



# PlanetData

Network of Excellence

FP7 – 257641

---

## D32.1b MetaReasons: system prototype (version 2, final)

---

**Coordinator: Loris Bozzato, Luciano Serafini (FBK)**

**1<sup>st</sup> Quality Reviewer: Marta Sabou (MODUL)**

**2<sup>nd</sup> Quality Reviewer: Francisco J. Lopez-Pellicer (UNIZAR)**

Deliverable nature:	Prototype (P)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M47
Actual delivery date:	M47
Version:	1.0
Total number of pages:	28
Keywords:	Metadata models, Metadata reasoning prototype

***Abstract***

Different models have been proposed for the management of meta-data in the context of linked data regarding different aspects (like e.g. provenance, access control, privacy and trust). In the MetaReasons approach we propose a framework that aims at seamlessly combining and reasoning with several metalevel aspects, each one encoded as a separate “meta-theory” module in the representation of metadata.

This deliverable documents the realization for the final prototype of the MetaReasons system: this version of the prototype implements the architecture and reasoning method presented in deliverable D31.1, the additional metatheories defined in deliverable D31.2 and refines the first version described in deliverable D32.1a.

We first introduce the structure of SPRINGLES, an extension to the Sesame RDF framework we implemented for rule-based reasoning across different named graphs and that we used as the base layer for the implementation of MetaReasons. We then present how the formal definitions of MetaReasons (and the proposed list of metatheories) have been implemented using RDF graphs and SPARQL based rules.

---

## EXECUTIVE SUMMARY

This deliverable presents the results of the final implementation task (T32.2) of the *MetaReasons* activity (WP31-WP33) in the PlanetData project.

The *MetaReasons* approach has as objective the definition of a unified framework to represent and reason about provenance, access control, privacy and trust meta information of linked data datasets. Activities in this line of work consist of: the definition of the theoretical architecture of the framework and the metatheories encoding different models for the considered aspects (WP31); the implementation of such framework in a working prototype (WP32); the evaluation of the resulting prototype with respect to modelling and scalability criteria (WP33).

Objective of the present task (T32.2, last task of WP32) is to provide a definition and an implementation for a prototype of the framework on the bases of the framework architecture defined in task T31.1 and the complete list of metatheories provided in task T31.2.

In this deliverable (which revises and extends previous deliverable D31.2a) we document the results of the task and we provide a description of the implemented prototype.

After a brief summary of the framework architecture, we introduce the basic layer that we use for the implementation of *MetaReasons* over RDF. This software layer, called *SPRINGLES*, is an extension of the Sesame RDF framework that offers the possibility to define and execute inference plans composed by forward SPARQL rules that can reason across different RDF named graphs.

In the following chapter we present how *MetaReasons* has been implemented in RDF using *SPRINGLES*. We describe the implementation of the *MetaReasons* formal architecture over RDF named graphs and then we show how the rules and strategy of the calculus presented in D31.1 have been implemented as SPARQL forward rules. A demo of the prototype together with sample data is also provided.

We conclude by giving some outlook on the following evaluation task.

## DOCUMENT INFORMATION

<b>IST Project Number</b>	FP7 – 257641	<b>Acronym</b>	PlanetData
<b>Full Title</b>	PlanetData		
<b>Project URL</b>	http://www.planet-data.eu/		
<b>Document URL</b>	http://planet-data-wiki.sti2.at/web/D32.1b		
<b>EU Project Officer</b>	Leonhard Maqua		

<b>Deliverable</b>	<b>Number</b>	D32.1b	<b>Title</b>	MetaReasons: system prototype (version 2, final)
<b>Work Package</b>	<b>Number</b>	WP32	<b>Title</b>	MetaReasons: Prototype Implementation

<b>Date of Delivery</b>	<b>Contractual</b>	M47	<b>Actual</b>	M47
<b>Status</b>	version 1.0		final <input checked="" type="checkbox"/>	
<b>Nature</b>	Report (R) <input type="checkbox"/> Prototype (P) <input checked="" type="checkbox"/> Demonstrator (D) <input type="checkbox"/> Other (O) <input type="checkbox"/>			
<b>Dissemination Level</b>	Public (PU) <input checked="" type="checkbox"/> Restricted to group (RE) <input type="checkbox"/> Restricted to programme (PP) <input type="checkbox"/> Consortium (CO) <input type="checkbox"/>			

<b>Authors (Partner)</b>	Loris Bozzato, Luciano Serafini (FBK)			
<b>Responsible Author</b>	<b>Name</b>	Luciano Serafini	<b>E-mail</b>	serafini@fbk.eu
	<b>Partner</b>	Fondazione Bruno Kessler (FBK)	<b>Phone</b>	+39 (0461) 314 319

<b>Abstract (for dissemination)</b>	<p>Different models have been proposed for the management of meta-data in the context of linked data regarding different aspects (like e.g. provenance, access control, privacy and trust). In the MetaReasons approach we propose a framework that aims at seamlessly combining and reasoning with several metalevel aspects, each one encoded as a separate “meta-theory” module in the representation of metadata.</p> <p>This deliverable documents the realization for the final prototype of the MetaReasons system: this version of the prototype implements the architecture and reasoning method presented in deliverable D31.1, the additional metatheories defined in deliverable D31.2 and refines the first version described in deliverable D32.1a.</p> <p>We first introduce the structure of SPRINGLES, an extension to the Sesame RDF framework we implemented for rule-based reasoning across different named graphs and that we used as the base layer for the implementation of MetaReasons. We then present how the formal definitions of MetaReasons (and the proposed list of metatheories) have been implemented using RDF graphs and SPARQL based rules.</p>
<b>Keywords</b>	Metadata models, Metadata reasoning prototype

<b>Version Log</b>			
<b>Issue Date</b>	<b>Rev. No.</b>	<b>Author</b>	<b>Change</b>
02/08/2014	0.1	Loris Bozzato, Luciano Serafini (FBK)	First version.
17/08/2014	0.2	Loris Bozzato, Luciano Serafini (FBK)	Addressed feedback from reviewers.
28/08/2014	1.0	Loris Bozzato, Luciano Serafini (FBK)	Final version.

## CONTENTS

EXECUTIVE SUMMARY	3
DOCUMENT INFORMATION	4
1 INTRODUCTION	9
2 IMPLEMENTATION OF MR IN SPRINGLES	11
2.1 SPRINGLES	11
2.1.1 Overview	11
2.1.2 Rulesets	14
2.1.3 Rules	15
2.1.4 Closure plans	16
2.2 Implementation updates	17
3 MR SCHEMAS AND RULESET IMPLEMENTATION	19
3.1 Concrete RDF implementation of MetaReasons	19
3.2 Schemas for architecture and metatheories	19
3.3 MetaReasons ruleset and strategy	20
3.4 Examples	22
3.4.1 Combining where provenance and access control information	22
3.4.2 Combining PROV-O how provenance and ROWLBAC access control information	24
4 CONCLUSIONS	27

## LIST OF FIGURES

1.1	MetaReasons architecture. . . . .	9
2.1	MetaReasons prototype architecture . . . . .	12
2.2	SPRINGLES vocabulary: an overview . . . . .	13

## ABBREVIATIONS

**CKR** Contextualized Knowledge Repository

**MR** MetaReasons

**NG** Named Graph

**SPRINGLES** SPARQL-based Rule Inference over Named Graphs Layer Extending Sesame

**UKB** Unified Knowledge Base

# 1 INTRODUCTION

In the representation of metadata for linked data management, several approaches have been proposed to address aspects such as provenance, access control, privacy and trust. With the *MetaReasons (MR)* approach we propose a framework that aims at seamlessly combining and reasoning with several metalevel aspects, each one encoded as a separate “meta-theory” module in the representation of metadata.

The framework architecture, together with an associated reasoning procedure and initial metatheories, has been first introduced in our previous deliverable D31.1 [2]. An intuitive representation of such architecture is depicted in Figure 1.1. In a nutshell, the framework is basically defined over a two layered structure: the upper layer,

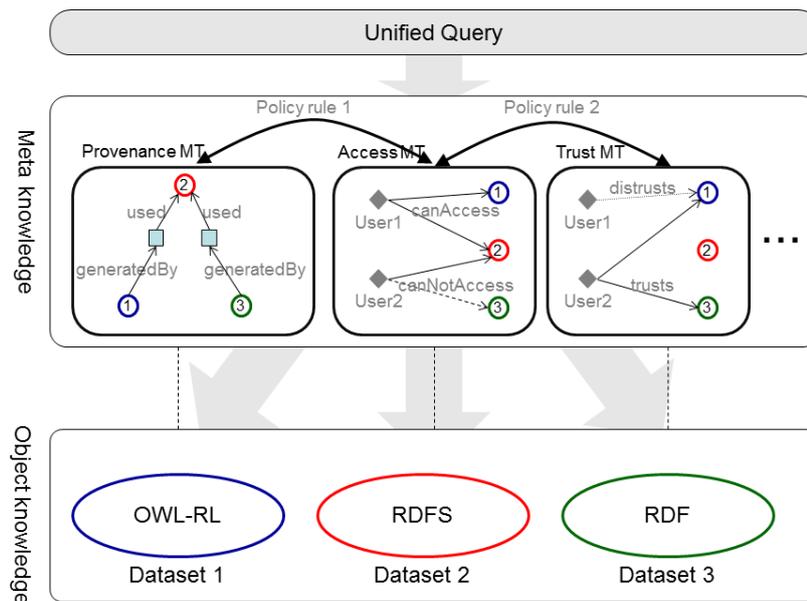


Figure 1.1: MetaReasons architecture.

called *metalevel* (or meta knowledge level), represents the metadata information of datasets while contents of the datasets themselves are represented in the lower layer, called *object level* (or object knowledge level). In the domain of the metalevel datasets are seen as atomic individuals and their meta information is linked to these dataset identifiers. The different metadata aspects (e.g. regarding the provenance and access level of each dataset) are encoded in distinct *metatheories*, metalevel theories that independently describe a particular aspect of the datasets. Each of the metatheories can use its own schema and possibly its own local reasoning formalism: the final representation of the metatheories has however to be reconduced to a single formalism, possibly encoded as one of the OWL2 fragments, in order to enable a unified reasoning over the different aspects. Similarly, formalism and reasoning at object level can be different from dataset to dataset: this modelling allows to decouple reasoning at meta and object level. *Policies*, connecting different metatheories can be defined in the part of the metaknowledge covering all of the single theories for the different aspects. On this architecture, *unified queries* over both the meta-knowledge and the object knowledge can be expressed in standard SPARQL (possibly extended with the primitives defined by the architecture). Such queries can span over the knowledge in the integrated datasets by constraining their properties in the metaknowledge: for example, if we want to retrieve “all of the facts about a certain event, from datasets which have a certain level of trust and for which we have access rights” we can express this in a SPARQL query that selects the datasets matching the meta-level requirements and then run the object-query “all of the facts about a certain event” on the selected datasets.

This architecture has been formally defined and further explained in deliverable D31.1 [2] and extended with other metalevel aspects in deliverable D31.2 [3]. In this document we describe how we have implemented the

MR framework (together with all of the proposed metatheories) over an extension of a RDF triple store that supports reasoning on named graphs called *SPRINGLES* (*SParql-based Rule Inference over Named Graphs Layer Extending Sesame*).

Following the similar approach we used in our previous works in the rule based implementation of the Contextualized Knowledge Repository (CKR) framework [1], we can summarize the implementation of MetaReasons in this sequence of steps:

1. Definition of the meta level and object level representation as RDF named graphs;
2. Definition of (OWL) schemas for symbols of architecture and single metatheories;
3. Definition and test for inference ruleset for reasoning at meta and object level.

In this deliverable we describe the results of these implementation phases. In Chapter 2 we describe the SPRINGLES platform and how MR framework can be implemented over SPRINGLES (intuitively addressing the first development phase). In Chapter 3 we describe how we implemented the architecture and specific metatheories schemas (addressing phase 2). We then describe the implementation of the evaluation plan and ruleset implementing the formal reasoning procedure defined in previous deliverable (thus addressing phase 3) and provide an example of use. A working prototype is provided: it implements the ruleset dealing with the proposed metatheories and two example repositories demonstrating the use of different metatheories in reasoning and querying. We conclude with a brief outlook of the following steps of our activity.

## 2 IMPLEMENTATION OF MR IN SPRINGLES

The abstract architecture of MR has been implemented on top of SPRINGLES, an inference engine that extends the Sesame RDF Platform [4]. The prototype accepts RDF input data expressing OWL-RL axioms and assertions for both the global and local knowledge modules: these different pieces of knowledge are represented as distinct named graphs. The prototype is based on an extension of the Sesame Java framework<sup>1</sup> and structured in a client-server architecture: the main component, called *MR core* module and residing in the server-side part, exposes the CKR primitives and a SPARQL 1.1 endpoint for query and update operations on the contextualized knowledge. The MR core module offers the ability to compute and materialize the inference closure of the input MR, add and remove knowledge and execute queries over the complete MR structure. The distribution of knowledge in different named graphs asks for a module to compute inference over multiple graphs in a RDF store, since inference mechanisms in current stores usually ignore the graph part. This component has been realized as a general software layer called *SPRINGLES (SPARQL-based Rule Inference over Named Graphs Layer Extending Sesame)*. Intuitively, the layer provides methods to demand a closure materialization on the RDF store data: rules are encoded as named graphs aware SPARQL queries and it is possible to customize both the input ruleset and the evaluation strategy. MR main implementation architecture is depicted in Figure 2.1 with three modules (*MR core*, *MR server* and *MR client*). In particular, the MR core is the central module of the system and provides the MetaReasons implementation on top of a Sesame RDF store. This core module implements the following functionalities:

**Knowledge addition/removal:** addition of a new dataset, its deletion and modification.

**Closure materialization:** implements the application of the SPRINGLES rules file according to the ruleset evaluation plans.

**SPARQL 1.1 queries:** interface to access (and modify) RDF data through SPARQL 1.1 queries.

In the next section we briefly introduce SPRINGLES. Successively we show how MR has been implemented using the SPRINGLES primitives.

### 2.1 SPRINGLES

SPRINGLES is an extension of the functionality of standard RDF stores which is required for the execution of inferences on multiple RDF named graphs. SPRINGLES' main features are:

- transparent/on-demand closure materialization based on rules;
- rules encoded as *SPARQL queries* addressing Named Graphs (NG);
- customizable *rule evaluation strategies*.

The two main features of SPRINGLES are the rules and the evaluation strategies. We describe them in details:

#### 2.1.1 Overview

The general syntax of SPRINGLES rules is the following:

```
:<rule-name> a spr:Rule ;
  spr:head "" <graph pattern>"" ;
  spr:body "" <sparql query>"" .
```

---

<sup>1</sup><http://www.openrdf.org/>

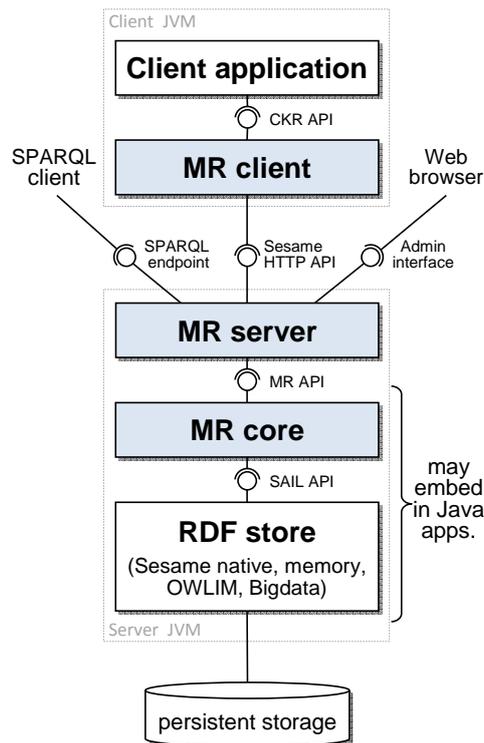


Figure 2.1: MetaReasons prototype architecture

<graph-pattern> is an RDF (named) graph that can contain a set of variables, which are bounded in the SPARQL query of the body. The body of a rule is a SPARQL query that is evaluated. The result of the evaluation of the body is a set of bindings for the variables that occurs in the rule head. For every such a binding the corresponding statement in the head of the rule is added to the repository.

An example of SPRINGLES rule is the following:

```

:pel-c-subc a spr:Rule ;
  spr:head """ GRAPH ?mx { ?x rdf:type ?z } """ ;
  spr:body """ GRAPH ?m1 { ?y rdfs:subClassOf ?z }
                GRAPH ?m2 { ?x rdf:type ?y }
                GRAPH sys:dep { ?mx sys:derivedFrom ?m1,?m2 }
                FILTER NOT EXISTS {
                  GRAPH ?m0 { ?x rdf:type ?z }
                  GRAPH sys:dep { ?mx sys:derivedFrom ?m0 }
                } """ .
  
```

where prefix `spr:` corresponds to symbols in the vocabulary of SPRINGLES objects and `sys:` prefixes utility “system” symbols used in the definition of the rules evaluation plan.

The above code corresponds to the following inference rule:

$$\begin{array}{l}
 m_1 : x \text{ rdf:type } y \\
 m_2 : y \text{ rdfs:subClassOf } z \\
 \text{sys:dep} : m_0 \text{ sys:derivedFrom } m_1 \\
 \text{sys:dep} : m_0 \text{ sys:derivedFrom } m_2 \implies m_0 : x \text{ rdf:type } z
 \end{array}$$

Intuitively, when the condition in the body part of the rule is verified in graphs ?m1 and ?m2, the head part is materialized in the inference graph ?mx. Note that in the formulation of the rule we work at the level of knowledge modules (i.e. named graphs). The reader should notice that the body of the rules contains a “filter” condition, which is a SPARQL based method to avoid the duplication of conclusions. In other words, the FILTER condition imposes a rule to be fired only if its conclusion is not already present in the data-set.

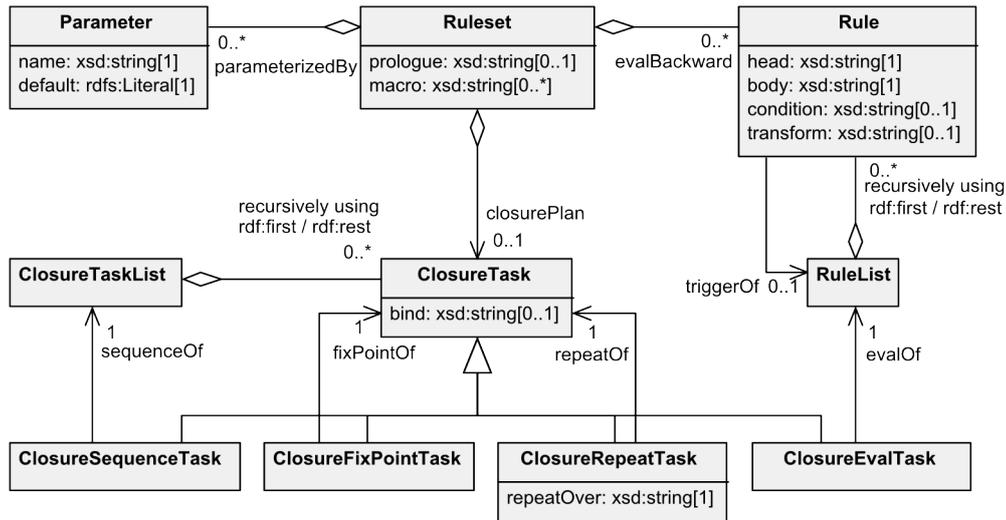


Figure 2.2: SPRINGLES vocabulary: an overview

Rules allow only to express what can potentially be deduced starting from some dataset. They need to have a strategy, or equivalently, a *plan* on how they should be applied. When we specify a theoretical deductive framework this aspect is usually not considered, and it is assumed that a rule is applied whenever it is possible, and whenever it allows us to infer something new. But when the deductive system is implemented one needs to pay attention on specifying an order on how rules should be applied.

SPRINGLES offers a number of primitives to control the applications of rules. Within SPRINGLES it is possible to declaratively specify rule application plans. In the following we describe the details of the syntax of SPRINGLES ruleset specification language.

The UML class diagram in Figure 2.2 informally presents an overview of the SPRINGLES vocabulary. Classes are rendered as UML classes, datatype properties as attributes and object properties as UML relations; minimum and maximum cardinalities and expected datatypes are also shown; note that `spr:ClosureTaskList` and `spr:RuleList` are refinements of RDF lists and as such are encoded using `rdf:first` and `rdf:rest` properties.

The components<sup>2</sup> of the vocabulary—namely *rulesets*, *rules* and *closure plans*—are detailed in the following sections. The vocabulary namespace is `http://dkm.fbk.eu/springles/ruleset#`. The suggested prefix for referencing the vocabulary is `spr:`. A list of classes and properties is reported below:

```

Classes: ClosureEvalTask | ClosureFixPointTask | ClosureRepeatTask |
ClosureSequenceTask | ClosureTask | ClosureTaskList | Parameter
| Rule | RuleList | Ruleset |

Properties: bind | body | closurePlan | condition | default | evalBackward
| evalForward | evalOf | fixPointOf | head | macro | name |
parameterizedBy | prologue | repeatOf | repeatOver | sequenceOf
| transform | triggerOf
    
```

Note that the vocabulary allows the definition of backward evaluation tasks: while the current implementation of SPRINGLES only executes forward rules, we permitted this definition (which is currently ignored) in order to easily extend SPRINGLES evaluation to combination of forward and backward rules. We remark that the kind of rule definitions offered by SPRINGLES are different from the similar mechanism of *rulesets* offered by Virtuoso (cfr. <http://docs.openlinksw.com/virtuoso/rdfsparqlrule.html>). With respect to SPRINGLES rule plans, Virtuoso rules seem limited to few RDFS and OWL constructs and they are evaluated backward at query

<sup>2</sup>The ontology defining the SPRINGLES vocabulary can be found in the distributed demo files as `spr.owl` in `03_schemas` folder.

time; moreover, they do not seem to offer a way to control the evaluation process of such rules and to support reasoning across different named graphs.

## 2.1.2 Rulesets

A *ruleset* consists of a set of rules that are evaluated either in forward- or backward- chaining and jointly specifies the type of inference provided by a SPRINGLES repository. A ruleset is an instance of class `Ruleset`; this must be explicitly stated in order for SPRINGLES to recognize a ruleset. An example of ruleset definition is shown below.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix spr: <http://dkm.fbk.eu/springles/ruleset#> .
@prefix : <http://dkm.fbk.eu/springles/rdfs-merged#> .

:ruleset a spr:Ruleset ;
  spr:parameterizedBy
    [ spr:name "enable_tbox_rules" ; spr:default "true"^^xsd:boolean ] ;
  spr:prologue ""
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
    BASE "" ;
  spr:macro "HOME = <>" ;
  spr:macro "DIFF(a,b,c) = FILTER (#a != #b && #a != #c && #b != #c)" ;
  spr:evalForward :pel-c-subc ;
  spr:closurePlan :plan .
```

Rules are evaluated each time a repository is modified in order to materialize a repository's logical closure. The closure computation process is specified through a closure plan (property `closurePlan`), which decomposes the overall process in tasks and sub-tasks and allows to fine-tune and optimize the process. A ruleset is optionally associated to a number of ruleset parameters, a SPARQL prologue and a set of macros:

**Parameters** (property `spr:parameterizedBy`) permit to inject user-supplied constant values to customize the evaluation of rules, such as parameter `enable_tbox_rules` in the example used to enable/disable RDFS rules for TBox reasoning. Each parameter is an instance of class `spr:Parameter` and is characterized by a name (property `spr:name`) and a default value (property `spr:default`), both mandatory. When a ruleset is loaded in SPRINGLES, the values of parameters provided by the user (or their default value, when not specified) are used together with the binding inheritance mechanism (see below) to fix the values of certain SPARQL variables in rules and other SPARQL expressions, thus allowing to customize their behaviour.

**SPARQL prologue** (property `spr:prologue`) is a SPARQL expression that defines the namespaces and/or the base URIs for parsing SPARQL expressions in the ruleset. The syntax is the one of SPARQL: prefixes are declared as `PREFIX pref: <uri>`, (as with prefixes `rdf:`, `rdfs:` and `xsd:` in the example), while the base uri is declared as `BASE <uri>`.

**Macros** (property `spr:macro`) are constants or functions (such as `HOME` and `DIFF(a,b,c)` in the example) which can be used inside SPARQL expressions in the ruleset to factorize SPARQL snippets and make ruleset definitions less verbose. The syntax for macro definitions is `macro(arg1, ..., argN) = template`. Name and arguments are identifiers, i.e., letter followed by letters, numbers or underscore. Arguments are optional; if absent, parenthesis can be omitted. If a macro accepts a parameter `P`, its template may reference `P` using syntax `#P`; at the same time, the macro cannot use a SPARQL variable `?P` with the same name (to avoid hard to spot errors caused by wrong typing). When a ruleset is loaded, macros are expanded by replacing their references in SPARQL expressions with the associated template, whose parameter placeholders are substituted with the values specified in the macro invocation.

**Additional ruleset properties** such as a `rdfs:label` can be specified: while ignored by SPRINGLES, they can be used by tools to display or manipulate metadata about rulesets.

### 2.1.3 Rules

An inference rule specifies using a SPARQL-based syntax how inferred statements can be deduced through the matching of statements already contained in the repository. An example of rule is reported below.

```
:rdfs11 a spr:Rule ;
  spr:triggerOf ( :rdfs2 :rdfs3 :rdfs7 :rdfs9 :rdfs11 ) ;
  spr:condition "?enable_tbox_rules" ;
  spr:head "" GRAPH ?inference_graph { ?x rdfs:subClassOf ?z } "" ;
  spr:body "" GRAPH ?graph { ?x rdfs:subClassOf ?y .
                               ?y rdfs:subClassOf ?z . }
              #DIFF(?x, ?y, ?z)
              FILTER NOT EXISTS { GRAPH ?graph { ?x rdfs:subClassOf ?z } } "" .
```

A rule has a mandatory body, that has to be matched by statements in the repository, and a mandatory head, that specifies which statements have to be inferred if the body is matched:

- The **body** (property `spr:body`) is a SPARQL expression of the kind used in WHERE clauses (GroupGraphPattern production in SPARQL 1.1 grammar). The body is evaluated against statements stored in the repository, setting all variables in the expression for which a ruleset parameter or an inherited binding (see below) is available with the corresponding value. In general, the body has the tasks of (1) implementing the premises of the rule; and (2) checking that the statements produced by the head are not already stated explicitly in the repository. Note that it is not necessary (and it may be detrimental to performances) to remove duplicates at this level, as this is already done efficiently by SPRINGLES engine. Note, also, that if the body is empty, the head will be always inferred and the rule will behave as an axiom.
- The **head** (property `spr:head`) is a SPARQL expression (a GroupGraphPattern too) restricted to use only triple patterns (as in a SPARQL CONSTRUCT clause), plus the GRAPH keyword that permits to generate statements inside specific graphs (differently from a CONSTRUCT query). The head expression may reference variables, whose values derive either from the matched rule body or from ruleset parameters and/or inherited bindings (as `?inference_graph` in the example, see binding inheritance later).

A rule is optionally associated to a condition, a transformer and a list of triggered rules:

- The **condition** (property `spr:condition`), if specified, is a generic SPARQL boolean expression (Constraint production in SPARQL 1.1 grammar) that must be satisfied by ruleset parameters and inherited bindings in order for the rule to be possibly evaluated. In the example above, if ruleset parameter `enable_tbox_rules` is false, then rule `:rdfs11` will never be considered for evaluation. Note that the expression cannot query for statements stored in the repository.
- The **transformer** (property `spr:transform`), if specified, is executed after the evaluation of the rule body and before the materialization of the rule head. It permits to invoke external Java code to implement specialized processing not possible or difficult/inefficient to realize in SPARQL. More precisely, the stream of solutions (tuples of variable bindings) extracted from the evaluation of the rule body is fed into the transformer, which produces another stream of solutions which is the one actually used to materialize the head. The transformer is identified by a SPARQL function call (FunctionCall production in SPARQL 1.1 grammar). Its URI identifies a Java static method implementing the transformer and is either in the form `<springles:builtin_transformer>` or `<java:class_name.method_name>`. Its arguments are used to configure the transformer and are SPARQL expressions evaluated based on ruleset parameters and inherited bindings (access to statements in the repository is disallowed).
- **Triggered rules** (property `spr:triggerOf`) are the rules that may fire as a consequence of the subject rule being fired, i.e., they are the successor of the rule in a rule dependency graph. This is an optional property that permit users to supply the inference engine with the result of a static analysis of rule dependencies,

which is then exploited to avoid unnecessary rule evaluations. If this property is not supplied, the engine will conservatively assume that every rule in the plan may fire as a result of the subject rule being fired. Note that the value of the property is a RuleList implemented as a specialization of RDF lists: a list is used in order to differentiate between the case the property is not supplied (e.g., because unknown) and the case there are no triggered rules (empty list); the order of rules in the list is irrelevant.

A rule's head, body and condition expression may use the macros defined in the ruleset. This is shown in the rule body of the example, where macro DIFF is called. When loading the ruleset, it will be expanded with the string FILTER (#x != #y && #x != #z && #y != #z), with macro parameters a, b and c replaced with supplied expressions ?x, ?y and ?z.

### 2.1.4 Closure plans

A closure plan instructs the system on how to compute and materialize a logical closure. A plan is decomposed into a number of closure tasks, arranged in a hierarchy. Execution of a plan starts from the root task identified by property closurePlan in the ruleset. Children tasks can then be executed as part of the execution of the parent task, that controls their execution according to a specific strategy (e.g., sequential vs fix-point execution); the process is repeated recursively for child tasks. It is allowed for a task to be included as a child in multiple positions of the hierarchy. Note that SPRINGLES can only process task hierarchies whose topology is a DAG with a single root.

An example of closure plan (implementing per-graph RDFS inference) is shown below. In the example, :plan is the root task referenced by the ruleset. It is a spr:ClosureRepeatTask that executes a child task :graph-closure for each graph in the repository. In turn, task :graph-closure is executed as the sequence of two tasks: :add-axioms and :do-closure. Note that the RDF definition of a task may be restricted by stating the minimum possible amount of statements (as in the case of the example for task :do-closure), relying on the system to infer missing information when loading a ruleset based on the semantic constraints in the SPRINGLES Ruleset Vocabulary (e.g., to infer that the task type is spr:ClosureFixPointTask); this principle applies in general to the whole ruleset RDF definition.

```
:plan a spr:ClosureRepeatTask ;
    spr:repeatOver "" SELECT ?graph WHERE { GRAPH ?graph { ?s ?p ?o } } " ;
    spr:repeatOf :graph-closure .

:graph-closure a spr:ClosureSequenceTask ;
    spr:bind "?inference_graph = IRI(concat(str(?graph), '-inf'))" ;
    spr:sequenceOf ( :add-axioms :do-closure ) .

:add-axioms a spr:ClosureEvalTask ;
    spr:evalOf ( :rdf_axioms :rdfs_axioms ) .

:do-closure spr:fixPointOf [ spr:evalOf (
    :rdf1 :rdfs2 :rdfs3 :rdfs4a :rdfs4b :rdfs5 :rdfs6
    :rdfs7 :rdfs8 :rdfs9 :rdfs10 :rdfs11 :rdfs12 :rdfs13 ) ] .
```

### Binding inheritance mechanism

When a closure task is executed, it inherits a set of variable bindings (i.e., variable = value pairs) that are used to fix the values of matching variables in referenced rules and SPARQL expressions. Variable bindings are propagated in a top-down mode. This starts with the root task, which inherits a binding for each ruleset parameter, where the variable is the parameter name and the value is the one supplied by the user at configuration time (or the default value of the ruleset parameter). Each task can then define its own bindings, which are merged with (and possibly override) the ones inherited and are then propagated down the hierarchy. The definition of binding is either implicit or explicit:

- **Implicit binding** definition is associated to certain kinds of tasks, such as the `spr:ClosureRepeatTask` task, which defines the bindings of loop variables in each iteration. This is shown in the example, where task `:plan` retrieves all the graphs in the repository with a query and then executes the child task `:graph-closure` binding variable `?graph` to a different (retrieved) graph URI each time.
- **Explicit binding** definition is controlled by the user through the use of the `spr:bind` property, which is supported for every type of closure task. The property takes as value a string expression of the form “variable = SPARQL expression”, where the SPARQL expression is evaluated based on all the bindings received as input by the task. This occurs in the example for task `:graph-closure`, which generates the URI of the graph where to store inferences and binds it to variable `?inference_graph`; this binding, together with the one for variable `?graph`, is then inherited by child tasks `:add-axioms` and `:do-closure`, and from them inherited by referenced rules `:rdf_axioms`, `:rdfs_axioms`, `:rdf1 ... :rdfs13`.

## Types of closure tasks

The following types of closure tasks are defined, each one with its own semantics:

- **Eval** (class `spr:ClosureEvalTask`). It consists in the parallel evaluation of zero or more rules, specified using property `spr:evalOf`. In case only a limited number `N` of rules can be evaluated in parallel (this is a configuration option of SPRINGLES which should be fixed based on the number of available CPU cores), rules are queued according to the order specified by `spr:evalOf` and the first `N` rules are evaluated, starting the evaluation of another rule waiting in the queue each time the evaluation of a preceding rule terminates. Inferred statements are buffered and written back to the repository only after all the rules have been evaluated, thus implying that each evaluated rule will not ‘see’ the inferences produced by other evaluated rules.
- **Sequence** (class `spr:ClosureSequenceTask`). It consists in the sequential execution of zero or more child tasks, specified using property `spr:sequenceOf`. Note that execution is strictly sequential, in the sense that the execution of a child task is started only after the execution of the previous task completes.
- **Fix point** (class `spr:ClosureFixPointTask`). It consists in the fix-point execution of a child task, i.e., the child task is repeatedly executed until it produces no more inferred statements. The child task is specified using property `spr:fixPointOf`.
- **Repeat** (class `spr:ClosureRepeatTask`). It realizes a sort of ‘for-each’ primitive that consists in the repeated execution of a child task, specified by property `spr:repeatOf`, for each tuple obtained from the evaluation of a SELECT query, specified by property `spr:repeatOver`. The query is executed once when the task is executed; query results are collected and buffered and the child task is executed for each result tuple in the order they are returned by the query. Note that the SPARQL bindings in the result tuple are propagated to the child task as additional variable bindings, providing a way to influence the execution of the child task.

## 2.2 Implementation updates

With respect to the implementation we presented in the preliminary version of this deliverable (D32.1a), in the current development of SPRINGLES we are working to update the software layer in the direction of increasing the scalability of the tool and the independence from the underlying triple store.

In particular, we extended the implementation of SPRINGLES in order to:

- Move from version 2.6 of Sesame framework to the current<sup>3</sup> version 2.7. (in the direction of providing independence from the specific version over which SPRINGLES was first created).
- Offer the support for creating SPRINGLES repositories based on OWLIM<sup>4</sup> in order to increase (and evaluate) the scalability of our approach with respect to the built-in Sesame (in-memory) repositories.

<sup>3</sup><http://www.openrdf.org/news.jsp>

<sup>4</sup><http://www.ontotext.com/owlim>

Both of these updates require to revise the implementation of the interaction between SPRINGLES and the underlying RDF store layer. In particular:

- With respect to the update to the current version of Sesame 2.7, we need to adapt the SPRINGLES reasoning interface to interact with the new methods to the Sesame 2.7 API. In particular, we are working on Sesame release 2.7.0.: however, the adaption to the 2.7 API should assure us the compatibility to all the following 2.7.x releases. In the direction of providing more independence from the Sesame framework in the client part, we are also working in the direction of providing a simple administrative web-interface that can substitute the current standard Sesame web application.
- With respect to the extension to OWLIM, it has been necessary to customize the behavior of SPRINGLES with respect to OWLIM repositories in order to be consistent with the management of blank nodes and transaction mechanisms of OWLIM. The prototype has been developed over the *OWLIM-Lite* version of OWLIM: the application interface should be however compatible to the *OWLIM-SE* and *OWLIM-Enterprise* versions.

The demo we distribute with this deliverable includes the support for OWLIM (and the example repositories are now implemented as OWLIM-based repositories) but it is still based on the web interface of Sesame 2.6. On the other hand, we remark that since the updates to the Sesame 2.7 version of SPRINGLES only concern the interface with the RDF store and do not modify the web interface or behavior of the prototype, the distributed demo is representative of the current status of the SPRINGLES implementation.

### 3 MR SCHEMAS AND RULESET IMPLEMENTATION

In this chapter we will present the implementation of the MetaReasons architecture as introduced in deliverables D31.1 and D31.2 [2, 3] over the software layer presented in previous chapter. We first introduce how the different parts of the architecture have been implemented over RDF graphs and SPARQL rules. We then describe how the schemas for the metatheories defined in D31.2 [3] have been implemented and how the reasoning strategy defined by the datalog translation defined in [2] has been implemented as a SPARQL rule plan. We conclude by showing how the university example presented in [2] (and its extension to PROV-O and ROWLBAC metatheories) is implemented in the prototype.

A demo for the implementation<sup>1</sup> for the ruleset, schemas and examples is available at:

<https://dkm-static.fbkc.eu/resources/tools/ckr/MetaReasons-d32.1b-demo.zip>

#### 3.1 Concrete RDF implementation of MetaReasons

The architecture presented for MetaReasons framework, similarly to what has been done for the CKR framework in [1], can be represented using RDF and named graphs.

In practice, RDF named graphs are used in the implementation to differentiate each of the datasets and the knowledge base for the metalevel. In the implementation, the metalevel knowledge base (comprising the information from all the metatheories) is stored in the graph `meta:metakb`, while the graphs of different datasets are named in the metalevel graph itself (e.g. `:d1`, `:d2` etc.). As discussed in the presentation of the calculus, in our current version of the implementation we assume that all of the graphs contain axioms and assertions that are expressed in OWL RL (in particular in the normal form for *SRQL*-RL presented in [2]). Metatheories are basically contained in the contents of `meta:metakb` knowledge base: their schema has been modelled as OWL ontologies that are referred (and imported) in the meta knowledge base in order to use their vocabulary. Such ontologies will be described in the following sections.

This distribution of knowledge across different graphs asks for the ability to compute and materialize inference referring to multiple graphs: this is the motivation for the use of SPRINGLES in implementing cross-graph inferences. In particular, as we will detail in the description of the ruleset, for each knowledge base (`meta:metakb` and each of the datasets), a new graph containing its inferences is computed. This is necessary in order to keep a clear separation for the inferred and asserted knowledge associated to a knowledge base.

#### 3.2 Schemas for architecture and metatheories

As introduced above, symbols used in the definition of the MetaReasons architecture and needed for the different metatheories have been defined in specific OWL ontologies<sup>2</sup>.

In the case of the MetaReasons architecture, the vocabulary (corresponding to *metareasons-meta.n3*) only defines the class of the declared datasets identifiers (that basically implements the set  $\mathbf{N}$  of dataset names from the formal definition of MetaReasons). This is obtained by declaring the OWL class `meta:Dataset`.

For each of the presented metatheory from deliverable D31.2 [3], we provide a schema defining their symbols and language definition.

In the case of the derivability metatheory  $MT_d$ , its vocabulary (provided in *mt-d.n3*) only defines the property `mtd:derivedFrom` to be a transitive property over datasets: the relation with the properties from other metatheories and the effect on the object layer will be defined at the level of inference rules.

For the access control metatheory  $MT_{ac}$ , the vocabulary (provided in *mt-ac.n3*) defines the class of access control tokens `mt-ac:AccessToken` with default token `mt-ac:dat`. The link between initial sources and their access control tokens is provided by the functional property `mt-ac:hasLabel`: on the other hand, composition of different access control information is defined by the `mt-ac:isComposedFrom` property over datasets.

<sup>1</sup>The distributed instance of the platform can be executed by running the shell script *run.bat* (or *run.sh*) in the folder *01\_springles* and accessing the local URL <http://localhost:50000/admin/>.

<sup>2</sup>In the distributed demo files, the discussed vocabularies can be found in *03\_schemas* folder.

The ROWLBAC [5] metatheory  $MT_{rbac}$  for access control is represented in the vocabulary *mt-rbac.n3*. The notion of action is defined by the class `mt-rbac:Action` with permitted and prohibited actions defined by the subclasses `mt-rbac:PermittedAction` and `mt-rbac:ProhibitedAction`. Subject and object of actions (`mt-rbac:Subject` and `mt-rbac:Object`) are linked to their actions through functional properties `mt-rbac:subject` and `mt-rbac:object`. Finally, access control roles are modelled as elements of `mt-rbac:Role` class, with the active role modelled as its subclass `mt-rbac:ActiveRole`.

In the case of the “where” provenance metatheory  $MT_{wp}$ , its vocabulary (in *mt-wp.n3*) defines the class of provenance “colors” with `mtwp:Color` and the (functional) link between the initial datasets and their color with `mtwp:hasColor`. The property defining the link between with the defining colors for the composed datasets is implemented as the object property `mtwp:hasDefiningColor`.

The “how” provenance (PROV-O) metatheory  $MT_{hp}$  is obtained as a restriction of the original PROV-O OWL ontology<sup>3</sup> to OWL RL by leaving out the non-RL axioms described in Appendix A of [6]. This version of the vocabulary is represented in *prov-o-rl.n3*.

The trust metatheory  $MT_{tr}$  is represented in the vocabulary *mt-tr.n3*. The concepts for trust representation are implemented as OWL elements following the structure described in [8] (as presented in [3]). In particular, the central notion of principal (subject that is trusted or trusts another entity) is modelled via the class `mt-tr:Principal` with subclasses `mt-tr:Trustor` and `mt-tr:Trustee`. The trust relation across the two entities is reified as an element of the class `mt-tr:Trusts` with functional object properties `mt-tr:hasTrustor` and `mt-tr:hasTrustee`. The trust relation is then characterized by different possible trust “dimensions” as defined in the original model: all such dimensions are modelled as elements of their respective class (e.g. `mt-tr:Action`, `mt-tr:Capability`, etc.) and are related by an object property (e.g. `mt-tr:hasAction`, `mt-tr:hasCapability`) to the reified `mt-tr:Trusts` relation.

The additional metatheory for data quality  $MT_{dq}$  is provided in a vocabulary (in *mt-dq.n3*) that implements the data quality measures from PlanetData deliverable D2.1 [7] as suggested in deliverable D31.2 [3]. In particular, most of the metrics are represented as functional data properties having domain in the `meta:Dataset` class (for example `mt-dq:LDSAvailability` representing the metric for availability of a resource), while “boolean” properties of a dataset are represented as membership to a specific class (for example, a dataset instance belongs to `mt-dq:HttpGetTurtle` only if it provides its representation as a RDF in Turtle syntax).

Finally, the metatheory for factuality  $MT_{fc}$  is provided in the vocabulary *mt-fc.n3*. The factuality value of dataset can be declared by asserting its membership to one of the classes `mt-fc:Factual`, `mt-fc:Counterfactual` or `mt-fc:NonFactual`. Similarly, the certainty value is asserted using the classes `mt-fc:Certain` or `mt-fc:Uncertain`. The class inclusion axioms provided in the schema implement the simple relations across factuality and certainty defined in [3].

### 3.3 MetaReasons ruleset and strategy

Based on this definition of symbols provided by the presented vocabularies, we can now briefly present how we implemented as a SPRINGLES ruleset<sup>4</sup> the deduction rules and strategy provided by the datalog translation in deliverable D31.1 [2] (and the additional inference rules for metatheories in D31.2 [3]). In the following we will walk through the definition of the ruleset and intuitively present the role of different phases of the evaluation strategy. After the definition of prefixes, the ruleset definition introduces the sequence of four steps that represents the main plan for rule evaluation:

```
:plan
  spr:bind "?metakb_inf = #INF_IRI(meta:metakb)" ; # Binds variable for metakb
                                     inferred graph to graph name
  spr:sequenceOf (
    :step1_metakb_dependencies # Step 1: compute module associations for metakb
    :step2_metakb_closure     # Step 2: local closure for metakb
    :step3_dataset_dependencies # Step 3: compute module associations for datasets
```

<sup>3</sup><http://www.w3.org/ns/prov-o>

<sup>4</sup>The resulting ruleset *metareasons\_ruleset.ttl* is provided in the demo archive in the folder *02\_rulesets*. The ruleset is also preloaded as a custom dataset in the demo instance of SPRINGLES.

```

      :step4_dataset_closure      # Step 4: local closure for datasets
    ) .

```

First step of the sequence computes the association of the inferred graph to the initial graph for `meta:metakb`. It is in fact composed by a single rule:

```

:step1_metakb_dependencies spr:evalOf ( :dep-metakb ) .

```

Intuitively, the rule `:dep-metakb` associates the metalevel inference graph to the initial graph and itself using the property `meta:derivedFrom`: in the evaluation of the rules, this is useful to determine all of the graphs referring to the metalevel (i.e. the original graph and the inferences graph).

The second step in the plan is `:step2_metakb_closure`: it represents the application of local inferences at the metalevel and it is defined by the following subtasks:

```

:step2_metakb_closure
  spr:bind "?g_inf = ?metakb_inf" ; # binds inference module to metakb inferences module
  spr:sequenceOf (
    :task_add_axioms      # Adds all OWL-RL axioms to metakb
    [ spr:fixPointOf [ spr:sequenceOf (
      :task_compute_rl_closure      # Computes closure for OWL-RL rules
      :task_compute_module_association # Adds inferred module to datasets
      :task_compute_dataset_disj     # Adds disjunction for all declared dataset names
      :task_compute_mtac_closure     # Computes metalevel closure for MTac metatheory
      :task_compute_mtrbac_closure   # Computes metalevel closure for MTrbac metatheory
      :task_compute_mtwp_closure     # Computes metalevel closure for MTwp metatheory
      :task_compute_mthp_closure     # Computes metalevel closure for MThp metatheory
    ) ] ]
  ) .

```

Intuitively, the first subtask of the sequence (`:task_add_axioms`) adds to the metalevel KB (i.e. in its inferred graph) all the initial OWL RL defining axioms. Then, as a fixpoint calculation, such sequence is computed: in a first step, all of the rules for the inference of *SRQI*-RL inferences are evaluated (corresponding to the rules of  $P_{rl}$  in Deliverable D31.1 [2]). After this, in `:task_compute_module_association` datasets are recognized and an inferred module is created and associated to each of them, storing this information in the metalevel KB. Similarly, each dataset is declared disjoint from all the others in the step `:task_compute_dataset_disj`, corresponding to the metalevel rule in  $I_{meta}$ . With this, the metalevel rules specific to the two implemented metatheories can be applied: for example, in `:task_compute_mtac_closure`, the following rule corresponding to (*pam-comp1*) is applied:

```

:pam-comp1 a spr:Rule ;
  spr:head "" GRAPH ?metakb_inf { ?d1 mtac:isComposedFrom ?d2 } "" ;
  spr:body "" GRAPH ?g { ?d1 mtd:derivedFrom ?d2 }
    GRAPH ?metakb_inf { ?metakb_inf meta:derivedFrom ?g }
    FILTER NOT EXISTS {
      GRAPH ?g0 { ?d1 mtac:isComposedFrom ?d2 }
      GRAPH ?metakb_inf { ?metakb_inf meta:derivedFrom ?g0 }
    } "" .

```

Intuitively, this rule states that if a dataset  $d1$  is `mtd:derivedFrom` a dataset  $d2$  in one of the graphs for the metalevel, then we can infer (in the inferred graph for the metalevel) that  $d1$  is also `mtac:isComposedFrom`  $d2$  (if this has not been derived before).

After the computation of the fixpoint for such rules, the computation for the metalevel terminates and the following steps regard the inference at the object level.

In the following step of the plan, `:step3_dataset_dependencies`, the rule `:dep-local-ds` is evaluated: this rule adds to the local inference graphs of each dataset the reference to the inferred graph, previously computed and stored in `meta:metakb`. As in the case of the metalevel KB, this is useful in the application of the local rules for identifying the graphs to be considered for each of the datasets (in particular in the evaluation of the body of the rules).

Finally, the last step in the plan, `:step4_dataset_closure`, provides the strategy to compute the local inferences inside datasets.

```

:step4_dataset_closure spr:fixPointOf [ spr:sequenceOf (
  :task_compute_rl_closure # Computes closure for OWL-RL rules
  :task_compute_mtd_closure # Computes knowledge level closure for MTd metatheory
) ] .

```

As in the case of the metalevel, a fixpoint iteration is computed: in the first sub-step of the sequence, all *SROIQ*-RL rules are applied to derive the local inferences. The following subtask `:task_compute_mtd_closure` evaluates the rule for the propagation to “derived” datasets, corresponding to the rules in  $P_{der-obj}$ . For example, the following rule corresponds to the propagation of a subsumption:

```

:ppo-subc a spr:Rule ;
  spr:head "" GRAPH ?g_inf { ?x rdf:type ?z } "" ;
  spr:body "" GRAPH ?g1 { ?y rdfs:subClassOf ?z }
    GRAPH ?g2 { ?x rdf:type ?y }
    GRAPH ?g3 { ?di mtd:derivedFrom ?g1 }
    GRAPH ?g_inf { ?g_inf meta:derivedFrom ?di, ?g2 }
    GRAPH ?metakb_inf { ?metakb_inf meta:derivedFrom ?g3 }
    FILTER NOT EXISTS {
      GRAPH ?g0 { ?x rdf:type ?z }
      GRAPH ?g_inf { ?g_inf meta:derivedFrom ?g0 }
    } "" .

```

This rule can be read as follows: if  $A \sqsubseteq B$  in a graph  $g1$ , where  $di$  is `mtd:derivedFrom`  $g1$  (in the metalevel), then, if  $A(x)$  in  $di$  (or its inferences graph), propagate  $B(x)$  to the inferences graph of  $di$ .

After the termination of such fixpoint operation over the object knowledge, the plan terminates and the inference is completed. We remark that, apart from technical details for the management of graphs and their dependencies, the implemented plan basically follows the strategy proposed in deliverable D31.1 [2] for the translation to the UKB program representing the whole input unified knowledge base.

## 3.4 Examples

### 3.4.1 Combining where provenance and access control information

The demo of the system we distribute with this deliverable contains an implementation for the university example we considered in previous deliverable D31.1 [2]. Goal of this example is to present the form of the input knowledge bases (now implemented in different named graphs) and to demonstrate the application of inference rules. The university example repository is contained in the distributed instance of the platform (repository *MetaReasonsExample*) and the input files can be found in the folder *04\_testcases/01\_university-example*.

Following the original example, we want to model a system for the management of users information in the context of an university. In the example, we have four datasets:  $d_o$ , representing the global ontology of the university,  $d_a$ , containing data about the student user *alice*,  $d_b$ , containing data about the professor user *bob*, and  $d_i$ , containing inferences from the combined sources. In the RDF representation, these datasets are declared in the `meta:metakb` knowledge base:

```

:d_o a meta:Dataset .
:d_a a meta:Dataset .
:d_b a meta:Dataset .
:d_i a meta:Dataset .

```

Using the definition for the  $MT_d$  metatheory, we can assert in the metalevel graph the information about the derivability of  $d_i$  as follows:

```

:d_i mtd:derivedFrom :d_o .
:d_i mtd:derivedFrom :d_a .
:d_i mtd:derivedFrom :d_b .

```

Similarly, the information about initial datasets with respect to the metatheories for access control  $MT_{ac}$  and provenance  $MT_{wp}$  can be encoded in the metalevel knowledge base as follows:

```

:c1 a mtwp:Color .           :a1 a mtac:AccessToken .
:c2 a mtwp:Color .           :a2 a mtac:AccessToken .

:d_o mtwp:hasColor :c1 .     :d_o mtac:hasLabel :a1 .
:d_a mtwp:hasColor :c2 .     :d_a mtac:hasLabel :a2 .
:d_b mtwp:hasColor :c2 .     :d_b mtac:hasLabel :a3 .

```

In the example, using the previous information about datasets, we want to model policies governing the visibility of datasets from the point of view of the two example users *alice* and *bob*. We can easily transform the axioms from the description logics based representation of [2] to their RDF serialization of the respective OWL axioms, taking care to express them inside the proposed normal form. For example, the policies referring to access control metatheory and defining the denied datasets for *alice* can be defined with the following OWL axioms, asserted in the metalevel KB:

```

:a1 :isDeniedLabel :alice .

:DeniedAliceLabel a owl:Class .
:Alice a :Class .
:alice a :Alice.
[ a owl:Restriction ;
  owl:onProperty :isDeniedLabel ;
  owl:someValuesFrom :Alice
] rdfs:subClassOf :DeniedAliceLabel .

:DeniedAliceDataset a owl:Class .
[ a owl:Restriction ;
  owl:onProperty mtac:hasLabel ;
  owl:someValuesFrom :DeniedAliceLabel
] rdfs:subClassOf :DeniedAliceDataset .

```

Previous statements intuitively state that *a1* is a denied label for *alice*, that every label that is denied for *alice* is classified as `:DeniedAliceLabel` and that every dataset that has some label denied for *alice* is a `:DeniedAliceDataset`. With respect to derived datasets, we have that if a dataset is composed from a dataset that is denied for the user, then also the derived dataset is denied:

```

[ a owl:Restriction ;
  owl:onProperty mtac:isComposedFrom ;
  owl:someValuesFrom :DeniedAliceDataset
] rdfs:subClassOf :DeniedAliceDataset .

```

As in the original example, analogous axioms are used to define the class `:DeniedBobDataset`: moreover, by similar axioms over `mtwp:hasColor` and `mtwp:hasDefiningColor` we can define the classes for `:UntrustedAliceDataset` and `:UntrustedBobDataset`. We can finally express that untrusted and denied dataset are not visible for *alice* as:

```

:DeniedAliceDataset rdfs:subClassOf [
  owl:complementOf :VisibleAliceDataset ] .
:UntrustedAliceDataset rdfs:subClassOf [
  owl:complementOf :VisibleAliceDataset ] .

```

By computing the inference closure by the presented plan, it is easy to check that the prototype is able to derive that, as expected, both the initial dataset  $d_o$  and the derived dataset  $d_i$  are classified as `:DeniedAliceDataset`. This can be easily verified by executing the SPARQL query (e.g. from the SPRINGLES user interface, on *Query* tab):

```

SELECT ?d
WHERE {?d a meta:Dataset , :DeniedAliceDataset}

```

At the object level we can similarly show that the rules for  $MT_d$  permit to propagate knowledge to the derived dataset  $d_i$ . In particular, we have that the university ontology dataset  $d_o$  contains:

```

d_o {
  :Agent a owl:Class .
  :Person a owl:Class ;
  rdfs:subClassOf :Agent .
}

```

```

:Student a owl:Class ;
    rdfs:subClassOf :Person .
:Professor a owl:Class ;
    rdfs:subClassOf :Person .
}

```

While the graphs defining datasets for *alice* and *bob* are defined as follows:

```

:d_a {
  :Student a owl:Class .
  :alice a :Student .
}

:d_b {
  :Professor a owl:Class .
  :bob a :Professor .
}

```

The graph relative to  $d_i$  empty (for simplicity of presentation). It is easy to verify that, by the encoding of propagation rules in  $P_{der-obj}$  we obtain  $\{ :alice\ a\ :Person,\ :Agent\}$  and  $\{ :bob\ a\ :Person,\ :Agent\}$  in the inference graph for  $d_i$ . This can be checked by the *Contexts* tab in the SPRINGLES user interface or by asking the following query in the *Query* tab:

```

SELECT ?a ?o
WHERE {
  GRAPH ?g_inf { ?g_inf meta:derivedFrom :d_i }
  GRAPH ?g_inf {?a a ?o}
  GRAPH meta:metakb {?a a :User}
}

```

### 3.4.2 Combining PROV-O how provenance and ROWLBAC access control information

The second example repository we propose is a revision of the previous example that uses  $MT_{rbac}$  to model access control and  $MT_{hp}$  to model provenance. Intuitively, it corresponds to a combination of the examples for  $MT_{rbac}$  and  $MT_{hp}$  provided in [3]. The example repository is contained in repository *MetaReasonsExample2* and the input files are distributed in the folder *04\_testcases/04\_university-example2*.

Following the original example, we still have the four datasets  $d_o$  containing the university ontology,  $d_a$  representing the data of the student user *alice*,  $d_b$  for the data of professor user *bob* and  $d_i$  containing inferences from the combined sources:

```

:d_o a meta:Dataset .
:d_a a meta:Dataset .
:d_b a meta:Dataset .
:d_i a meta:Dataset .

```

Using the definition for the  $MT_d$  metatheory, we can again assert in the metalevel graph the information about the derivability of  $d_i$  as follows:

```

:d_i mtd:derivedFrom :d_o .
:d_i mtd:derivedFrom :d_a .
:d_i mtd:derivedFrom :d_b .

```

With respect to provenance information encoded in  $MT_{hp}$  (PROV-O metatheory), we define two activities performed by the agents of the system which correspond to the creation and acceptance (validation) of a dataset:

```

:Create a owl:Class ;
    rdfs:subClassOf mthp:Activity .
:Accept a owl:Class ;
    rdfs:subClassOf mthp:Activity .

```

We declare four agents, corresponding to the two users, the secretary (*sec*) and university administrator (*univ*).

```
:alice a mthp:Agent .
:bob   a mthp:Agent .
:univ  a mthp:Agent .
:sec   a mthp:Agent .
```

Then we define the activities regarding the creation and acceptance of the datasets and we associate them to their subject agents and used datasets. For example, the acceptance (i.e. validation activity) of the dataset  $d_a$  by *alice* is defined by:

```
:accept-da a :Accept .
:accept-da mthp:wasAssociatedWith :alice .
:accept-da mthp:used :d_a .
```

On the other hand, in the access control metatheory  $MT_{rbac}$  (ROWLBAC metatheory) we model the actions and permitted actions for the considered users. In particular, for *alice* we define the class of read access to her data and we assert that she has the active role of reader for her data:

```
:ReadAlice a owl:Class ;
  rdfs:subClassOf mtrbac:Action .
:PermittedReadAlice a owl:Class ;
  rdfs:subClassOf mtrbac:PermittedAction .

:ActiveAliceReader a owl:Class ;
  rdfs:subClassOf mtrbac:ActiveRole .
:alice a :ActiveAliceReader .
```

We state that a `:ReadAlice` is a `:PermittedReadAlice` if it is performed by an `:ActiveAliceReader` and has as object the dataset  $d_a$ . This can be expressed by translating to OWL the DL axiom

$$\text{ReadAlice} \sqcap \exists \text{subject.ActiveAliceReader} \sqcap \exists \text{object.}\{d_a\} \sqsubseteq \text{PermittedReadAlice}$$

(cfr. [3]) taking care to express it inside the proposed normal form.

```
:ActiveAliceRead a owl:Class .
[ a owl:Restriction ;
  owl:onProperty mtrbac:subject ;
  owl:someValuesFrom :ActiveAliceReader
] rdfs:subClassOf :ActiveAliceRead .

:Da a owl:Class .
:d_a a :Da .
:DaAction a owl:Class .
[ a owl:Restriction ;
  owl:onProperty mtrbac:object ;
  owl:someValuesFrom :Da
] rdfs:subClassOf :DaAction .

:AliceDaRead a owl:Class .
[ owl:intersectionOf ( :DaAction :ActiveAliceRead ) ]
  rdfs:subClassOf :AliceDaRead .

[ owl:intersectionOf ( :AliceDaRead :ReadAlice ) ]
  rdfs:subClassOf :PermittedReadAlice .
```

Note that we have to introduce the additional class `:AliceDaRead` since the current rules for *SROIQ*-RL only support intersections of size 2.

In the example, using the previous information about datasets, we want to model policies governing the reliability of datasets from the point of view of the two users *alice* and *bob*. In particular, we want to include in reliable datasets the ones that are read accessible and accepted by the users. Thus, with respect to access control, we define as follows the read-accessible datasets for *alice*:

```
:ReadableAliceDataset a owl:Class .
[ a owl:Restriction ;
  owl:onProperty :isObjectOf;
  owl:someValuesFrom :PermittedReadAlice
] rdfs:subClassOf :ReadableAliceDataset .
```

where `isObjectOf` is the inverse of `mtrbac:object`. Similarly, for provenance we declare as follows the accepted datasets for the user *alice*:

```
:AssociatedAliceActivity a owl:Class .
[ a owl:Restriction ;
  owl:onProperty mthp:wasAssociatedWith ;
  owl:someValuesFrom :Alice
] rdfs:subClassOf :AssociatedAliceActivity .

:AssociatedAliceAccept a owl:Class .
[ owl:intersectionOf ( :Accept :AssociatedAliceActivity ) ]
  rdfs:subClassOf :AssociatedAliceAccept .

:AcceptedAliceDataset a owl:Class .
[ a owl:Restriction ;
  owl:onProperty :usedBy ;
  owl:someValuesFrom :AssociatedAliceAccept
] rdfs:subClassOf :AcceptedAliceDataset .
```

where `:usedBy` is the inverse of `mthp:used`. Finally, reliable datasets for *alice* are defined by the intersection of accepted and read-accessible datasets:

```
:ReliableAliceDataset a owl:Class .
[ owl:intersectionOf ( :AcceptedAliceDataset :ReadableAliceDataset ) ]
  rdfs:subClassOf :ReliableAliceDataset .
```

We then add a read operation on *alice* data:

```
:read01 a :ReadAlice .
:read01 mtrbac:subject :alice .
:read01 mtrbac:object :d_a .
```

(and similarly for *bob*) and we can ask to verify that the accessed dataset is reliable.

By computing the inference closure by the presented ruleset, it is easy to check that the prototype is able to derive that, as expected, only the dataset  $d_a$  is considered reliable by *alice*. This can be easily verified by executing the SPARQL query:

```
SELECT ?d
WHERE {?d a meta:Dataset , :ReliableAliceDataset}
```

The same can be verified for `:ReliableBobDataset` and the dataset  $d_b$ , for which similar definitions are provided.

Moreover, we note that information at the object level is modelled analogously to previous example: it can be checked that object level inference also produces similar results, in particular w.r.t. the knowledge propagation to the dataset  $d_i$  and the object level query proposed in previous example.

## 4 CONCLUSIONS

In this deliverable we presented the results of the activity for the prototype implementation of MetaReasons framework. We first presented SPRINGLES, the software layer we developed and adopted for the representation and reasoning over multiple RDF named graphs, and we shown how MetaReasons can be implemented using its primitives. We then provided details about the implementation of architecture and metatheories schemas and the definition of the ruleset and strategy for rule based inferences.

In the last tasks of our activity (corresponding to WP33), we will evaluate the performances of the final prototype, in particular with respect to the scalability of the inference mechanism. In particular, we will experimentally evaluate the performances and scalability of the reasoning procedure with respect to different reasoning regimes (e.g. RDFS or OWL RL, both for the meta and object level reasoning) and different solutions for RDF storage (e.g. built-in Sesame storage or OWLIM, in-memory or with storage in secondary memory). The evaluation will be carried out both on synthetic repositories and real datasets: in particular, we may consider real usecases and datasets possibly available in the partnered ICT program, allowing us to assess the usefulness of the proposed framework in different usage scenarios.

## BIBLIOGRAPHY

- [1] Loris Bozzato and Luciano Serafini. Materialization Calculus for Contexts in the Semantic Web. In *DL2013*, CEUR-WP. CEUR-WS.org, 2013.
- [2] Loris Bozzato and Luciano Serafini. D31.1 MetaReasons: theoretical architecture description. Deliverable D31.1, PlanetData, January 2014. <http://planet-data-wiki.sti2.at/web/D31.1>.
- [3] Loris Bozzato and Luciano Serafini. D31.2 MetaReasons: metatheories list and description. Deliverable D31.2, PlanetData, March 2014. <http://planet-data-wiki.sti2.at/web/D31.2>.
- [4] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.
- [5] Timothy W. Finin, Anupam Joshi, Lalana Kagal, Jianwei Niu, Ravi S. Sandhu, William H. Winsborough, and Bhavani M. Thuraisingham. ROWLBAC: representing role based access control in OWL. In *SACMAT 2008*, pages 73–82, 2008.
- [6] Deborah McGuinness, Timothy Lebo, and Satya Sahoo. PROV-O: The PROV ontology. W3C recommendation, W3C, April 2013. <http://www.w3.org/TR/2013/REC-prov-o-20130430/>.
- [7] Pablo N. Mendes, Christian Bizer, Zoltán Miklos, Jean-Paul Calbimonte, Alexandra Moraru, and Giorgos Flouris. D2.1 Conceptual model and best practices for high-quality metadata publishing. Deliverable D2.1, PlanetData, March 2012. <http://planet-data-wiki.sti2.at/web/D2.1>.
- [8] Lea Viljanen. Towards an ontology of trust. In *TrustBus 2005*, volume 3592 of *Lecture Notes in Computer Science*, pages 175–184. Springer, 2005.