicates what can and what cannot be produced by the application of the rule. Then for each occurrence of a negated sub-goal containing constants, the adornments are used to indicate which rules can produce tuples that match for this predicate. Any rules that can produce tuples that either match or don’t match the predicate, are decomposed in to two separate rules, one that can and one that cannot produce matching tuples for the negated predicate.

The first step is to move as many constant values as possible in each rule in to the rule head so as to indicate in as restrictive a manner as possible the extent of the domain over which the rule can produce new facts.

The following rule:

$$p(Z, X) :- r(X), \text{ not } q(b, X), Z=a \quad (1)$$

would be re-written as:

$$p(a, X) :- r(X), \text{ not } q(b, X) \quad (2)$$

It is now clear that this rule can only produce tuples for predicate ‘p’ that have the constant value ‘a’ as their first term.

The second step is to adorn all the rule head predicates to show their ‘domain of influence’. For each head predicate an adornment is added for each term that indicates if the term is either:

- free, i.e. it is a variable and is represented with ‘?’
- a constant, which is represented by the constant value itself
- not a specific constant, which is indicated by ‘!’ followed by the constant value

The significance of the third adornment type will be shown later.

The example rules from section 3.3 would thus be written:

$$p^{a,?}(a, X) :- r(X), \text{ not } q(b, X) \quad (3)$$

$$q^{?,?}(X, Y) :- p(X, Y) \quad (4)$$

Step 3 now involves iterating through the body literals of every rule. Whenever a negated literal is found that contains constant terms, a search is made for each rule that can produce tuples that can fit this literal.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

Rules that can feed a particular negated sub-goal are analysed to ascertain whether they:

1. can not produce tuples to feed the negated sub-goal, i.e. no match
2. can only produce tuples that feed the negated sub-goal, i.e. an exact match
3. can produce tuples that might feed the negated sub-goal, i.e. superset

In cases 1) and 2) above, no work needs be done, because in case 1) there is no dependency and in case b) there is an exact dependency which cannot be further decomposed.

However, for those rules in 3) above, the rule can be split in to two separate rules, one which is an exact match and one that does not match for the interesting negated sub-goal.

For a given negated sub-goal $N(T_1 \dots T_n)$, the 'perfect match' rule is created by taking a copy of the rule and replacing the variables in the head with the constants from the negated sub-goal and so on in to the rule body. Then the rule adornments are updated with the constant values from each position in the rule head.

To create the non-matching rule, in-equality literals are added to the rule body, one for each variable-constant pair, where the variables are from the rule head and the constants from $N(T_1 \dots T_n)$. For each in-equality literal added, the adornment for this position in the rule head is updated by adding '!<constant_value>'.
 Following our example, the 'not q(b,X)' literals from (3) would therefore match with the head of rule (4).

However, the literal 'not q(b,X)' contains a constant, which enables the re-writing of rule (4) in to two separate rules: One that can produce tuples for 'not q(b,X)' and one that can not:

$$q^{b,?}(b, Y) :- p(b, Y) \quad (5)$$

$$q^{!b,?}(X, Y) :- p(X, Y), X \neq b \quad (6)$$

This process is repeated until no more rules can be found to 'split'. From our example, the resulting set of rules looks like this:

$$p^{a,?}(a, X) :- r(X), \text{not } q(b, X) \quad (7)$$

$$q^{b,?}(b, Y) :- p(b, Y) \quad (8)$$

$$q^{!b,?}(X, Y) :- p(X, Y), X \neq b \quad (9)$$

The standard stratification algorithm can now be applied, except that head predicate adornments as well as their names are now taken in to consideration.

From the above, it can be seen that negated literal 'not q(b,X)' from rule (7) cannot have a dependency on rule (9) with head predicate 'q!b,?(X,Y)', since no tuples are generated from rule (9) with constant value 'b' in the first term.

However, the negated literal 'not q(b,X)' from rule (7) does have a dependency on rule (8) that only produces tuples with the constant value 'b' in the first term.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

By inspection then, it can be seen that the order of evaluation of this logic program is as follows:

stratum 0: rule 8

stratum 1: rules 7 and 9

Once the stratum have been decided, the adornments can be removed and evaluation can progress as normal:

Stratum 0:

$$q(b, Y) :- p(b, Y)$$

Stratum 1:

$$p(a, X) :- r(X), \text{ not } q(b, X)$$

$$q(X, Y) :- p(X, Y), X \neq b$$

Consider the trivial program containing the single rule:

$$p(a, X) :- q(X), \text{ not } p(b, X) \quad (10)$$

The algorithm would cause this rule to be adorned as follows:

$$p^{a,?}(a, X) :- q(X), \text{ not } p(b, X) \quad (11)$$

The negated sub-goal 'not p(b,X)' has no dependency on head predicate 'p^{a,?}(a,X)', so no cyclic dependency exists.

The situation is more interesting with the addition of a second rule:

$$p(X, Y) :- p(Y, X) \quad (12)$$

Following the negated sub-goal from (10) would lead us to re-write rule (12) to give the complete set of rules as:

$$p^{a,?}(a, X) :- q(X), \text{ not } p(b, X) \quad (13)$$

$$p^{b,?}(b, Y) :- p(Y, b) \quad (14)$$

$$p^{!b,?}(X, Y) :- p(Y, X), X \neq b \quad (15)$$

The negated sub-goal 'not p(b,X)' from rule (13) requires that rule (13) exist in a higher stratum than (14), with no direct dependency on rule (15).

However, the literal 'p(Y,b)' from rule (14) matches with the head literal of every other rule, requiring that rule (14) exist in at least as high a stratum as the other two rules.

This program cannot therefore be stratified.

Project	SUPER	SUPER-Project-No	026850
	Process Ontology and Query Reasoner – Final Implementation	Work Package 6	
Document	Deliverable 6.9	Date	28.10.08

More complicated cases can arise where a rule is split more than once, either on different terms or on the same term.

$$p^{?,?,?}(X, Y, Z) :- q(X, Y, Z)$$

could be split like this:

$$p^{a,b,?}(a, b, Z) :- q(a, b, Z)$$

$$p^{!a,!b,?}(X, Y, Z) :- q(X, Y, Z), X \neq a, Y \neq b$$

and then further like this:

$$p^{a,b,?}(a, b, Z) :- q(a, b, Z) \tag{16}$$

$$p^{!a!c,!b,?}(X, Y, Z) :- q(X, Y, Z), X \neq a, X \neq c, Y \neq b \tag{17}$$

$$p^{c,!b,?}(X, Y, Z) :- q(X, Y, Z), X = c, Y \neq b \tag{18}$$

Here the adornments for rule (17) indicate that it produces tuples for predicate 'p', where the first term is neither 'a' nor 'c', the second term is not 'b' and the third term is not determined.