oneM2M Technical Report

Document Number	TR-0033 -V3.0.0
Document Name:	Study on Enhanced Semantic Enablement
Date:	2019-05-13
Abstract:	In this study requirements on enhanced semantic enablement and approaches for addressing these requirements will be developed and discussed. The intention is to achieve agreement between the interested participants on the approaches to be pursued in oneM2M. On this basis normative contributions to Technical Specifications can then be made.

Template Version: January 2017 (Do not modify)

About oneM2M

The purpose and goal of oneM2M is to develop technical specifications which address the need for a common M2M Service Layer that can be readily embedded within various hardware and software, and relied upon to connect the myriad of devices in the field with M2M application servers worldwide.

More information about oneM2M may be found at: http//www.oneM2M.org

Notice of Disclaimer & Limitation of Liability

The information provided in this document is directed solely to professionals who have the appropriate degree of experience to understand and interpret its contents in accordance with generally accepted engineering or other professional standards and applicable regulations. No recommendation as to products or vendors is made or should be implied.

NO REPRESENTATION OR WARRANTY IS MADE THAT THE INFORMATION IS TECHNICALLY ACCURATE OR SUFFICIENT OR CONFORMS TO ANY STATUTE, GOVERNMENTAL RULE OR REGULATION, AND FURTHER, NO REPRESENTATION OR WARRANTY IS MADE OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR AGAINST INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS. NO oneM2M PARTNER TYPE 1 SHALL BE LIABLE, BEYOND THE AMOUNT OF ANY SUM RECEIVED IN PAYMENT BY THAT PARTNER FOR THIS DOCUMENT, WITH RESPECT TO ANY CLAIM, AND IN NO EVENT SHALL oneM2M BE LIABLE FOR LOST PROFITS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES. oneM2M EXPRESSLY ADVISES ANY AND ALL USE OF OR RELIANCE UPON THIS INFORMATION PROVIDED IN THIS DOCUMENT IS AT THE RISK OF THE USER.

Contents

1 Scope

The present document discusses proposed solutions for semantic oneM2M features. The solutions covered fulfil requirements related to semantics that have been identified for oneM2M Release 3. It serves as the basis for selecting agreed solutions, which are then put into the fitting oneM2M Technical Specifications. Due to the progress of normative work, there may be inconsistencies between what is described in this document and the normative oneM2M technical specifications, which take precedence.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] oneM2M Drafting Rules.

NOTE: Available at http://www.onem2m.org/images/files/oneM2M-Drafting-Rules.pdf.

[i.2] oneM2M TS-0002: "onM2M Requirements".

[i.3] oneM2M TS-0001: "Functional Architecture".

[i.4] oneM2M TS-0004: "Service Layer Core Protocol Specification".

[i.5] W3C Recommendation: "OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax".

NOTE: Available at http://www.w3.org/TR/owl2-syntax/#IRIs.

[i.6] oneM2M TS-0012: "Base Ontology".

[i.7] oneM2M TS-0030: "Generic Interworking".

[i.8] W3C Recommendation: "RDF 1.1 Concepts and Abstract Syntax".

NOTE: Available at http://www.w3.org/TR/rdf11-concepts/.

[i.9] W3C Recommendation: "RDF 1.1 XML Syntax".

NOTE: Available at https://www.w3.org/TR/rdf-syntax-grammar/.

[i.10] W3C Recommendation: "OWL Web Ontology Language Semantics and Abstract Syntax".

NOTE: Available at http://www.w3.org/TR/owl-semantics/.

[i.11] ETSI TS 103 264: "SmartM2M; Smart Appliances; Reference Ontology and oneM2M Mapping".

[i.12] oneM2M TS-0034: "Semantics Support".

[i.13] oneM2M TR-0007: "Study on Abstraction and Semantics Enablement".

[i.14] W3C Recommendation: "SPARQL Query Language for RDF".

NOTE: Available at http://www.w3.org/TR/rdf-sparql-query/.

[i.15] IETF RFC 3987: "Internationalized Resource Identifiers (IRIs)".

NOTE: Available at https://www.ietf.org/rfc/rfc3987.txt.

3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ACP	Access Control Policy
AE	Application Entity
CSE	Common Services Entity
CSF	Common Services Function
FOAF	Friend Of A Friend
HGI	Home Gateway Initiative
HTTP	Hypertext Transfer Protocol
IN	Infrastructure Node
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
M2M	Machine-to-Machine
MA	Mandatory Announced
Mca	Reference Point for M2M Communication with AE
Mcc	Reference Point for M2M Communication with CSE
Mcc'	Reference Point for M2M Communication with CSE of different M2M Service Provider
MR	Mashup Requester
NA	Not Announced
OA	Optional Announced
OWL	Web Ontology Language
RDBMS	Relational Data Base Management System
RDF	Resource Description Framework
RH	Resource Host
RO	Read Only
RW	Read Write
SAREF	Smart Applicance REFerence Ontology
SD	Semantic Descriptor
SEM	Semantics CSF
SGS	Semantic Graph Scoping
SMF	Semantic Mashup Function
SMI	Semantic Mashup Instance
SMJP	Semantic Mashup Job Profile
SMR	Semantic Mashup Resource
SMS	Semantic Mashup System
SPARQL	*SPARQL* Protocol and RDF Query Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WO	Write Once
XML	Extensible Markup Language

4 Conventions

The key words "Shall", "Shall not", "May", "Need not", "Should", "Should not" in this document are to be interpreted as described in the oneM2M Drafting Rules [i.1].

5 Requirements

5.1 Overview

Table 5.1-1 lists the requirements on Abstraction and Semantics. It is based on those identified in oneM2M TS-0002 [i.2]. As the result of the work described in this Technical Report, the original requirements have been refined, new requirements have been added and the overall consistency has been checked. The requirements have been categorized according to different key semantic aspects, i.e. semantics annotation, ontology, semantics query, semantics mashup, data analytics and semantic reasoning.

Table 5.1-1: Requirements on abstraction and semantics

No.	Category	Requirement
1	Semantics Annotation	The M2M System shall provide capabilities to manage semantic information about the oneM2M resources, e.g. create, retrieve, update, delete, associate/link.
2	Ontology	The M2M System shall support modelling semantic descriptions of Things (including relationships among them) by using ontologies.
3	Semantics Annotation	The M2M System shall support a common language for semantic description, e.g. RDF.
4	Ontology	The M2M System shall support a common modeling language for ontologies (e.g. OWL).
5	Ontology	The M2M System should be able to provide translation capabilities from different modeling languages for ontologies to the language adopted by oneM2M if the expressiveness of the imported ontology allows.
6	Semantics Query	The M2M System shall provide capabilities to discover M2M Resources based on semantic descriptions.
7	Ontology	The M2M System shall provide the capability to retrieve semantic descriptions and ontologies stored outside of the M2M System.
8	Data Analytics	The M2M System shall be able to support capabilities (e.g. processing function) for performing M2M data analytics based on semantic descriptions from M2M Applications and /or from the M2M System.
9	Semantics Mashup	The M2M system shall provide the capability to host processing functions for mash-up.
10	Semantics Mashup	The M2M system shall enable Applications to provide processing functions for mash-up.
11	Semantics Mashup	The M2M system itself may provide pre-provisioned or dynamically created processing functions for mash-up.
12	Semantics Mashup	The M2M system shall be able to create and execute mash-ups based on processing functions.
13	Semantics Mashup	The M2M system shall be able to expose mash-ups as resources e.g. virtual devices.
14	Semantics Annotation	The M2M System shall support semantic annotation of oneM2M resources for example application related data contained in containers.
15	Semantics Annotation	The M2M System shall support semantic annotation based on related ontologies.
16	Ontology	The M2M System shall provide support for linking ontologies defined in the context of the M2M system with ontologies defined outside this context.
17	Ontology	The M2M System shall be able to support extending ontologies in the M2M system.
18	Ontology	The M2M System shall be able to use ontologies that contain concepts representing aspects (e.g. a room) that are not represented by resources of the M2M System.
19	Ontology	The M2M System shall be able to re-use common ontologies (e.g. location, time ontologies, etc.) which are commonly used in M2M Applications.
20	Ontology	The M2M System shall be able to support simultaneous usage of multiple ontologies for the same M2M resource.
21	Semantics Reasoning	The M2M system shall be able to update ontologies as a result of the ontology reasoning.
22	Ontology	The M2M system shall provide the capability for making ontology available in the M2M System, e.g. through announcement.
23	Ontology	The M2M system shall be able to support mechanisms to import external ontologies into the M2M system.
24	Ontology	The M2M System shall be able to support update of ontologies.

No.	Category	Requirement
25	Semantics Reasoning	The M2M System shall be able to support semantic reasoning e.g. ontology reasoning or semantic rule-based reasoning.
26	Semantics Reasoning	The M2M System shall be able to support adding and updating semantic information based on semantic reasoning.
27	Data Analytics	The M2M System shall provide the capability of interpreting and applying service logic (e.g. rules/policies of triggering operations upon other resources or attributes according to the change of the monitored resource) described with semantic annotation and ontology.
28	Data Analytics	The M2M system shall support a standardized format for the rules/policies used to define service logic.
29	Ontology	The M2M System shall enable functions for data conversion based on ontologies.
30	Semantics Annotation	The M2M System shall provide the capability for making semantic descriptions available in the M2M System, e.g. announcement.
31	Semantics Annotation	The M2M system shall enable applications to retrieve an ontology representation related to semantic information used in the M2M system.
32	Ontology	The M2M system shall be able to model devices based on ontologies which may be available outside the M2M system (e.g. HGI device template).
33	Ontology	The M2M System shall support storage, management and discovery of ontologies.

Figure 5.1-1 shows dependencies and relations between the requirements listed in table 5.5-1. It can be used as a basis for prioritizing the addressing of requirements and to make sure that all preconditions are addressed first.

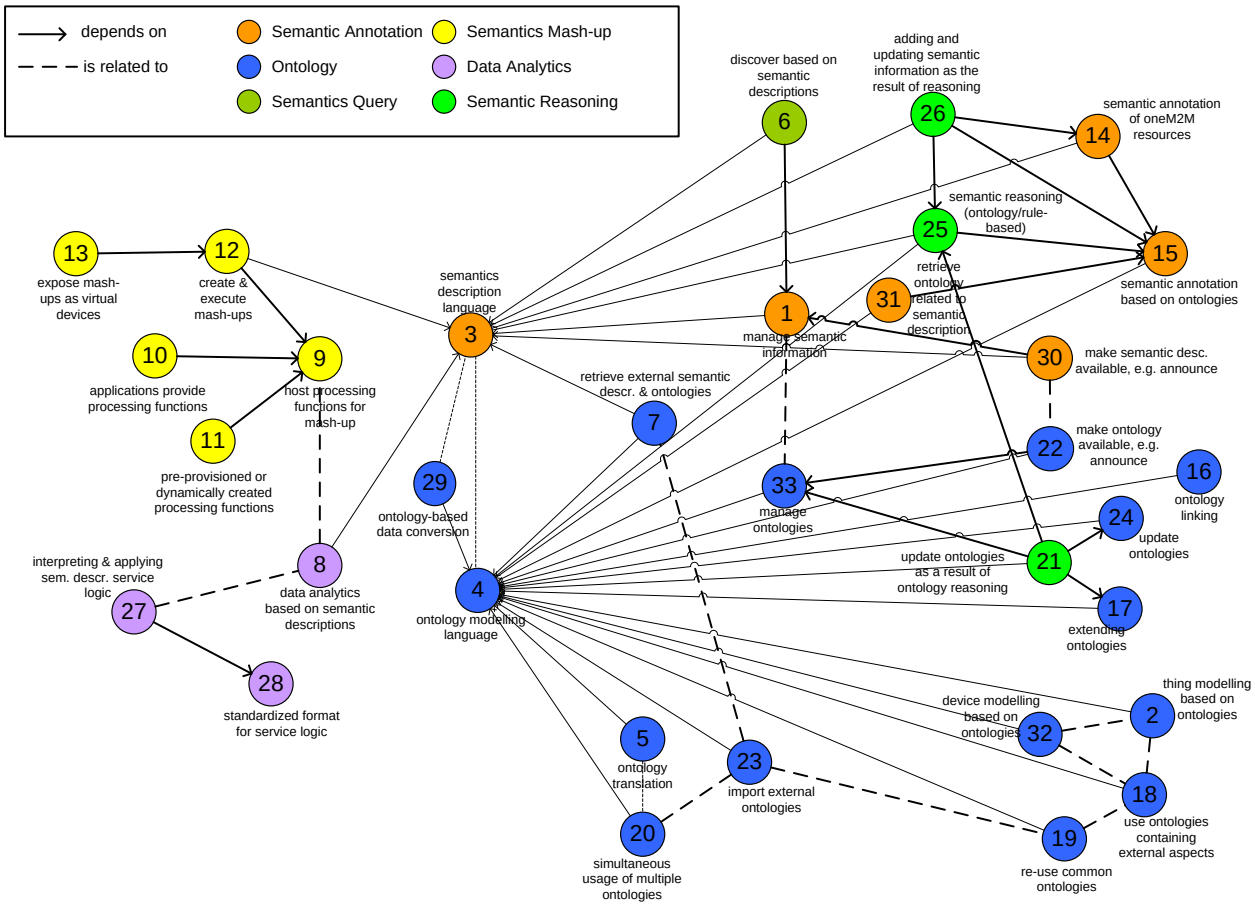

Figure 5.1-1: Dependencies and relations between requirements

6 Semantic modelling & ontologies

Semantic information is at the basis of all semantic features of oneM2M. Semantic information explicitly represents the meaning of what is stored in oneM2M resources, e.g. <AE>, <container> or <contentInstance> reosurces. The concepts and related properties according to which the semantic information is modelled are defined in ontologies. oneM2M does not restrict, which ontologies are used, but agreement on using the same ontologies across applications may enable reuse and increase the value provided by the oneM2M system. The Base Ontology [i.6] defined by oneM2M models those aspects that are needed for enabling Generic Interworking [i.7]. This means that the resource structure in oneM2M representing devices and servicessemantically modelled according to the Base Ontology - or a derived ontology - can be directly determined based on the semantic model.

Semantic information in oneM2M is modelled as RDF triples [i.8] and represented in the RDF/XML serialization of RDF [i.9]. Ontologies are modelled in OWL [i.10] and are also represented in the RDF/XML serialization.

7 Architecture for semantics

7.1 Introduction

The subclauses of clause 7 introduce the aspects of semantics related to architecture. Claus 7.2 introduces the high-level semantic functionality as defined in oneM2M TS-0001 [i.3]. Clause 7.3 introduces the resources related to semantics. Different implementation options are discussed in clause 7.4. Ways of implementing access control for semantic information, in particular if stored in triple stores, are presented in clause 7.5.

7.2 Semantic functionality in CSFs

Semantic functionalities are grouped in the Semantics Common Services Function (SEM CSF). The SEM CSF enables applications to manage semantic information and provides functionalities based on this information. Thus the SEM CSF brings value-added features related to the meaning of data and resources. The SEM CSF functionality is based on semantic descriptions and supports features such as: annotation, resource filtering and discovery, querying, validation, mash-up, reasoning, analytics, etc. The SEM CSF also provides input for Access Control applied to semantic content and is responsible for the management of ontologies. More details can be found in oneM2M TS-0001 [i.3].

7.3 Resources related to semantics

7.3.1 Resource <semanticDescriptor>

The <*semanticDescriptor*> resource is used to store a semantic description pertaining to a resource and potentially sub-resources. Such a description can be provided according to ontologies. The semantic information is used by the semantic functionalities of the oneM2M system and is also available to applications or CSEs.

Resources of several types (e.g. <AE>, <container>, <contentInstance>) optionally can have one or more semantic descriptor resources. See oneM2M TS-0001 [i.3], clause 9.6.1.1, for a complete list of resources which may have a <semanticDecriptor> child resource and oneM2M TS-0001 [i.3], clause 9.6.30, for a complete description of this resource.

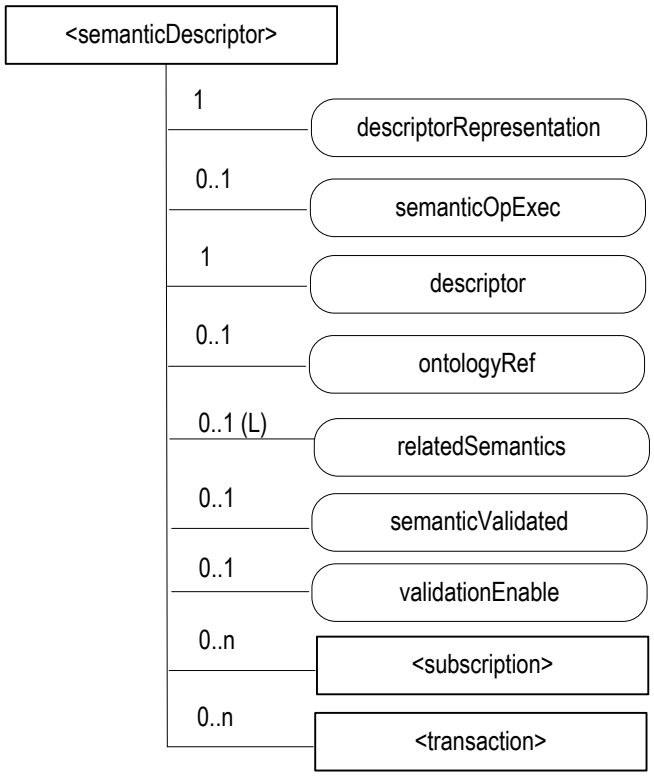

Figure 7.3.1-1: Structure of <*semanticDescriptor*> resource

The <*semanticDescriptor*> resource contains the child resources specified in table 7.3.1-1.

Table 7.3.1-1: Child resources of <*semanticDescriptor*> resource

Child Resources of <*semanticDescriptor*>	Child Resource Type	Multiplicity	Description	<*semanticDescriptorAnnc*> Child Resource Types
[variable]	<*subscription*>	0..n	See [i.3], clause 9.6.8 where the type of this resource is described.	<*subscription*>
[variable]	<*transaction*>	0..n	See [i.3], clause 9.6.48.	<*transaction*>

The <*semanticDescriptor*> resource contains the attributes specified in table 7.3.1-2.

Table 7.3.1-2: Attributes of <*semanticDescriptor*> resource

Attributes of <*semanticDescriptor*>	Multiplicity	RW/ RO/ WO	Description	<*semanticDescriptorAnnc*> Attributes
resourceType	1	RO	See [i.3], clause 9.6.1.3.	NA
resourceID	1	RO	See [i.3], clause 9.6.1.3.	NA
resourceName	1	WO	See [i.3], clause 9.6.1.3.	NA
parentID	1	RO	See [i.3], clause 9.6.1.3.	NA
accessControlPolicyIDs	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	MA
creationTime	1	RO	See [i.3], clause 9.6.1.3.	NA
expirationTime	1	RW	See [i.3], clause 9.6.1.3.	MA
lastModifiedTime	1	RO	See [i.3], clause 9.6.1.3.	NA
labels	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	MA
announceTo	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	NA
announcedAttribute	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	NA
dynamicAuthorizationConsultationIDs	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	OA
creator	0..1	RO	See [i.3], clause 9.6.1.3.	NA
descriptorRepresentation	1	RW	Indicates the type used for the serialization of the descriptor attribute, e.g. RDF serialized in XML.	OA
semanticOpExec	0..1	RW	This attribute cannot be retrieved. Contains a SPARQL query request for execution of semantic operations on the *descriptor* attribute e.g. SPARQL update as described in oneM2M TS-0004 [i.4].	NA
descriptor	1	RW	Stores a semantic description pertaining to a resource and potentially sub-resources. Such a description shall be according to subject-predicate-object triples as defined in the RDF graph-based data model [i.8]. The encoding of the RDF triples used in oneM2M is defined in oneM2M TS-0004 [i.4]. The elements of such triples may be provided according to ontologies.	OA
ontologyRef	0..1	WO	A reference (URI) of the ontology used to represent the information that is stored in the *descriptor attribute*. If this attribute is not present, the *ontologyRef* from the parent resource is used if present.	OA
relatedSemantics	0..1(L)	WO	List of resource identifiers containing related semantic information to be used in processing semantic queries. The esource identifiers may reference either a <*group*> resource or <*semanticDescriptor*> resources and <contentInstance> resources with semantic information in their content attributes as indicated by their contentInfo attribute. In the latter case, the resource identifier may reference a <latest> resource representing the most recent <contentInstance> in a container..	OA
semanticValidated	0..1	RO	A Boolean value representing the validation result of the triples in the *descriptor* attribute. The validation is against the referenced ontology as pointed by the *ontologyRef* attribute as well as other associated <*semanticDescriptor*> resources (and their referenced ontologies) linked by *relatedSemantics* attribute and triples in the *descriptor* attribute.	OA

Attributes of <*semanticDescriptor*>	Multiplicity	RW/ RO/ WO	Description	<*semanticDescriptorAnnc*> Attributes
validationEnable	0..1	RW	A Boolean value indicating whether the triples in the *descriptor* attribute needs to be validated by the hosting CSE. Note: the hosting CSE may override this value according to local policy to enforce or disable semantic validation despite the suggested value from the issuer.	OA

7.3.2 Example showing the uses of the Semantic Descriptor resource

This clause gives an example of how semantic annotations based on the Smart Appliance REFerence Ontology (SAREF) [i.11] can be used to describe an AE representing a smart appliance.

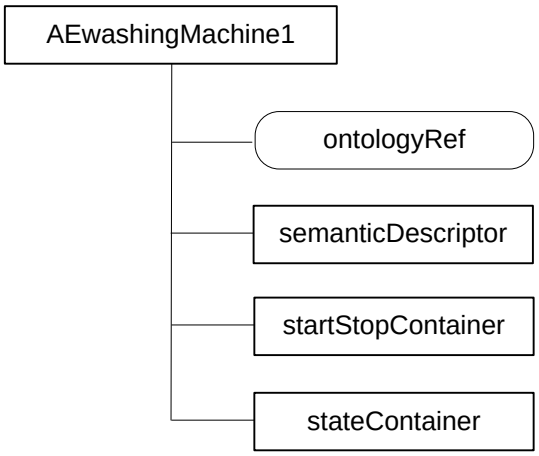

Figure 7.3.2-1: Resource structure of smart washing machine AE

Figure 7.3.2-1 shows the resource structure of an AE representing a smart washing machine.

NOTE: The assumption here is that the Washing Machine acts as an AE using the Mca interface. Here no distinction is made between the Washing Machine AE and the washing machine as a physical device. This can be done under the given assumption, but could lead to problems in other cases, e.g. if an interworking proxy is involved.

The resource includes an ontologyRef attribute, which contains the URI of the ontology concept of the smart washing machine, e.g. "http://ontology.tno.nl/saref#WashingMachine". The startStopContainer and the stateContainer represent the functional interface aspects of the washing machine, i.e. it can be started and stopped and the current state can be requested.

Table 7.3.2-1 shows the semantic annotation stored in the descriptor attribute of the semanticDescriptor resource. The information provides the link between the operations of the washing machines and the containers of the smart washing machine AE and describes the REST methods that can be executed on the containers. The washing operation can be started by executing a Create request on the startStopContainer whose URI is provided, the same for the state operation, where a Retrieve request on the *latest* contentInstance of the stateContainer will provide the current state of the washing machine.

Table 7.3.2-1: Semantic resource description of smart washing machine AE based on SAREF [i.11]

```
<rdf:RDF
   <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
      <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
      <saref:hasManufacturer>LG</saref:hasManufacturer>
      <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
       <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
      <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
      <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
   </rdf:Description>

  <rdf:Description rdf:about="http://ontology.tno.nl/saref#WashingService_123">
      <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingService"/>
      <msm:hasOperation rdf:resource="http://ontology.tno.nl/saref#WashingOperation_123"/>
   </rdf:Description>

  <rdf:Description rdf:about="http://ontology.tno.nl/saref#WashingOperation_123">
  <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingOperation"/>
      <hr:hasMethod>Create</hr:hasMethod>
      <hr:hasURITemplate>/CSE1/WASH_LG_123/startStopContainer </hr:hasURITemplate>
      <msm:hasInput rdf:resource="http://ontology.tno.nl/saref#Action"/>
   </rdf:Description>

   <rdf:Description rdf:about="http://ontology.tno.nl/saref#StateService123">
 <rdf:type rdf:resource="http://ontology.tno.nl/saref#StateService"/>
      <msm:hasOperation rdf:resource="http://ontology.tno.nl/saref#StateOperation123/>
   </rdf:Description>

   <rdf:Description rdf:about= http://ontology.tno.nl/saref#StateOperation123>
  <rdf:type rdf:resource="http://ontology.tno.nl/saref#StateOperation"/>
      <hr:hasMethod>Retrieve</hr:hasMethod>
      <hr:hasURITemplate>/CSE1/WASH_LG_123/state/stateContainer/latest</hr:hasURITemplate>
      <msm:hasOutput rdf:resource="http://ontology.tno.nl/saref#State"/>
   </rdf:Description>
</rdf:RDF>
```

7.3.4 *Proposed* Resource <semanticContentInstance>

This clause describes a proposal to define a <semanticContentInstance> resource that was not adopted in Release 3. Instead, it was decided to treat a <contentInstance> with semantic content (as identified by the contentInfo attribute)as a semantic resource with respect to semantic functionality, e.g. a semantic query would take into account the semantic content contained in the content attribute of a <contentInstance> resource if this resource is within the scope of the semantic query.

The <*semanticContentInstance*> resource represents a data instance in the <*container*> resource that contains semantic information as RDF triples.

Like the <*contentInstance*> resource, it shall not be modified once created. An AE shall be able to delete a *semanticContentInstance* resource explicitly or it may be deleted by the platform based on policies. If the platform has rentention policies for *contentInstance* resources, these shall also apply to *semanticContentInstance* resources and be represented by the attributes *maxByteSize*, *maxNrOfInstances* and/or *maxInstanceAge* attributes in the <*container*> resource. If multiple policies are in effect, the strictest policy shall apply.

The <*semanticContentInstance*> resource inherits the same access control policies of the parent <*container*> resource, and does not have its own *accessControlPolicyIDs* attribute.

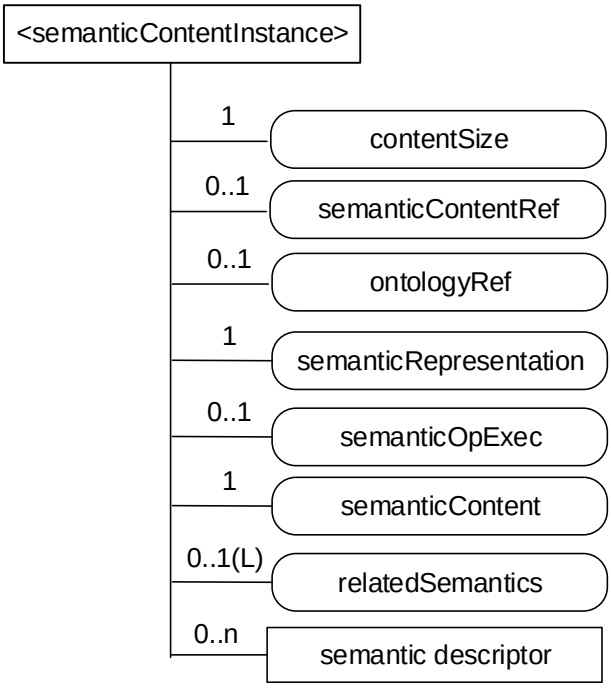

Figure 7.3.4-1: Structure of *<semanticContentInstance>* resource

The <*semanticContentInstance*> resource shall contain the child resources specified in table 7.3.4-1.

Table 7.3.4-1: Child resources of *<semanticContentInstance>* resource

Child Resources of *<semanticContentInstance>*	Child Resource Type	Multiplicity	Description	*<semanticContentInstanceAnnc>* Child Resource Types
[variable]	<*semanticDescriptor*>	0..n	See [i.3], clause 9.6.30	<*semanticDescriptor*>, <*semanticDescriptorAnnc*>

The <semanticContentInstance> resource shall contain the attributes specified in table 7.3.4-2.

Table 7.3.4-2: Attributes of *<semanticContentInstance>* resource

Attributes of *<semanticContentInstance>*	Multiplicity	RW/ RO/ WO	Description	*<semanticContentInstanceAnnc>* Attributes
resourceType	1	RO	See [i.3], clause 9.6.1.3.	NA
resourceID	1	RO	See [i.3], clause 9.6.1.3.	NA
resourceName	1	WO	See [i.3], clause 9.6.1.3.	NA
parentID	1	RO	See [i.3], clause 9.6.1.3.	NA
labels	0..1 (L)	WO	See [i.3], clause 9.6.1.3.	MA
expirationTime	1	WO	See [i.3], clause 9.6.1.3.	NA
creationTime	1	RO	See [i.3], clause 9.6.1.3.	NA
lastModifiedTime	1	RO	See [i.3], clause 9.6.1.3.	NA
stateTag	1	RO	See [i.3], clause 9.6.1.3. The *stateTag* attribute of the parent resource should be incremented first and copied into this *stateTag* attribute when a new instance is added to the parent resource.	OA
announceTo	0..1 (L)	WO	See [i.3], clause 9.6.1.3.	NA
announcedAttribute	0..1 (L)	WO	See [i.3], clause 9.6.1.3.	NA
dynamicAuthorizationConsultationIDs	0..1 (L)	RW	See [i.3], clause 9.6.1.3.	OA
creator	0..1	RO	See [i.3], clause 9.6.1.3.	NA
contentSize	1	RO	Size in bytes of the semanticC*ontent* attribute.	OA
semanticContentRef	0..1	RW	This attribute contains a list of name-value pairs. Each entry expresses an associative reference to a <semanticContentInstance> resource. The name of the entry indicates the relationship and the value of the entry the indicates reference (URI) to the resource.	OA
ontologyRef	0..1	WO	A reference (URI) of the ontology used to represent the information that is stored in the semanticC*ontentInstances* resources of the <*container*> *resource*. If this attribute is not present, the semanticC*ontentInstance* resource inherits the *ontologyRef* from the parent <*container*> resource if present (see note).	OA
semanticRepresentation	1	RW	Indicates the type used for the serialization of the semanticContent attribute, e.g. RDF serialized in XML	OA
semanticOpExec	0..1	RW	This attribute cannot be retrieved. Contains a SPARQL query request for execution of semantic operations on the *semanticContent* attribute e.g. SPARQL update as described in [i.4].	NA
semanticContent	1	WO	Actual content of the semanticC*ontentInstance*. Stores semantic content according to subject-predicate-object triples as defined in the RDF graph-based data model [8]. The encoding of the RDF triples used in oneM2M is defined in oneM2M TS-0004 [i.4]. The elements of such triples may be provided according to ontologies.	OA

Attributes of <*semanticContentInstance*>	Multiplicity	RW/ RO/ WO	Description	<*semanticContentInstanceAnnc*> Attributes
relatedSemantics	0..1(L)	WO	List of URIs for resources containing related semantic information to be used in processing semantic queries. The URI(s) may reference either a <*group*> resource or other *semantic* resources, i.e. <semanticDescriptor> or <semanticContentInstance> resources.	OA
NOTE: Access to this URI is out of scope of oneM2M.				

7.3.5 Resource <*ontologyRepository*>

An <*ontologyRepository*> resource is capable of storing multiple ontologies in the unified languages adopted by the M2M system, e.g. RDFS/OWL. For easy illustration of the examples, in this clause its is assumed that the M2M system adopts RDFS/OWL in describing ontologies.

This structure provides support for re-use of existing ontologies, the ability to access both internal and external ones and for ontology import into the system. It also allows to fulfil the requirements for ontology discovery, as well as addition and updates, via CRUD operations.

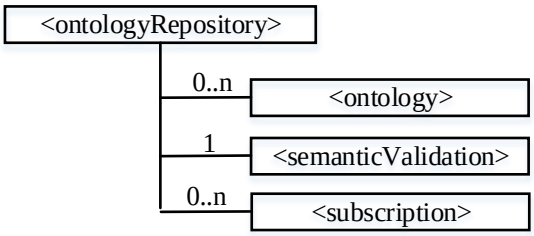

Figure 7.3.5-1: Structure of <*ontologyRepository*> resource

Resources of type <CSEBase> and <AE> optionally can have one <*ontologyRepository*> resource.The <*ontology*> resource is further described in clause 7.3.6.

The <*ontologyRepository*>resource may also contain a (virtual) sub-resource <*semanticValidation*> <*semanticValidation*> as the interface to accept semantic validation requestfrom an <AE> or a remote <CSE>. Upon receiving a Update request with <semanticDescriptor> resource representation addressing the <*semanticValidation*> <*semanticValidation*>> sub-resource, the hosting CSE performs the semantic validation procedures as described in clause 8.3.2.

The <*ontologyRepository*> resource shall contain the child resources as specified in table 7.3.5-1.

Table 7.3.5-1: Child resources of <*ontologyRepository*> resource

Child Resources of <*ontologyRepository*>	Child Resource Type	Multiplicity	Description	<*ontologyRepository*>Child Resource Types
[variable]	<*ontology*>	0..n	See clause 7.3.6	<*ontologyAnnc*>
smv	<*semanticValidation*>	1	See [i.3], clause 9.6.52	*None*
[variable]	<*subscription*>	0..n	See [i.3], clause 9.6.8	<*subscription*>

The < *ontologyRepository* > resource above contains the attributes specified in table 7.3.5-2.

Table 7.3.5-2: Attributes of *<ontologyRepository>* resource

Attribute Name	Multiplicity	RW/RO/WO	Description	*<ontologyRepositoryAnnc>* Attributes
resourceName	1	WO	See clause [i.3], clause 9.6.1.3.	NA
parentID	1	RO	See clause [i.3], clause 9.6.1.3.	NA
expirationTime	1	RW	See clause [i.3], clause 9.6.1.3.	NA
accessControlPolicyIDs	0..1 (L)	RW	See clause [i.3], clause 9.6.1.3.	NA
labels	0..1 (L)	RW	See clause [i.3], clause 9.6.1.3.	MA
creationTime	1	RO	See clause [i.3], clause 9.6.1.3.	MA
lastModifiedTime	1	RO	See clause [i.3], clause 9.6.1.3.	MA
announceTo	0..1 (L)	RW	See clause [i.3], clause 9.6.1.3.	NA
announcedAttribute	0..1 (L)	RW	See clause [i.3], clause 9.6.1.3.	NA
dynamicAuthorizationConsultationIDs	0..1 (L)	RW	See clause [i.3], clause 9.6.1.3.	OA
creator	0..1	RO	See clause [i.3], clause 9.6.1.3.	NA

7.3.6 Resource <ontology>

7.3.6.1 Introduction

The <*ontology*> resource is used to store the representation of an ontology. This representation may contain ontology descriptions in a variety of formats, given the requirements for re-use of existing ontologies, for support for ontologies available only externally and for support of ontology import into the system. The ontology description is made available to the semantic-related functions of the oneM2M system provided by applications or CSEs.

In the following, a number of examples are given what applications, but also the semantic functionalities supported by the oneM2M platform itself, may need from an ontology:

1) get all classes of an ontology;

2) get all object | data properties of ontology;

3) get direct subclasses of class A;

4) get also transitive subclasses class A;
e.g. if information from instances of class A is requested, all subclasses of class A also need to be included as they are also instances of class A;

5) get all the superclasses of class A;
e.g. if for derived ontologies the class of the base ontology needs to be found from which the class is derived, for example to apply rules defined for the base ontology, e.g. for creating a resource structure;

6) get all object | data properties where class A is in the domain;
e.g. to find out what properties an instance of class A can possibly have;

7) get all object | data properties where class A is in the range;

8) get all sub-properties of a property A;
e.g. if information concerning property A is requested all sub-properties of A also need to be included;

9) get classes that are equivalent to class A.

7.3.6.2 Resource definition

Using OWL 2.0 [i.5] as an ontology format example to be supported by the oneM2M system and based on W3C specifications (http://www.w3.org/TR/owl2-syntax/#IRIs) the following apply:

- *"Ontologies and their elements are identified using Internationalized Resource Identifiers (IRIs) [RFC3987]; thus, OWL 2 extends OWL 1, which uses Uniform Resource Identifiers (URIs). Each IRI MUST be absolute (i.e. not relative). In the structural specification, IRIs are represented by the IRI UML class. Two IRIs are structurally equivalent if and only if their string representations are identical."*

And

- *"Ontology documents are not represented in the structural specification of OWL 2, and the specification of OWL 2 makes only the following two assumptions about their nature:*

  - *Each ontology document can be accessed via an IRI by means of an appropriate protocol.*

  - *Each ontology document can be converted in some well-defined way into an ontology (i.e. into an instance of the Ontology UML class from the structural specification)."*

Therefore current methods of accessing and importing ontologies requires access to the respective ontology document via an IRI (Internationalized Resource Identifiers) as specified in IETF RFC 3987 [i.15]. Given that access to the ontology document has been obtained, this approach also provides for local storage of the document in a *content* attribute which is available to the platform based on access control rules.

Given the possible need to have access to multiple versions of an ontology, and to different formats, a specialized attribute *contentFormat* provides information necessary for the system to interpret the information available in the *content* attribute.

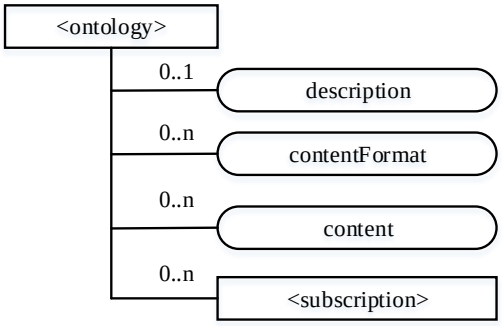

Figure 7.3.6.2-1: *<ontology>* resource for ontology document access

The <*ontology*> resource above contains the child resources specified in table 7.3.6.2-1.

Table 7.3.6.2-1: Child resources of *<ontology>* resource in the unstructured approach

Child Resources of <semanticDescriptor>	Child Resource Type	Multiplicity	Description
[variable]	*<subscription>*	0..n	See [i.3], clause 9.6.8 where the type of this resource is described.

7.3.7 Semantic Mashup Function (SMF) Implementation Details

7.3.7.1 Resource Type semanticMashupJobProfile

The <*semanticMashupJobProfile*> resource represents a Semantic Mashup Job Profile (SMJP). The <*semanticMashupJobProfile*> resource shall contain the child resources specified in table 7.3.7.1-1.

Table 7.3.7.1-1: Child resources of <*semanticMashupJobProfile*> resource

Child Resources of <semanticMashupJobProfile>	Child Resource Type	Multiplicity	Description
<*variable*>	<*semanticMashupInstance*>	0..n	Represents semantic mashup instances which have been created based on this <*semanticMashupJobProfile*> resource. This child resource is optional as related<*semanticMashupJobProfile*> and <*semanticMashupInstance*> may be stored separately within the resource tree or on different CSEs.
<*variable*>	<*semanticDescriptor*>	0..1	Describes general semantic information about this <*semanticMashupJobProfile*> resource.
<*variable*>	<*subscription*>	0..n	Represents subscriptions on this resource.

The <*semanticMashupJobProfile*> resource shall contain the attributes specified in table 7.3.7.1-2.

Table 7.3.7.1-2: Attributes of <*semanticMashupJobProfile*> resource

Attributes of <semanticMashupJobProfile>	Multiplicity	RW/ RO/ WO	Description
memberFilter	1	RW	Semantically describes the types of member resources which are involved in this semantic mashup job profile <*semanticMashupJobProfile*>. When a <*semanticMashupInstance*> is created based on this <*semanticMashupJobProfile*>, the member resources of the <*semanticMashupInstance*> shall be discovered and selected based on this *memberFilter* attribute. The value of this attribute is a SPARQL query.
smilID	0..1(L)	RO	List of identifiers (e.g. URI) of related semantic mashup instance resources which have been created based on this <*semanticMashupJobProfile*>.
inputDescriptor	0..1	RW	Semantically (e.g. in semantic triples) describes the types of input parameters, which are required as input parameters in order to use this <*semanticMashupJobProfile*>. A Mashup Requestor needs to know and understand all types of input parameters as described in this attribute in order to create a <*semanticMashupInstance*> based on this <*semanticMashupJobProfile*>. Some semantic mashup job profiles may not need input parameters and as such this attribute is optional.
outputDescriptor	1	RW	Semantically (e.g. in semantic triples) describes the types of output parameters generated as semantic mashup results if using this <*semanticMashupJobProfile*>.
functionDescriptor	1	RW	Semantically (e.g. in semantic triples) describes the mashup function of this <*semanticMashupJobProfile*>. The mashup function specifies how semantic mashup results should be generated based on input parameters (defined by the *inputDescriptor* attribute) and original member resources (defined by the *memberFilter* attribute).

The structure of a <*semanticMashupJobProfile*> resource is also illustrated in figure 7.3.7.1-1.

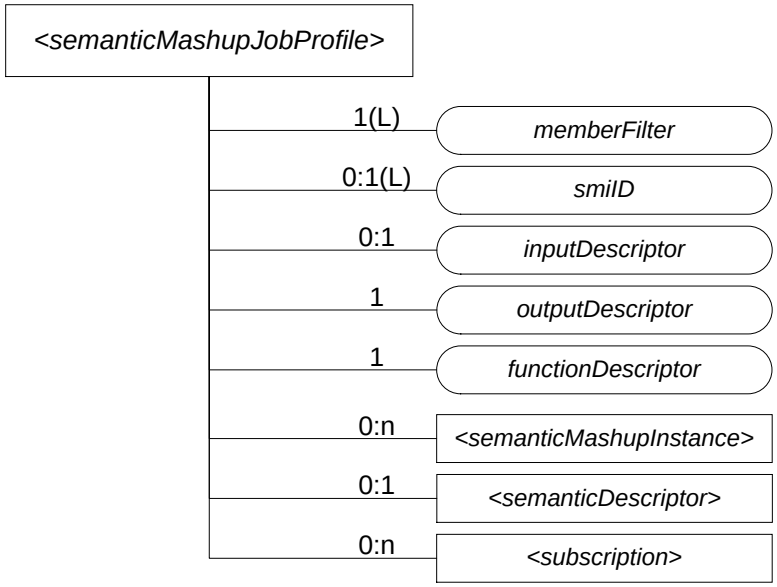

Figure 7.3.7.1-1: Structure of <*semanticMashupJobProfile*> Resource

7.3.7.2 <semanticMashupJobProfile> Procedures

7.3.7.2.0 Introduction

A <*semanticMashupJobProfile*> resource can be provisioned to a Hosting CSE which provides semantic mashup function; alternatively, an AE or CSE can request to create <*semanticMashupJobProfile*> resource at the Hosting CSE. Once a <*semanticMashupJobProfile*> resource is provisioned or created at the Hosting CSE, other oneM2M CSEs/AEs, which act as Mashup Requestors, can discover, retrieve, update, or delete it based on the requirements.

Figure 7.3.7.2-1 illustrates a generic procedure (e.g. Create/Retrieve/Update/Delete) to operate on a <*semanticMashupJobProfile*> resource. Detailed descriptions are given in following clauses 7.3.7.2.1, 7.3.7.2.2, 7.3.7.2.3, and 7.3.7.2.4, respectively.

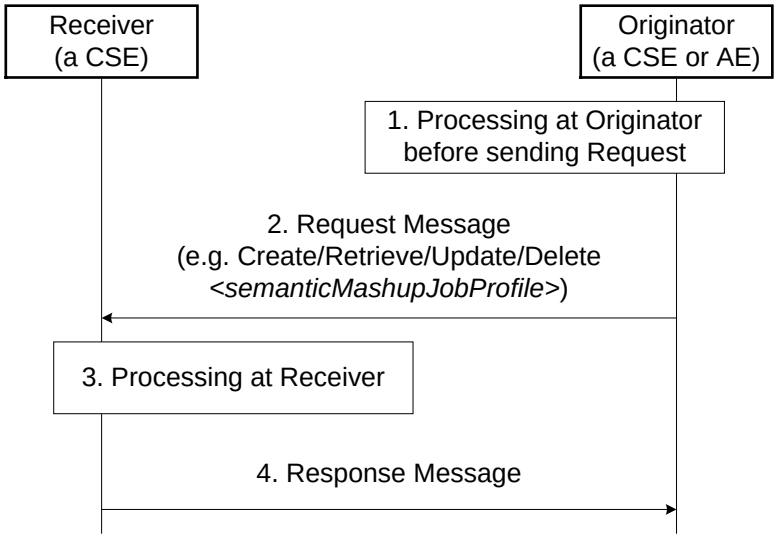

Figure 7.3.7.2-1: Generic procedures for operating a <*semanticMashupJobProfile*> resource

7.3.7.2.1 Create <semanticMashupJobProfile>

This procedure shall be used for creating a <*semanticMashupJobProfile*> resource as described in table 7.3.7.2.1-1.

Table 7.3.7.2.1-1: <*semanticMashupJobProfile*> CREATE

<*semanticMashupJobProfile*> CREATE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content**: The resource content shall provide the information about an <*semanticMashupJobProfile*> resource (e.g. attribute values) as described in the clause 7.3.7.1.
Processing at Originator before sending Request	According to clause 10.1.1.1 in [i.3].
Processing at Receiver	According to clause 10.1.1.1 in [i.3].
Information in Response message	All parameters defined in table 8.1.3-1 in [i.3] apply with the specific details for: **Content:** Address of the created <*semanticMashupJobProfile*> resource, according to clause 10.1.1.1 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.1.1 in [i.3].
Exceptions	According to clause 10.1.1.1 in [i.3]

7.3.7.2.2 Retrieve <semanticMashupJobProfile>

This procedure shall be used for retrieving the attributes of a <*semanticMashupJobProfile*> resource as described in table 7.3.7.2.2-1.

Table 7.3.7.2.2-1: <*semanticMashupJobProfile*> RETRIEVE

<*semanticMashupJobProfile*> RETRIEVE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content:** void.
Processing at Originator before sending Request	According to clause 10.1.2 in [i.3].
Processing at Receiver	The Receiver shall verify the existence (including **Filter Criteria** checking, if it is given) of the target resource or the attribute and check if the Originator has appropriate privileges to retrieve information stored in the resource/attribute. Otherwise clause 10.1.2 in [i.3] applies.
Information in Response message	All parameters defined in table 8.1.3-1 in [i.3] apply with the specific details for: **Content:** attributes of the <*semanticMashupJobProfile*> resource as defined in clause 7.3.7.1.
Processing at Originator after receiving Response	According to clause 10.1.2 in [i.3].
Exceptions	According to clause 10.1.2 in [i.3]. In addition, a timer has expired. The Receiver responds with an error.

7.3.7.2.3 Update <semanticMashupJobProfile>

This procedure as described in table 7.3.7.2.3-1 shall be used to update an existing <*semanticMashupJobProfile*> resource, e.g. an update to its *inputDescriptor* attribute. The generic update procedure is described in clause 10.1.3 in [i.3].

Table 7.3.7.2.3-1: <*semanticMashupJobProfile*> UPDATE

<*semanticMashupJobProfile*> UPDATE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content:** attributes of the <*semanticMashupJobProfile*> resource as defined in clause 7.3.7.1 to be updated.
Processing at Originator before sending Request	According to clause 10.1.3 in [i.3].
Processing at Receiver	According to clause 10.1.3 in [i.3].
Information in Response message	According to clause 10.1.3 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.3 in [i.3].
Exceptions	According to clause 10.1.3 in [i.3].

7.3.7.2.4 Delete <semanticMashupJobProfile>

This procedure as described in table 7.3.7.2.4-1 shall be used to delete an existing <*semanticMashupJobProfile*> resource. The generic delete procedure is described in clause 10.1.4.1 in [i.3].

Table 7.3.7.2.4-1: <*semanticMashupJobProfile*> DELETE

<*semanticMashupJobProfile*> DELETE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply.
Processing at Originator before sending Request	According to clause 10.1.4.1 in [i.3].
Processing at Receiver	According to clause 10.1.4.1 in [i.3]: • If the <*semanticMashupJobProfile*> to be deleted has *smilD* attribute and the *smilD* attribute has a value, the Receiver notifies each <*semanticMashupInstance*> resource as included in the *smilD* attribute of the removal of the <*semanticMashupJobProfile*> since those <*semanticMashupInstance*> resources use this <*semanticMashupJobProfile*>. • If the <*semanticMashupJobProfile*> to be deleted has <*semanticMashupInstance*> child resources, all those <*semanticMashupInstance*> child resources shall be removed accordingly.
Information in Response message	According to clause 10.1.4.1 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.4.1 in [i.3].
Exceptions	According to clause 10.1.4.1 in [i.3].

7.3.7.3 Resource Type *semanticMashupInstance*

<*semanticMashupInstance*> models and represents a Semantic Mashup Instance (SMI) resource. A CSE/AE as a Mashup Requestor can request to create <*semanticMashupInstance*> resources at another oneM2M CSE which implements the semantic mashup function as described in the clause 7.3.2. Each created <*semanticMashupInstance*> resource corresponds to a semantic mashup job profile (i.e. a <*semanticMashupJobProfile*> resource); in other words, how the <*semanticMashupInstance*> resource should execute the mashup operation to calculate the mashup result is specified in the corresponding <*semanticMashupJobProfile*> resource. Note that the <*semanticMashupInstance*> and its corresponding <*semanticMashupJobProfile*> resources may be placed at the same CSE or at different CSEs, and the *smjpID* attribute of the <*semanticMashupInstance*> allows locating the corresponding <*semanticMashupJobProfile*> resource. If the <*semanticMashupInstance*> resource has a <*semanticMashupResult*> as its child resource, the Mashup Requestor may use it to retrieve the mashup result.

<*semanticMashupInstance*> resource shall contain the child resources specified in table 7.3.7.3-1.

Table 7.3.7.3-1: Child resources of <*semanticMashupInstance*> resource

Child Resources of <semanticMashupInstance>	Child Resource Type	Multiplicity	Description
<*variable*>	<*semanticMashupResult*>	0..n	Contains mashup result. A <*semanticMashupInstance*> resource may have multiple <*semanticMashupResult*> child resources, with each mashup result instance resulting from different member resource values. The hosting CSE generates <*semanticMashupResult*> each time when it executes the mashup operation and calculate a new semantic mashup result (e.g. for long-lived mashup application as described in the clause 8.8.3.3).
<*variable*>	<*semanticDescriptor*>	0..1	Describes general semantic information about this <*semanticMashupInstance*> resource.
<*variable*>	<*subscription*>	0..n	Stands for any subscription on this <*semanticMashupInstance*>. This is an existing oneM2M resource.
<*mashup*>	<*mashup*>	0..1	This is a standard oneM2M virtual resource. When a Mashup Requestor sends a RETRIEVE operation on this virtual resource, it triggers a re-calculation and re-generation of the mashup result.

<*semanticMashupInstance*> resource shall contain the attributes specified in table 7.3.7.3-2.

Table 7.3.7.3-2: Attribute of <*semanticMashupInstance*> resource

Attributes of <semanticMashupInstance>	Multiplicity	RW/ RO/ WO	Description
smjpID	1	RW	Denotes the identifier (e.g. URI) of the semantic mashup job profile resource <*semanticMashupJobProfile*> which this <*semanticMashupInstance*> is based on.
smjpInputParameter	1	RW	Contains the value of all input parameters which are required to calculate the mashup result. Note that the types of these input parameters are specified by the *inputDescriptor* attribute of the corresponding <*semanticMashupJobProfile*> which is denoted by the *smjpID* attribute of this <*semanticMashupInstance*> resource. This attribute is not needed if the corresponding <*semanticMashupJobProfile*> does not have *inputDescriptor* attribute.
memberStoreType	1	RW	Indicates the way which member resources should be stored under this <*semanticMashupInstance*>. For example: • If *memberStoreType*="URI Only", the *mashupMember* attribute contains the URI of each member resource. • If *memberStoreType*="URI and Value", the *mashupMember* attribute contains both the URI and the value of each member resource.
mashupMember	0:1(L)	RW	Stores the URI and/or value of each mashup member resource, which is dependent on the value of *memberStoreType* attribute.
resultGenType	1(L)	RW	Describes how the mashup result should be generated using this <*semanticMashupInstance*>. Example values for this attribute could be one of the following or a combination of them: • If *resultGenType*="When SMI Is Created", the semantic mashup result is generated when this <*semanticMashupInstance*> is created by running semantic functions specified by the corresponding <*semanticMashupJobProfile*>. • If *resultGenType*="When Mashup Requestor Requests", the mashup result is to be calculated and generated when requested or triggered by a Mashup Requestor which sends a RETRIEVE operation on the virtual child resource *mashup*. • If *resultGenType*="Periodically", the CSE which hosts <*semanticMashupInstance*> calculates and generates the semantic mashup result periodically based on the *periodForResultGen* attribute. • If *resultGenType*="When A Mashup Member Is Updated", the CSE which hosts <*semanticMashupInstance*> calculates and generates the semantic mashup result whenever there is any update on the *mashupMember* attribute of <*semanticMashupInstance*>.
periodForResultGen	0:1	RW	Is the time period for re-calculating and generating the semantic mashup result. When it is the time to re-calculate the semantic mashup result, the CSE hosting this <*semanticMashupInstance*> needs to retrieve the latest content value of each member resource if it is not obtained yet. This attribute is needed when *resultGenType*="Periodically".

The structure of <*semanticMashupInstance*> resource is also illustrated in figure 7.3.7.3-1.

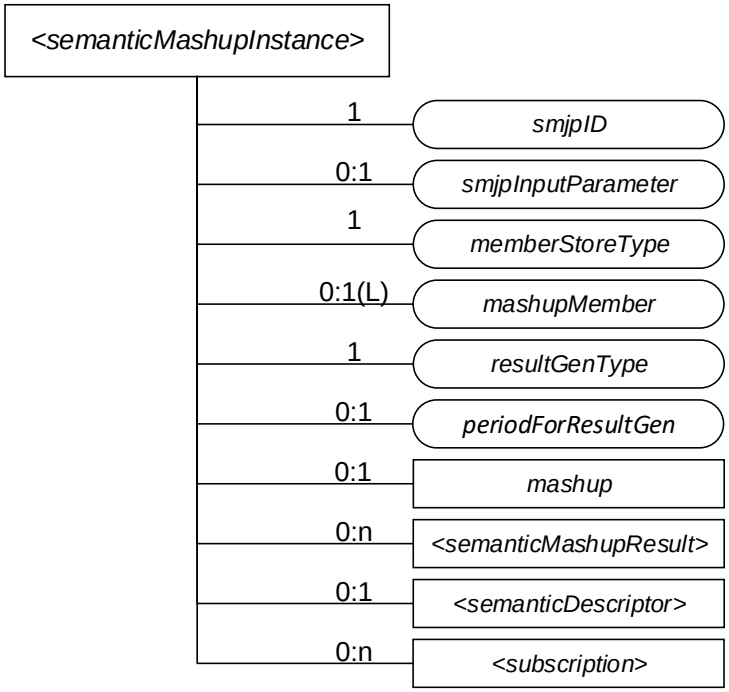

Figure 7.3.7.3-1: Structure of <*semanticMashupInstance*> Resource

7.3.7.4 <semanticMashupInstance> Procedures

7.3.7.4.0 Introduction

Figure 7.3.7.4.0-1 illustrates the procedure to operate a <*semanticMashupInstance*> resource (e.g. Create/Retrieve/Update/Delete a <*semanticMashupInstance*> resource). Detail descriptions are given in the clauses 7.3.7.4.1, 7.3.7.4.2, 7.3.7.4.3, 7.3.7.4.4 and 7.3.7.4.5, respectively.

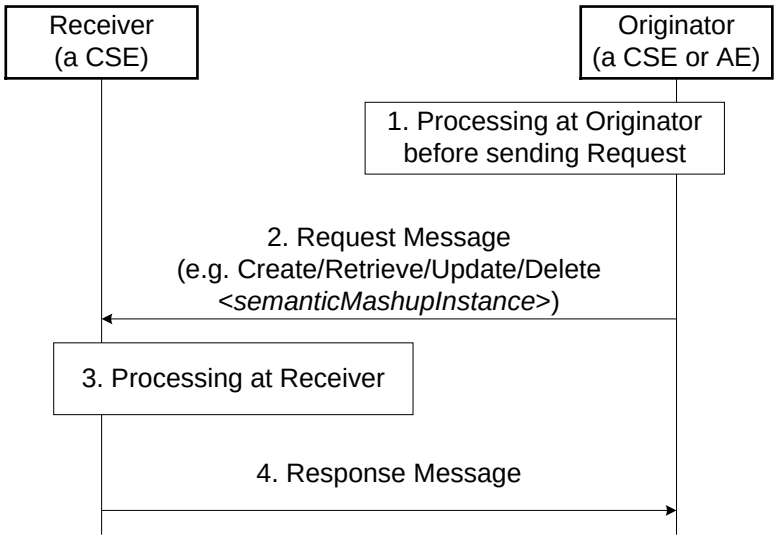

Figure 7.3.7.4.0-1: Procedures for Operating a <*semanticMashupInstance*> Resource

7.3.7.4.1 Create <semanticMashupInstance>

This procedure shall be used for creating a <*semanticMashupInstance*> resource as described in table 7.3.7.4.1-1.

Table 7.3.7.4.1-1: <*semanticMashupInstance*> CREATE

<*semanticMashupInstance*> CREATE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content**: The resource content shall provide the information about a <*semanticMashupInstance*> resource (e.g. attribute values) as described in the clause 7.3.7.3.
Processing at Originator before sending Request	According to clause 10.1.1.1 in [i.3]: • If the Originator knows the identifier or URI of each mashup member, it can include the value of *mashupMember* in the Request message.
Processing at Receiver	According to clause 10.1.1.1 in [i.3]: • The Receiver shall first check if the corresponding <*semanticMashupJobProfile*> as denoted by *smjpID* attribute exists or not. If it does not exist, the Receiver shall not create the <*semanticMashupInstance*> and shall report an error (e.g. "<*semanticMashupJobProfile*> does not exist") in the Response message to the Originator. If it exists, the Receiver shall retrieve its content. • The Receiver shall check if *smjpInputParameter* included in the Request message meets the input parameter requirement as specified by the *inputDescriptor* attribute of corresponding <*semanticMashupJobProfile*>. If it does not meet the requirement, the Receiver shall not create the <*semanticMashupInstance*> and shall report an error (e.g. "*smjpInputParameter*" does not meet the requirement") in the Response message to the Originator. • According to the *memberFilter* attribute of the retrieved <*semanticMashupJobProfile*>, the Receiver extracts the SPARQL query contained in *memberFilter*and use it to discover and determine mashup member resources for the <*semanticMashupInstance*> to be created. • Dependent on the *memberStoreType* attribute contained in the Request message, the Receiver maintains each member resource in different ways. If *memberStoreType*="URI Only", the Receiver creates the *mashupMember* attribute containing the URIs of the determined member resources. If *memberStoreType*="URI and Value", the Receiver creates the *mashupMember* attribute, retrieves the content value of each member resource and then stores both the identifier and the content value of each member resource in the *mashupMember* attribute. • Depending on the *resultGenType* attribute contained in the Request message, the Receiver prepares to execute the corresponding semantic mashup job profile as follows: – If *resultGenType*=" When SMI Is Created", the Receiver retrieves the content value of each member resource if not retrieved yet; then it executes mashup functions as specified by the <*semanticMashupJobProfile*> and generates semantic mashup result, which shall be stored in the <*semanticMashupResult*> child resource. – If *resultGenType*="When A Mashup Requestor Requests", there is no further processing at the Receiver. – If *resultGenType*="Periodically", the Receiver shall set up a timer according to the *periodForResultGen* attribute contained in the Request message. When the timer expires, the Receiver shall retrieve the content value of each member resource and re-generate the mashup result; then it renews the timer. – If *resultGenType*="When A Mashup Member Is Updated", there is no further processing at the Receiver.
Information in Response message	All parameters defined in table 8.1.3-1 in [i.3] apply with the specific details for: **Content:** Address of the created <*semanticMashupInstance*> resource and address of created <*semanticMashupResult*> resource if any, according to clause 10.1.1.1 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.1.1 in [i.3].
Exceptions	According to clause 10.1.1.1 in [i.3]

7.3.7.4.2 Retrieve <semanticMashupInstance>

This procedure shall be used for retrieving the attributes of a <*semanticMashupInstance*> resource as described in table 7.3.7.4.2-1.

Table 7.3.7.4.2-1: <*semanticMashupInstance*> RETRIEVE

<*semanticMashupInstance*> RETRIEVE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content:** void.
Processing at Originator before sending Request	According to clause 10.1.2 in [i.3].
Processing at Receiver	The Receiver shall verify the existence (including **Filter Criteria** checking, if it is given) of the target resource or the attribute and check if the Originator has appropriate privileges to retrieve information stored in the resource/attribute. Otherwise clause 10.1.2 in [i.3] applies.
Information in Response message	All parameters defined in table 8.1.3-1 in [i.3] apply with the specific details for: **Content:** attributes of the <*semanticMashupInstance*> resource as defined in the clause 7.3.7.3.
Processing at Originator after receiving Response	According to clause 10.1.2 in [i.3].
Exceptions	According to clause 10.1.2 in [i.3].

7.3.7.4.3 Retrieve <semanticMashupInstance>/<mashup>

This procedure shall be used for triggering the CSE which hosts the <*semanticMashupInstance*> to recalculate mashup results and returning the mashup result back to the requestor (e.g. an AE) of this retrieve request as described in table 7.3.7.4.3-1.

Table 7.3.7.4.3-1: <*semanticMashupInstance*>/<*mashup*> RETRIEVE

<*semanticMashupInstance*>/<*mashup*> RETRIEVE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **To:** <*semanticMashupInstance*>/<*mashup*> **Content:** void.
Processing at Originator before sending Request	According to clause 10.1.2 in [i.3].
Processing at Receiver	The Receiver shall check if the Originator has appropriate privileges. Otherwise clause 10.1.2 in [i.3] applies: • The Hosting CSE triggers the recalculation of semantic mashup result for this <*semanticMashupInstance*>. The recalculated mashup result shall be stored in a child <*semanticMashupResult*> resource.
Information in Response message	All parameters defined in table 8.1.3-1 in [i.3] apply with the specific details for: **Content:** the mashup result, if indicated in the request
Processing at Originator after receiving Response	According to clause 10.1.2 in [i.3].
Exceptions	According to clause 10.1.2 in [i.3]. In addition: a timer has expired. The Receiver responds with an error.

7.3.7.4.4 Update <semanticMashupInstance>

This procedure as described in table 7.3.7.4.4-1 shall be used to update an existing <*semanticMashupInstance*>, e.g. an update to its *memberStoreType* attribute. The generic update procedure is described in clause 10.1.3 in [i.3].

Table 7.3.7.4.4-1: <*semanticMashupInstance*> UPDATE

<*semanticMashupInstance*> UPDATE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply with the specific details for: **Content:** attributes of the <*semanticMashupInstance*> resource as defined clause 7.3.7.3 to be updated.
Processing at Originator before sending Request	According to clause 10.1.3 in [i.3].
Processing at Receiver	According to clause 10.1.3 in [i.3]: • If the updated attribute in the Request message is *smjpInputParameter* and if the original *resultGenType*="When SMI Is Updated", the Receiver needs to recalculate the semantic mashup result using the new values of input parameters. • If the updated attribute in the Request message is *memberStoreType*, the Receiver needs to change the way to maintain mashup member resources. For example, if *memberStoreType* is updated from "URI Only" to "URI and Value", the Receiver needs to retrieve the content value of each mashup member resource and store the values together with URI in *mashupMember* attribute. If *memberStoreTy*pe is updated from "URI and Value" to "URI Only", the Receiver needs *mashupMember* attribute to only maintain the identifier of each mashup member. • If the updated attribute in the Request message is *resultGenType*, the Receiver changes the way to calculate/generate the semantic mashup result accordingly.
Information in Response message	According to clause 10.1.3 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.3 in [i.3].
Exceptions	According to clause 10.1.3 in [i.3].

7.3.7.4.5 Delete <semanticMashupInstance>

This procedure as described in table 7.3.7.4.5-1 shall be used to delete an existing <*semanticMashupInstance*>. The generic delete procedure is described in clause 10.1.4.1 in [i.3].

Table 7.3.7.4.5-1: <*semanticMashupInstance*> DELETE

<*semanticMashupInstance*> DELETE	
Associated Reference Point	Mca, Mcc and Mcc'.
Information in Request message	All parameters defined in table 8.1.2-3 in [i.3] apply.
Processing at Originator before sending Request	According to clause 10.1.4.1 in [i.3].
Processing at Receiver	According to clause 10.1.4.1 in [i.3]: • In addition, The Receiver removes this <*semanticMashupInstanc*e> from the *smilD* attribute of the corresponding <*semanticMashupJobProfile*>.
Information in Response message	According to clause 10.1.4.1 in [i.3].
Processing at Originator after receiving Response	According to clause 10.1.4.1 in [i.3].
Exceptions	According to clause 10.1.4.1 in [i.3].

7.3.7.5 Examples for <semanticMashupJobProfile> and <semanticMashupInstance>

A concrete example is illustrated below to show how the proposed semantic mashup related resources can be utilized to realize the smart parking mashup application.

For this mashup application, two ontologies are illustrated in figures 7.3.7.5-1 and 7.3.7.5-2, respectively for parking spot ontology and semantic mashup job profile ontology:

- Figure 7.3.7.5-1 shows a general ontology on parking spot which can be leveraged by various different mashup applications. The following classes are defined in this ontology:
  - parkingSpot, parkingSpotInAParkingBuilding, streetParkingSpot, address, spotSelectionAlgorithm;
  - parkingSpotInAParkingBuilding is a subclass of parkingSpot;
  - streetParkingSpot is a subclass of parkingSpot;
  - spotSelectionAlgorithm is used to apply "adoptedSelectionCriteria" on the candidate parking spots and address (i.e. user destination) to calculate suitable parking spots.
- Figure 7.3.7.5-2 shows the particular ontology for this mashup application (called SMJP ontology). The following classes are defined in this ontology:
  - semanticMashupJobProfile, input, output, memberCandidate, and mashupFunction.
  - Note that, depending on different applications, those classes defined in the SMJP ontology will be linked to classes of specific ontologies when realizing different mashup applications. For example, when describing the SMJP of the smart parking assistance as shown in figure 7.3.7.5-3, the classes defined in the SMJP ontology will be linked to the specific classes defined in the parking spot ontology shown in figure 7.3.7.5-1.

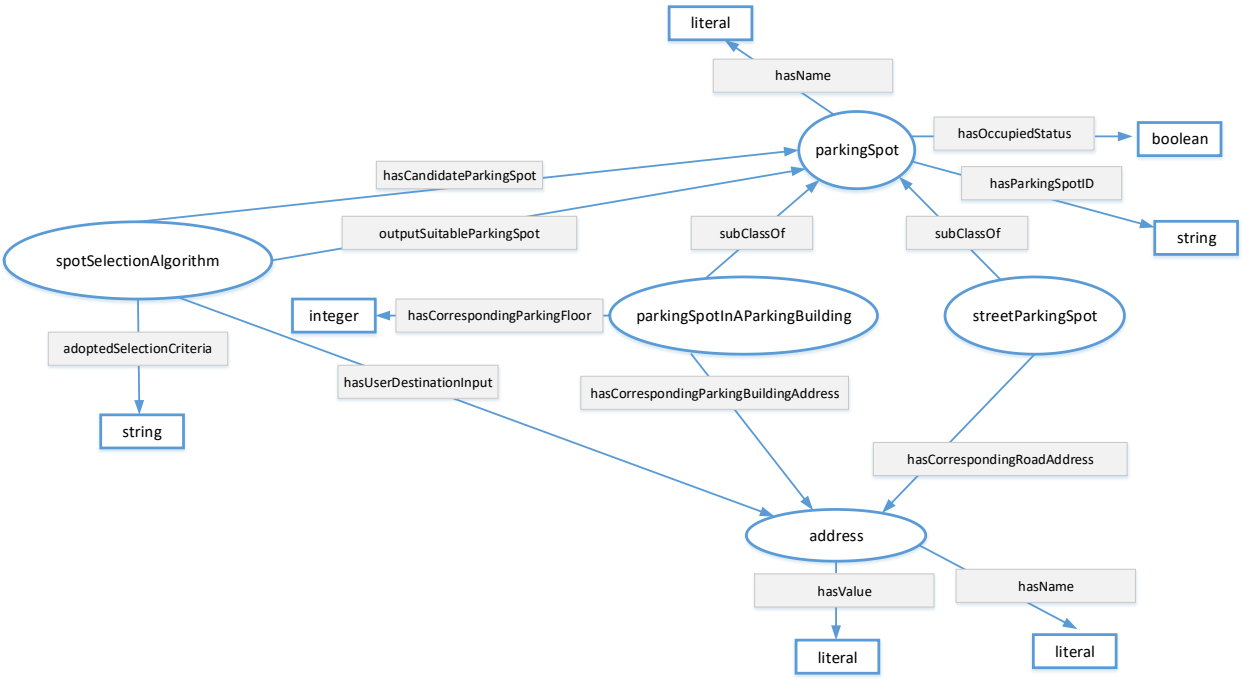

Figure 7.3.7.5-1: Parking Spot Ontology

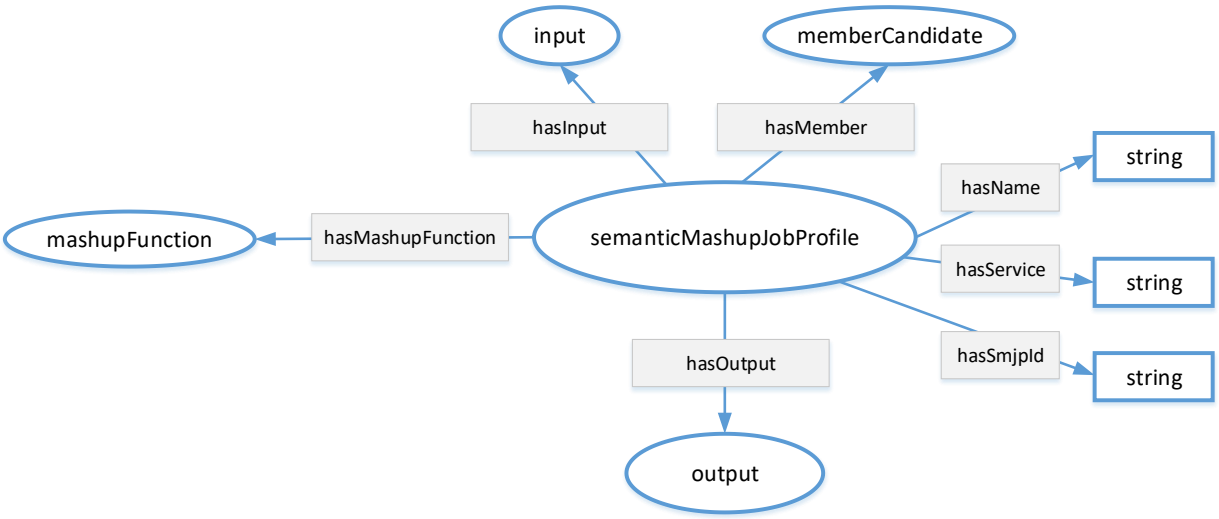

Figure 7.3.7.5-2: Semantic Mashup Job Profile (SMJP) Ontology

Based on those two ontologies, an example of <*semanticMashupJobProfile*> resource (i.e. <*SmartParkingAssistance*>) is shown in figure 7.3.7.5-3. Basically, this <*SmartParkingAssistance*> resource exposes all the necessary information about the smart parking mashup application. For example:

- *memberFilter:* This attribute includes the information about which resources are eligible or qualified to provide data inputs for this smart parking mashup application. For example, <*SmartParkingAssistance*> has two type of member candidates: the first type of member candidate is the type of "building parking spot" and the second type of member candidate is the type of "street parking spot". This is where how the classes defined in the SMJP ontology are linked to the specific classes defined in the parking spot ontology, i.e. an instance of memberCandidate class defined in SMJP ontology is also an instance of parkingSpot class defined in the parking spot ontology. In the meantime, certain constraints are also defined to describe further application requirements. For example, for a given parking spot (either a building parking spot or a street parking spot), it needs to be in an "unoccupied" status. For this example, the *memberFilter* contains a SPARQL query as below. The returned result from this SPARQL query will be a list of RDF triples; each triple represents an instance of parkingSpotInAParkingBuilding or an instance of streetParkingSpot; then the mashup function as described in *functionDescriptor* is able to know the type of each member candidate and can appropriately apply them accordingly to generate mashup result:

```
PREFIX ps:<http://parkingspot.example.org>
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#>

CONSTRUCT {
   ?BuildingParkingSpot       rdf:type             ps:parkingSpotInAParkingBuilding .
   ?StreetParkingSpot         rdf:type             ps:streetParkingSpot .
}
where
{
  ?BuildingParkingSpot  rdf:type            ps:parkingSpotInAParkingBuilding ;
                        ps:hasOccupiedStatus    "unoccupied" .

  ?StreetParkingSpot    rdf:type                ps:streetParkingSpot ;
                        ps:hasOccupiedStatus    "unoccupied" .
}
```

- *inputDescriptor:* This attribute includes what information is needed from the Mashup Requestor (MR) to leverage/apply this semantic mashup profile. For example, in the smart parking example shown in figure 7.3.7.5-3, the user/MR will provide his/her destination address (described as "smjp:input1"). Similarly, here an instance of input class defined in SMJP ontology is an instance of address class defined in the parking spot ontology. It is also possible that other inputs may also be provided such as "parking preference", which is not shown in this example.

- *functionDescriptor:* This attribute indicates how the mashup result will be generated, which is based on certain application business logic. For example, in the smart parking mashup application, given all the eligible/candidate parking spots, different types of selection criteria could be adopted. In the example shown in figure 7.3.7.5-3, it indicates that the parking spot having the shortest walking distance will be selected as the "suitable parking spot" that is to be returned to the user (as the mashup result). The real implementation of the application business logic (e.g. how to calculate the walking distance between a parking spot and the user destination? and how to find the shortest walking distance in this smart parking assistance example?) is an implementation-specific (e.g. some script code); this is an internal process of the Common Service Entity (CSE) where <*semanticMashupInstance*> resides and will not be exposed to via the <*semanticMashupInstance*> resource.

- *outputDescriptor:* This attribute describes what the mashup result looks like. For example, in the smart parking example shown in figure 7.3.7.5-3, this attribute indicates that the mashup result has the name of "suitable parking spot" and it is in fact either a building parking spot or a street parking spot. Since the mashup result is a type of ex:parkingSpotInAParkingBuilding or ex:streetParkingSpot (which are the classes/concepts defined in the adopted ontology), it will be clear that the mashup result in terms of an instance of output class defined in SMJP ontology is also an instance of parkingSpotInAParkingBuilding or streetParkingSpot classes defined in the parking spot ontology, and then the mashup result will have corresponding properties, such as address information, spot ID, etc.

- <*semanticDescriptor*>*/descriptor:* The information stored in the <*semanticDescriptor*>*/descriptor* can include the general metadata about a <*semanticMashupJobProfile*>. For example, in figure 7.3.7.5-3, it shows that the <*SmartParkingAssistance*> is type of <*semanticMashupJobProfile*> resource, and it is for supporting smart parking application and can provide a service called "find suitable parking spot".

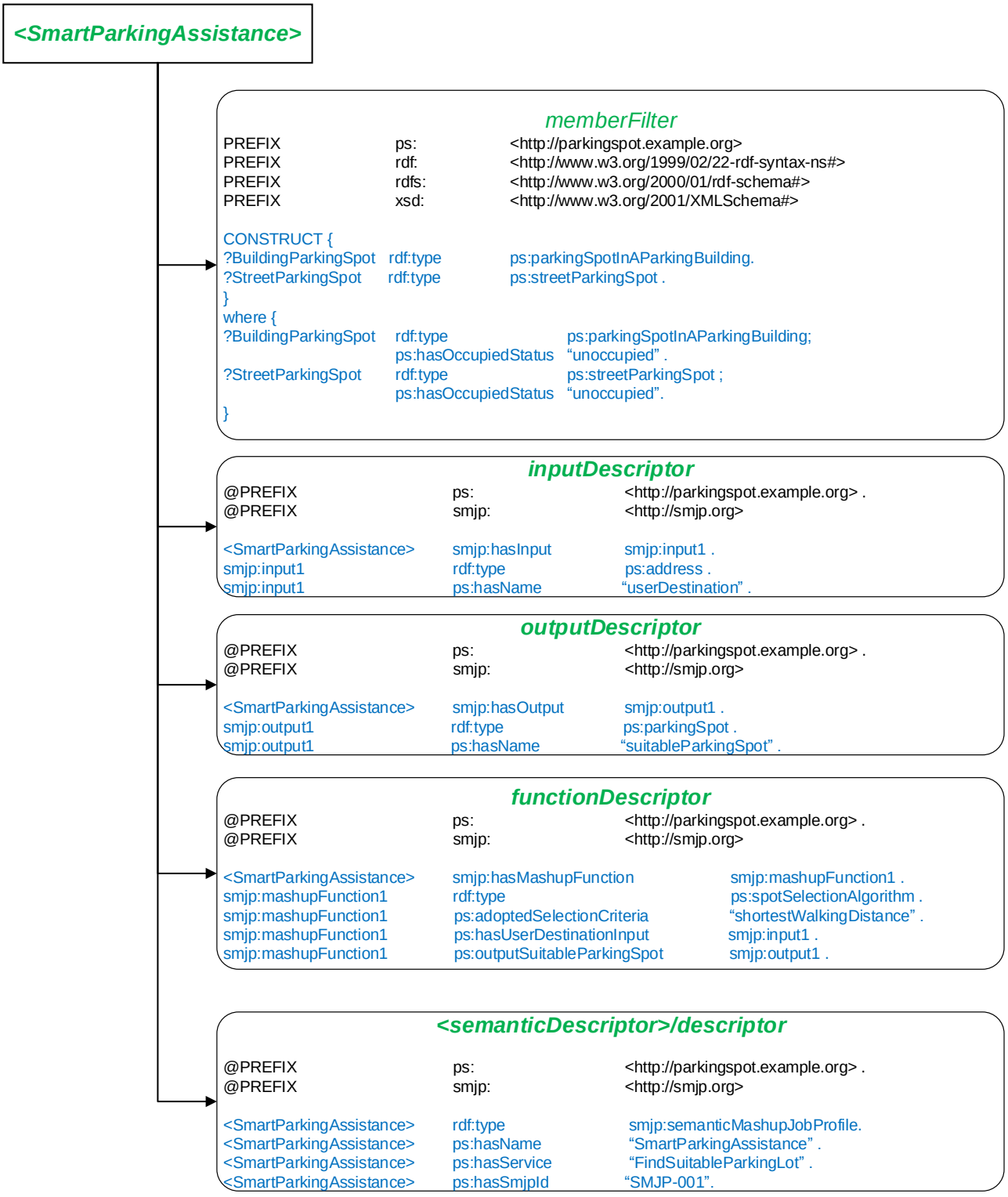

Figure 7.3.7.5-3: An SMJP of Smart Parking Mashup Application using a <*semanticMashupJobProfile*> Resource

Once an MR identifies a desired <*semanticMashupJobProfile*>, it can initialize a real mashup process (which corresponds to a "working instance", i.e. a <*semanticMashupInstance*>). In the example of smart parking mashup application, when a user intends to find a suitable parking spot around his/her destination, it will send a request to the hosting CSE to trigger a real mashup process for the smart parking assistance application. Accordingly, the hosting CSE will refer to the <*semanticMashupJobProfile*> of the smart parking assistance (as shown in figure 7.3.7.5-3), and create a <*semanticMashupInstance*> based on this <*semanticMashupJobProfile*> and the inputs given by the user. Figure 7.3.7.5-4 shows a concrete example of the <*semanticMashupInstance*> for the smart parking assistance application. In figure 7.3.76.5-4, it shows a specific <*SuitableParkingSpot-1*> resource, which is a type of <*semanticMashupInstance*> resource, and it exposes all the necessary information about this specific SMI. For example:

- *smjpID:* This attribute indicates this <*semanticMashupInstance*> is created based on which <semanticMashupJobProfile> resource. In the example shown in figure 7.3.7.5-4, it shows that the *smjpID* is "SmartParkingAssistance-001", which corresponds to smart parking assistance mashup application profile as shown in figure 7.3.7.5-3.

- ***smjpInputParameter:*** This attribute includes the inputs from the MR/user side. For example, it shows that the user is looking for a suitable parking spot around his/her destination, which is "255 36th street, New York City, NY, 10001".

- ***memberStoreType:*** The value of this attribute is "URI and Value", which means that the *mashupMember* attribute will show both the URIs of member resources and their attribute values in RDF triples in this example.

- ***mashupMember:*** This attribute indicates all the qualified member resources for this particular <*semanticMashupInstance*>. For example, as shown in figure 7.3.7.5-4, given user's destination address (i.e. around "255 36th street, New York City, NY, 10001"), all the potential eligible parking spots will be listed here, which were identified based on the "*memberFilter*" attribute as defined in the corresponding SMJP of this <*semanticMashupInstance*>. For example, two parking spots are listed in figure 7.3.7.5-4, i.e. <*buildingParkingSopt-1*> (which is a spot in a parking building) and <*streetParkingSopt-1*> (which is a street parking spot). In particular, the detailed information about those two parking spots are also included in this attribute. For example, the detailed location information about <*buildingParkingSopt-1*> is as follows: This parking spot is in a parking building at "255 37th street, New York City, NY, 10001", and this spot is in the 3rd floor, and the specific spot ID is "23".

- ***resultGenType:*** This attribute indicates how the mashup process will be triggered. In this smart parking assistance example, the value is "when SMI is created". It means that when a user is looking for a suitable parking spot by sending a request to the hosting CSE for SMS, once a corresponding SMI is created, it will immediately generate the mashup result in terms of a suitable parking spot. Note that, the smart parking assistance is type of "short-lived semantic mashup" in the sense that each user will create a respective SMI for his/her own parking needs and the created SMI may be just used once. For another type of mashup application, i.e. "long-lived semantic mashup application" such as the weather reporting mashup application, a given SMI could be re-used and shared by different MRs for multiple times. For example, once an SMI has been created for reporting weather information of New York City Area, the mashup result of this SMI can either be refreshed periodically due to real-time weather changes or the mashup process can be triggered by users' requests after the SMI is created.

- <*semanticMashupResult*>*:* To store the mashup result. For example, in figure 7.3.7.5-4, from this <*semanticMashupResult*> child resource, it can be seen that the suitable parking sport is <*buildingParkingSopt-1*>, among all the eligible parking spots near the user's destination.

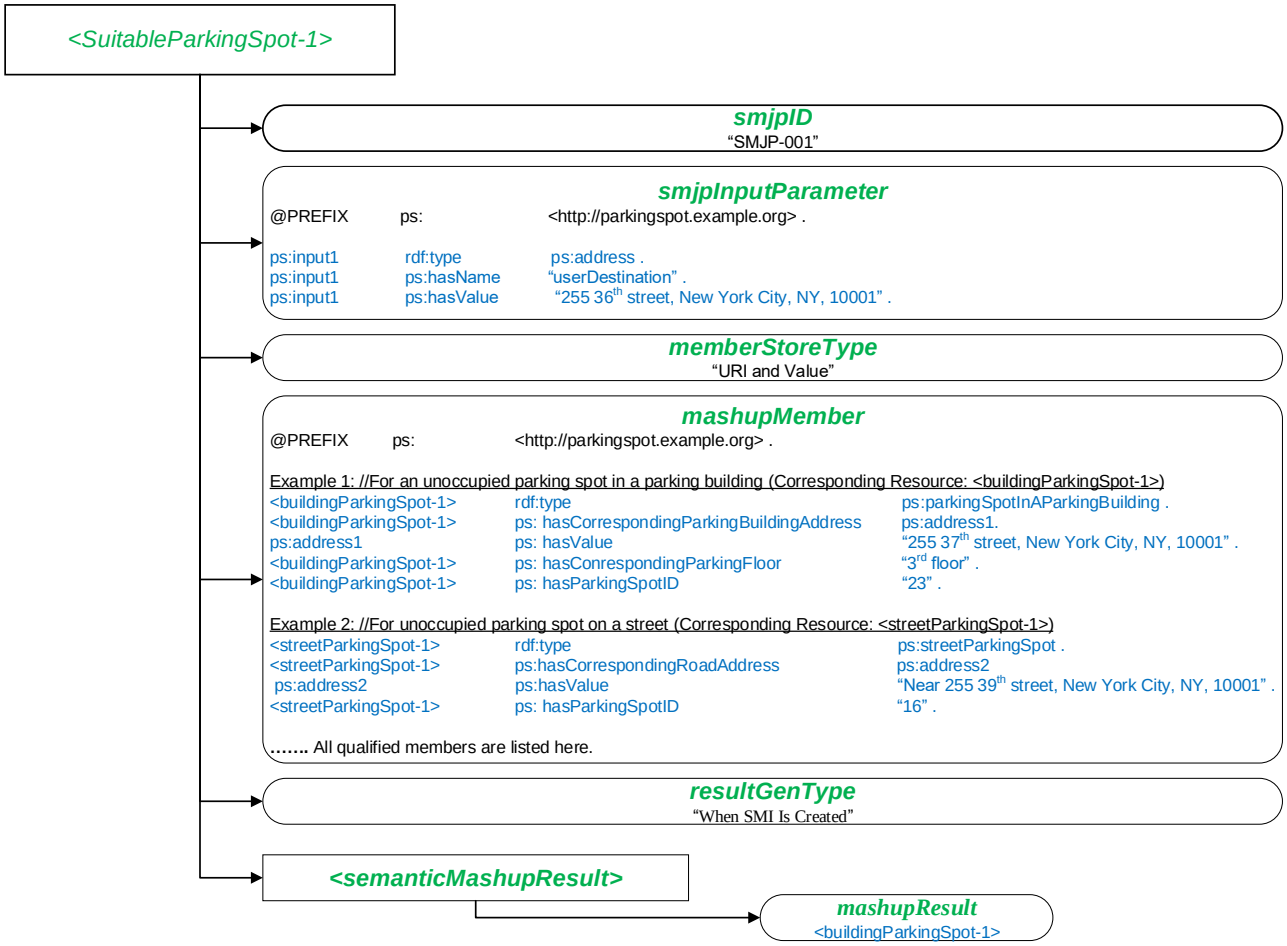

Figure 7.3.7.5-4: An SMI of Smart Parking Mashup Application Represented by a <*semanticMashupInstance*> Resource

7.4 Implementation options

7.4.1 Introduction

The efficiency of queries, semantic resource discovery and access control for sematic information depend on the underlying implementation. In this clause architectural and implementation perspectives are presented.

7.4.2 The hierarchically layered architecture

7.4.2.0 Introduction

While the current oneM2M architectural view is exclusively based on resources, these can be implemented in different ways. Especially for managing semantic triples, triple stores are the straightforward choice. This has implications for how the access control to resources and triples is handled. In this clause different implementation options are discussed assuming a hierarchical structuring into a semantic layer and a data layer.

As shown in figures 7.4.2.1-1and 7.4.2.2-1, data resources and semantic triples may be integrated in a hierarchically layered architecture. In the hierarchical model the Upper Layer provides an interface for RESTful operations, controls and manages the Access Control Policies (ACPs). The Lower Layer supports the upper layer with specific services.

From a functional perspective there are two layers: aData Layer containing the data resources and related functions and a Semantic Layer containing semantic triples and related semantic functions. Each functional Layer may be used as either the Upper Layer or the Lower Layer as detailed later in this clause.

The layers may reside on different CSEs, but integration on the same CSE may be more performance efficient.

7.4.2.1 Data layer as upper layer

In figure 7.4.2.1-1 the Upper Layer is the Data Layer and the Lower Layer is the Semantic Layer which provides the Upper Layer with semantic information.

There are three alternatives for implementation of the Semantic Layer as the Lower Layer:

- Using Semantic Descriptors distributed in the oneM2M Resource Tree, which corresponds to the Release 2 data resource driven approach.
- Using permanent Sematic Graph Stores, which may be distributed or centralized.
- In a hybrid option, using temporary Semantic Graph Stores together with the distributed resource model to provide support for semantic functionality e.g. in Release 2.

Figure 7.4.2.1-1 shows a typical data resource driven approach for M2M scenarios. Resource Discovery through the data resource tree may be supported by the semantic leaves (i.e. distributed graph stores) in the Semantic Layer. The ACPs are maintained under <*semanticDescriptor*> resources in the Data Layer. Further description for the Access Control methods in this architecture are provided in clause 7.5.

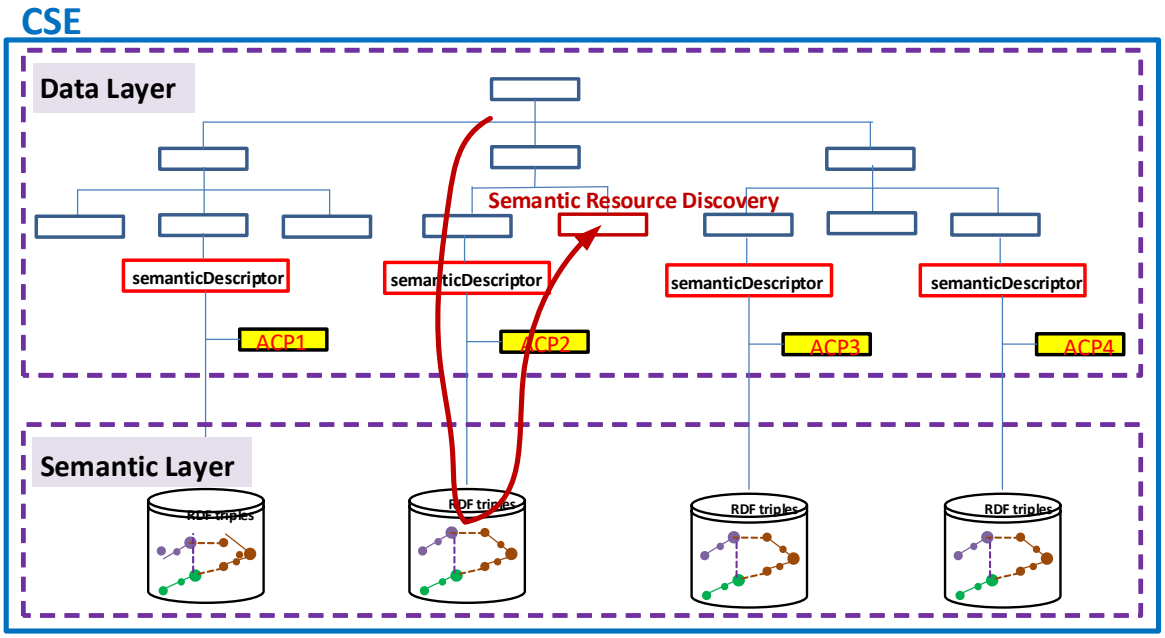

Figure 7.4.2.1-1: Hierarchically layered structure - controlled by the data layer

7.4.2.2 Semantic layer as upper layer

In figure 7.4.2.2-1, the Upper Layer is the Semantic Layer and the Lower Layer is the Data Layer supporting the upper layer with raw data. The main implementation of the Sematic Layer as an Upper Layer is using Sematic Graph Stores. Further description for the Access Control methods in this architecture are provided in clause 7.5.

NOTE: In the following clauses the terms "Semantic Graph Stores" and "Triple Stores" are used interchangeably.

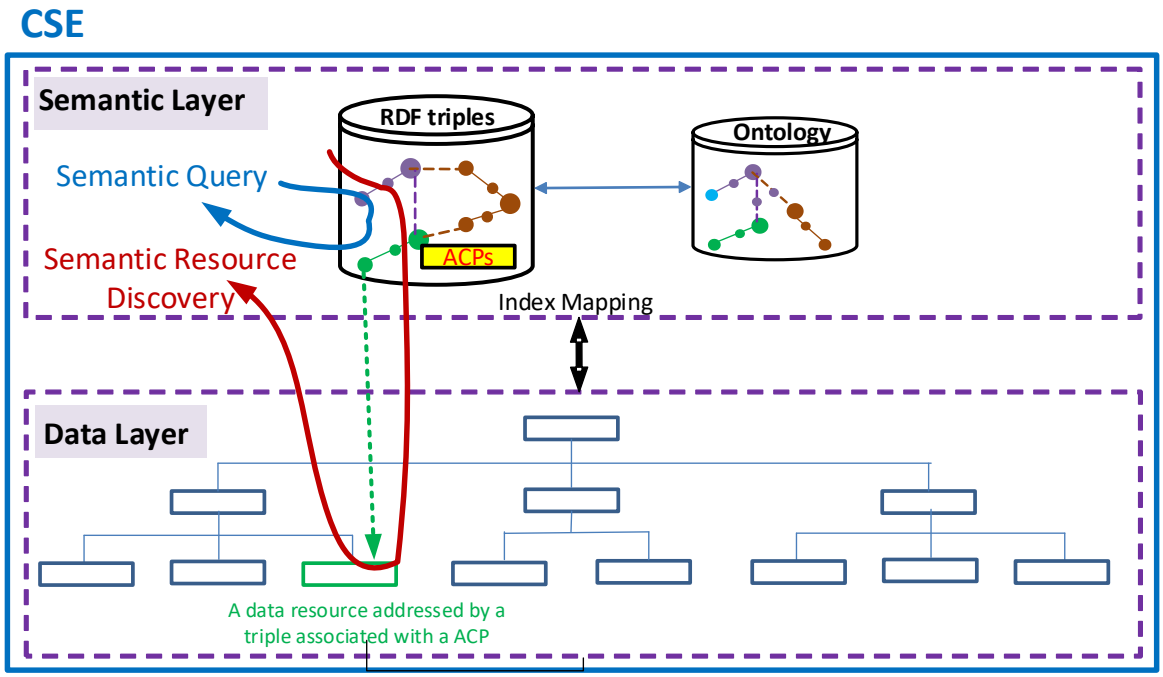

Figure 7.4.2.2-1: Hierarchically layered structure - controlled by the semantic layer

Figure 7.4.2.2-1 shows a typical semantics driven scheme for Semantic Web scenarios. Semantic Query may be conducted in the Semantic Layer with the return of the URI or URL of the data resources in the Data Layer. Semantic Resource Discovery may also be realized with the return of the data resources in the Data Layer via proper mapping between these two layers. The triples in the Semantic Layer are associated with their specific ACPs. A data resource in the Data Layer is addressed by a triple (e.g. via its URI or URL) associated with the ACP.

7.4.3 The parallel architecture

Looking beyond the currently supported semantic functionality, a more advanced architecture may be needed. In the following clause the concepts of Data Entity and Semantic Entity are introduced. These concepts may be used in different configurations, supporting more advanced semantic functionality like semantic mash-up and the possible interaction with other semantic platforms like the semantic web.

Figure 7.4.3-1 shows a parallel architecture with distinct entities, i.e. a Data Entity and a Semantic Entity.

A Data Entity and/or a Semantic Entity may each have its own Access Control Policies (ACPs) for managing the access control within its scope.

The semantic triples or data resources in the Data Entity may be exposed to the Semantic Entity with specific ACPs associated. For example:

a) a Semantic Publication function in the Data Entity may expose semantic triples to the Central Graph Store in Semantic Entity with the corresponding ACPs associated with the triples; or

b) a Data Annotation function in the Data Entity may expose data and associated ACPs to the Central Graph Store via the local temporary or caching Relational Data Base Management System (RDBMS) and Semantic Reasoning and Mapping functions in Semantic Entity.

The data resources or semantic triples in the Semantic Entity may also be exposed to the Data Entity with specific ACPs associated. For example:

c) a Semantic Mash-up function in the Semantic Entity may expose new data resources and related ACPs from the semantic mash-up to the Data Entity; or

d) a Semantic Annotation function in the Semantic Entity may expose semantic triples and related ACPs to the <*semanticDescriptor*> resources in the Data Entity.

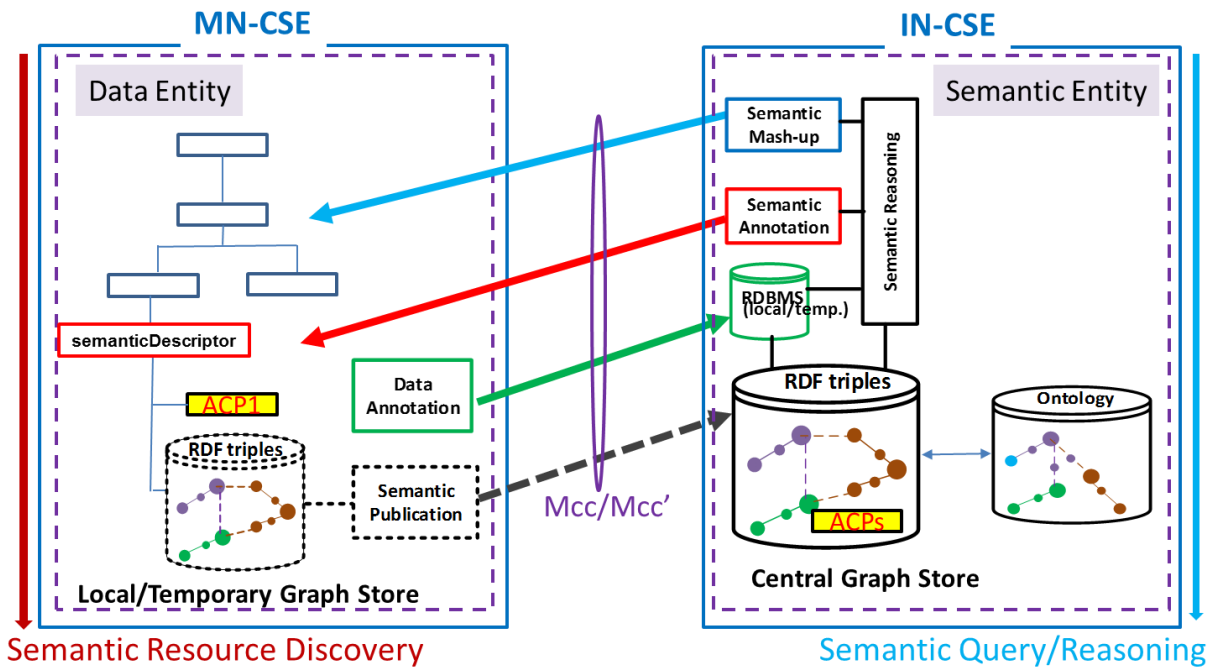

Figure 7.4.3-1: Access control in a parallel structure

Figure 7.4.3-1 shows that a Data Entity and a Semantic Entity may reside on different CSEs. But a Data Entity and a Semantic Entity may also reside on the same CSE. Figure 7.4.3-2 shows a logical resource tree with both a Data Entity and a Semantic Entity.

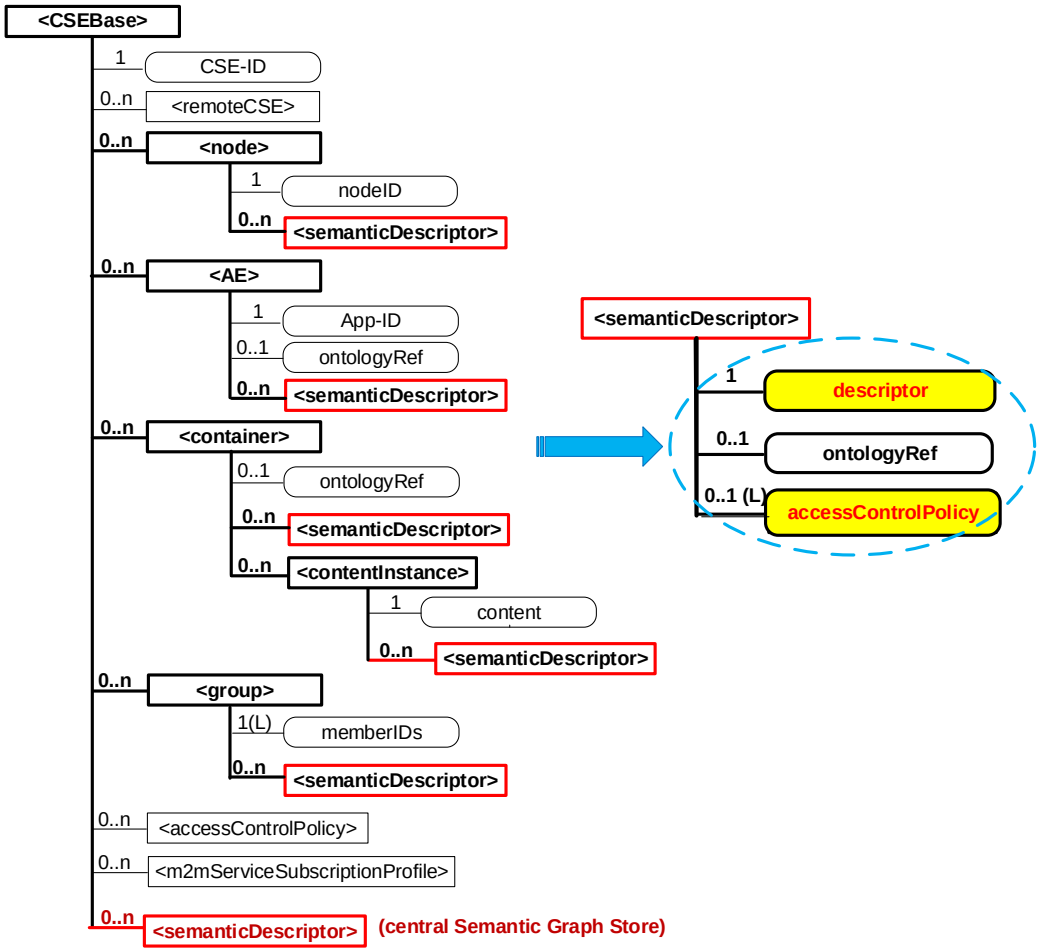

Figure 7.4.3-2: Logic resource tree with both a data entity and a semantic entity

7.5 Approaches to Access Control

7.5.1 Access Control for semantic discovery, based on the ACPs in resource tree and triples in the graph store

7.5.1.1 Introduction

Implementing access control for semantic related functionalities (e.g. semantic resource discovery, semantic query) based on the ACPs in resource tree can be seen as an attractive approach since the compatibility with existing resource tree ACPs will be kept.

7.5.1.2 Solution A: Semantic filtering based on graph store

In the solution descriptions below, some assumptions are considered.

1) There is a centralized graph store to store the triples in all <semanticDescriptor>s.

2) Based on the query request with a target URL in the resource tree, the scope of the query is limited to the triples in both the <semanticDescriptor>s as child resources in the sub-tree under the target URL and the relevant <semanticDescriptor>s linked to the <semanticDescriptor>s under the target URL.

In solution A, the triples in the <semanticDescriptor>s will be stored in one graph of a graph store. To retain the effect of ACPs in resource tree, the <semanticDescriptor>is used as the anchor to link the ACPs in resource tree and the access control during the query on semantic repository. The procedure of the solution A are described as follows.

Pre-steps before semantic query process:

1) The CSE hosting the Graph store creates an internal ontology with class *SemanticDescriptor* and *atomDescription,* and the property *describedIn*, *hasSubject* *hasObject* and *hasProperty.*

2) For each <semanticDescriptor>with ACP in resource tree(s), the CSE hosting the graph store creates corresponding semantic descriptor instances in the semantic graph store using IRI/URL of the respective <semantic Descriptor>. The semantic descriptor instances are the instances of the predefined class *SemanticDescriptor*.

3) The CSE hosting the graph store adds triples in semantic graph store to associate the semantic triples in the <semanticDescriptor>s in resource tree with the created semantic descriptor instances. The triples in the <semanticDescriptor>s in the resource tree(s) of other CSEs should be notified to the CSE hosting the graph store.

Considering that one subject can be described in multiple <semanticDescriptor>s with different ACPs, the association should be implemented with each triple for classification, and the association triples are added based on each triple described in the <semanticDescriptor>. Figure 7.5.1.2-1 shows the association between the triple and the semantic descriptor instance.

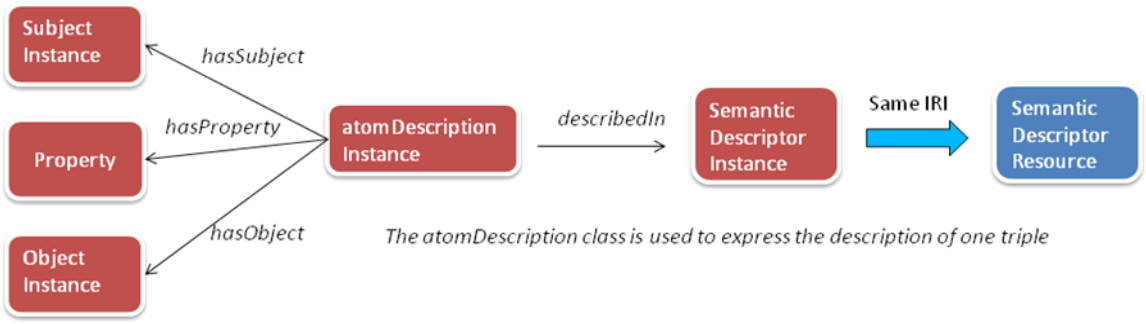

Figure 7.5.1.2-1: Association between the triple and the SemanticDescriptor instance

For example, for the triple in <semanticDescriptor> A as *classX property Y classZ,* the following association triples are needed to be added. These association triples are termed "SD Relationship Triples":

- *atomDescriptionA hasSubject classX*
- *atomDescriptionA hasObject classX*
- *atomDescriptionA hasProperty classZ*
- *atomDescriptionA describedIn SemanticDescriptorA*

The process after receiving the semantic query request with SPARQL statement:

1) The receiver CSE finds the <semanticDescriptor>s where the Originator (AE ID or CSE ID) is allowed to use for querying based on the ACPs in resource tree and the target URL in the request.

2) The receiver CSE identifies the corresponding *SemanticDescriptor* instances (same IRI/URL with the <semanticDescriptor>) in the semantic graph store.

3) In the received original SPARQL semantic query statements, the receiver CSE adds new sentences to indicate that the target variable triples are associated with the identified *SemanticDescriptor* (i.e. new SD Relationship Triples) instances as follows:

   a) find the variables and their relevant triples in the SPARQL query;

   b) create *atomdescription* variables for each triple with variables in the query;

   c) associate the *atomdescription* variables with each triple with variables in the query;

   d) add the sentence to associate the *atomdescription* variables and the identified *SemanticDescriptor* instances.

Figure 7.5.1.2-2 shows the association between the triple with variables and the *SemanticDescriptor* instances.

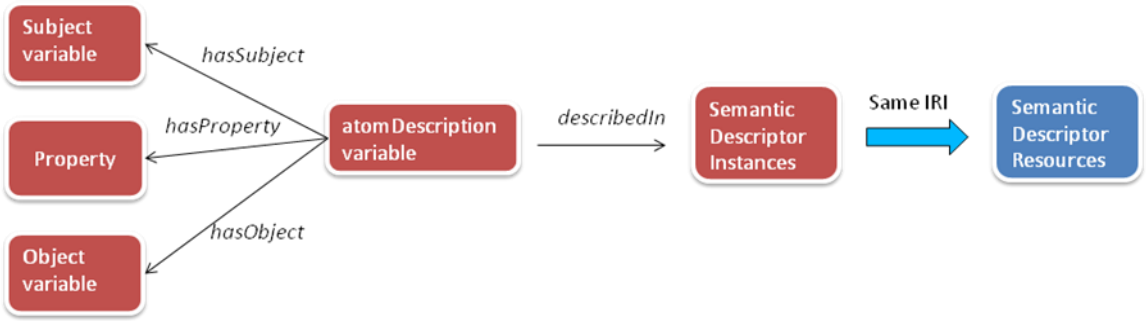

Figure 7.5.1.2-2: Association between the triple with variables in SPARQL query and the SemanticDescriptor instances

For example, for the original SPARQL query:

```
SELECT ?device ?operation
WHERE {
?device rdf:type m2m:WashMachine.
?device m2m:hasOperation ?operation.
}
```

If the allowed *SemanticDescriptor* instance is *SemanticDescriptorA*, then the modified SPARQL query is given as:

```
 SELECT ?device ?operation
WHERE {
?device rdf:type m2m:WashMachine.
?device m2m:hasOperation ?operation.
?atom1 temp:hasSubject ?device.
?atom1 temp:hasObject ?operation
?atom1 temp:hasProperty m2m:hasOperation
?atom2 temp:hasSubject ?device
?atom2 temp:hasObject m2m:WashMachine
?atom2 temp:hasProperty rdf:type
?atom1  temp:desribedIn SemanticDescriptorA(IRI/URL).
?atom2  temp:desribedIn SemanticDescriptorA(IRI/URL).
}
```

1) The receiver CSE sends the modified SPARQL semantic query statement to the CSE hosting the graph store for querying the graph store.

2) The receiver CSE compose Response according to the semantic query results feedback from the CSE hosting the graph store.

In the following part, we give a complete example to explain solution A. The considered resource tree is shown in figure 7.5.1.2-3 where three <semanticDescriptor> (SD) resources are involved.

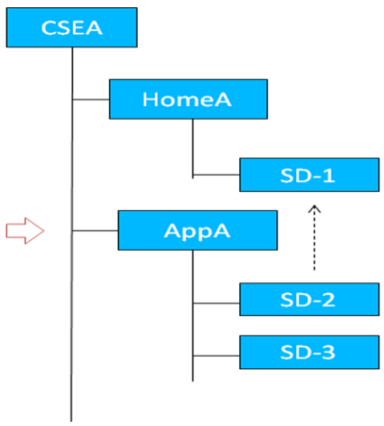

Figure 7.5.1.2-3: Considered resource tree in the example

The triples in SD-1, SD-2 and SD-3 are assumed as follows:

- SD-1:
  - HomeA rdf:type ex:Home
  - HomeA ex:hasLocation LocationA
  - LocationA ex:hasLatitude "300"
  - LocationA ex:hasLongitude "200"
- SD-2:
  - DeviceA rdf:type m2m:TemperatureSensor
  - DeviceA ex:hasLocation LocationA

There is a link to SD-1 in SD-2 that indicates Location(class) related information can be retrieved from SD-1.

- SD-3
  - DeviceB rdf;type ex:DoorLock
  - DerviceB ex:hasLocation LocationA

The CSEA hosts the graph store. The triples in the graph store will include all the triples in the<semanticDescriptor>, and there will be 4 additional triples (i.e. SD Relationship Triples) for each triple in the<semanticDescriptor>. The total additional triples are given as follows:

- For SD-1:
  - atom1 temp:hasSubject HomeA
  - atom1 temp:hasObject ex:Home
  - atom1 temp:hasProperty rdf:type
  - atom1 temp:describedIn SD-1
  - atom2 temp:hasSubject HomeA
  - atom2 temp:hasObject LocationA
  - atom2 temp:hasProperty ex:hasLocation
  - atom2 temp:describedIn SD-1
  - atom3 temp:hasSubject LocationA
  - atom3 temp:hasObject "300"
  - atom3 temp:hasProperty ex:Latitude
  - atom3 temp:describedIn SD-1
  - atom4 temp:hasSubject LocationA
  - atom4 temp:hasObject "200"
  - atom4 temp:hasProperty ex:hasLongtitude
  - atom4 temp:describedIn SD-1
- For SD-2:
  - atom5 temp:hasSubject DeviceA

- atom5 temp:hasObject m2m:TemperatureSensor
- atom5 temp:hasProperty rdf:type
- atom5 temp:describedIn SD-2
- atom6 temp:hasSubject DeviceA
- atom6 temp:hasObject LocationA
- atom6 temp:hasProperty ex:hasLocation
- atom6 temp:describedIn SD-2

- For SD-3:
  - atom7 temp:hasSubject DeviceB
  - atom7 temp:hasObject m2m:DoorLock
  - atom7 temp:hasProperty rdf:type
  - atom7 temp:describedIn SD-3
  - atom8 temp:hasSubject DeviceB
  - atom8 temp:hasObject LocationA
  - atom8 temp:hasProperty ex:hasLocation
  - atom8 temp:describedIn SD-3

It is assumed that an originator sends the query request in which the target URL is the URL of AppA and SPARQL query filter is:

```
SELECT ?device
WHERE {
?device ex:hasLocation ?Location.
?Location ex:hasLatitude ?val1.
?Location ex:hasLagitude ?val2.
FILTER(?val1=="300" && ?val2 =="200")
}
```

When receiving this query request, the CSEA will first identify the scope of the <semanticDescriptor>s related to the query as follows.

1) Find that there are two SDs, i.e. SD-2 and SD-3, under the target URL, and then check the ACPs linked to these SDs and find that only SD-2 is allowed to be used by the originator in the query.

2) Find that there is a link (e.g. relatedSemantics attribute of <semanticDescriptor>) in the SD-2 to SD-1 for the class Location, and then check the original SPARQL query and find that class Location is involved.

3) Check the ACP linked to SD-1 and find that SD-1 is not allowed to be used by the originator in the query.

After identifying the scope of the <semanticDescriptor>s (SD_2) related to the query, the CSEA revises the original SPARQL query as:

```
SELECT ?device
WHERE {
?device ex:hasLocation ?Location.
?Location ex:hasLatitude ?val1.
?Location ex:hasLagitude ?val2.
?agtom1 temp:hasSubject ?device.
?atom1 temp:hasObject ?Location.
?atom1 temp:hasProperty ?hasLocation.
?atom2 temp:hasSubject ?Location.
?atom2 temp:hasObject ?val1.
?atom2 temp:hasProperty ex:hasLatitude.
?atom3 temp:hasSubject ?Location.
?atom3 temp:hasObject ?val2.
?atom3 temp:hasProperty ex:hasLongitude
```

```
?atom1 temp:describedIn SD-2
?atom2 temp:describedIn SD-2
?atom3 temp:describedIn SD-2
FILTER(?val1=="300" && ?val2 =="200")
}
```

The CSEA applies the revised SPARQL query to the graph store, and return the result.

The returned result is *no device*.

7.5.1.3 Solution B: Graph division based semantic filtering

In the solution descriptions below, some assumptions are considered.

1) There is a centralized graph store to store the triples in all <semanticDescriptor>s.

2) The returned information for query request is in the scope of the triples in all possible <semanticDescriptor>s. The scope may contain more triples (i.e. from <semanticDescriptor>s not explicitely linked to the <semanticDescriptor>s under the original target URL) compared with solution A.

In solution B, the triples in the <semanticDescriptor>s will be stored in separated graphs in the graph store. There are two options for the graph division:

- Option 1:
  - Store the triples in the same <semanticDescriptor> in a graph.
- Option 2:
  - Store the triples in the semantic descriptors linked to the same ACP in a graph.

For Option 1, there will be many graphs in the graph store and the query speed will be slow when the query across the union of a lot of graphs, but each graph of Option 1 is not necessary to be updated with the update of ACPs. For Option 2, the number of graphs will be small but it needs the synchronization between the graphs and the ACPs.

The procedure of solution B is simple:

Pre-step:

1) The CSE hosting the graph store stores the triples in the separated graphs of graph store as Option 1 or Option 2.

2) The ACP information related to <semanticDescriptor>s in the resource tree(s) of other CSEs should be notified to the CSE hosting the graph store.

Query-step

The query request will be forwarded from the receiver CSE to the CSE hosting the graph store for query:

For Option 1:

- After receiving the forwarded request, the CSE hosting the graph store will:
  1) Find the <semanticDescriptor> which is allowed to be used in the query according to the ACP.
  2) Identify the graphs that correspond to the found <semanticDescriptor> in the previous step.
  3) Apply the SPARQL query on the union of the identified graphs.
  4) Return the query result.

For Option 2:

- After receiving the forwarded request, the CSE hosting the graph store will:

    1) Identify the access permissions of the Originator of the original request before forwarding - according to ACPs, and find the relevant ACPs that includes the Originator to have discovery permissions.

    2) Identify the graphs that corresponds to the found ACPs in the previous step.

    3) Apply the SPARQL query on the union of the identified graphs.

    4) Return the query results.

7.5.2 Direct access control of semantic graph store

7.5.2.1 Introduction

As described in clauses 7.5.1.2 and 7.5.1.3, <accessControlPolicy> specified by the accessControlPolicyIDs attribute of <semanticDescriptor> may be used for access control in the Semantic Graph Store when executing SPARQL operations as a part of semantic queries. One approach is to implement access control policies directly in the Semantic Graph Store, which makes it more efficient and scalable to control the access to a centralized Semantic Graph Store. This approach may contain the following main steps with the following assumptions:

1) There is a centralized Semantic Graph Store.

2) The proposed approach will be used for semantic query over this centralized Semantic Graph Store.

3) Each semantic query matches all semanticDescriptors but not cross multiple semanticDescriptors.

4) There is a need for synchronization of access control policies between the CSE and the Semantic Graph Store:

    - Construct *Access Control Rules* specified by <*accessControlPolicy*> in the Semantic Graph Store. Note that <*accessControlPolicy*> is specified by the accessControlPolicyIDs attribute of <semanticDescriptor> resource.

    - Associate targeted *Semantic Triples* (i.e. RDF triples as described by the descriptor attribute of <semanticDescriptor> but stored in the Semantic Graph Store) with their a*ccessControlPolicyIDs* or <*accessControlPolicy*> with related *Access Control Rules*.

    - Semantic triple operations are conducted with the selected semantic triples which are associated with the *Access Control Rules* allowing the Originator to operate.

Figure 7.5.2.1-1 gives an example of access control policy for two <semanticDescriptor> resources, where there are two access control policies (i.e. <accessControlPolicy1> and <accessControlPolicy2>). The access to <semanticDescriptor1> is controlled by <accessControlPolicy1> and <accessControlPolicy2>, while the access to <semanticDescriptor2> is only controlled by <accessControlPolicy2>.

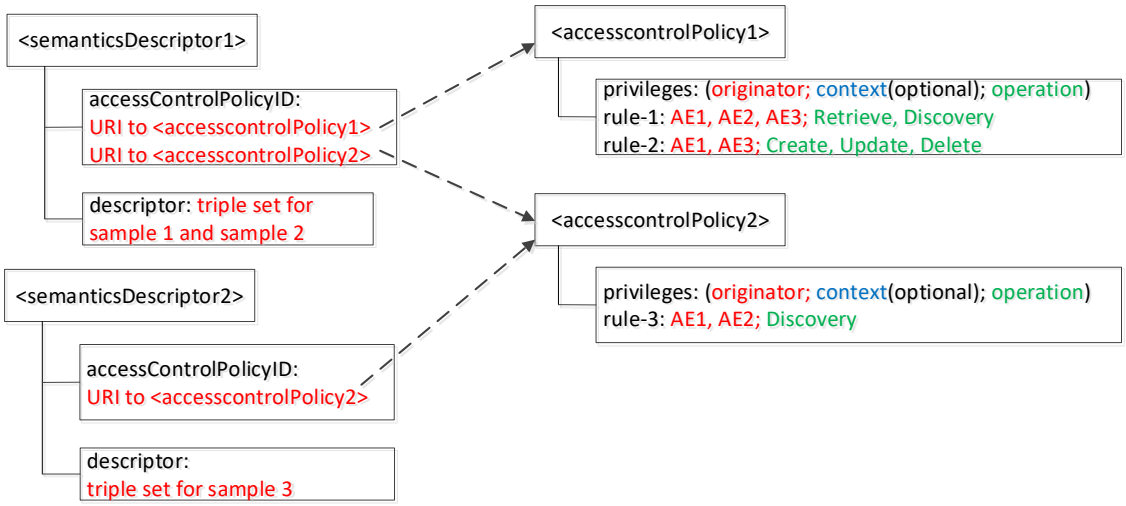

Figure 7.5.2.1-1: Example of access control policy for <semanticDescriptor>

7.5.2.2 Access control modelling in semantic graph store

Examples for access control modelling in the Semantic Graph Store are shown in figure 7.5.2.2-1, figure 7.5.2.2-2, and figure 7.5.2.2-3.

The **Access Control Ontology** shown in figure 7.5.2.2-1 defines two new classes: accessControlPolicy and accessControlRule. In addition, five new properties (i.e. hasACPRule, hasACOriginator, hasACOperations, hasACContexts and appliedTo) are defined. hasACPRule is used to link an accessControlPolicy instance with an accessControlRule instance. Properties hasACOriginator, hasACOperations and hasACContexts (optional) basically describe an accessControlRule instance and are used to specify who can issue what operations under which conditions. As these triples describe the ACP themselves, they are termed "ACP Triples".

Property appliedTo is used to describe which <semanticDescriptor> resource an accessControlPolicy instance can be applied to As these triples bind <accessControlPolicy> and <semanticDescriptor> they are temed "ACP-SD Binding Triples".

Note that this ontology is defined by following how oneM2M <accessControlPolicy> resource is specified in oneM2M TS-0001 [i.3], where an access-control-rule-tuple consists of parameters such as accessControlOriginators, accessControlOperations, and accessControlContexts.

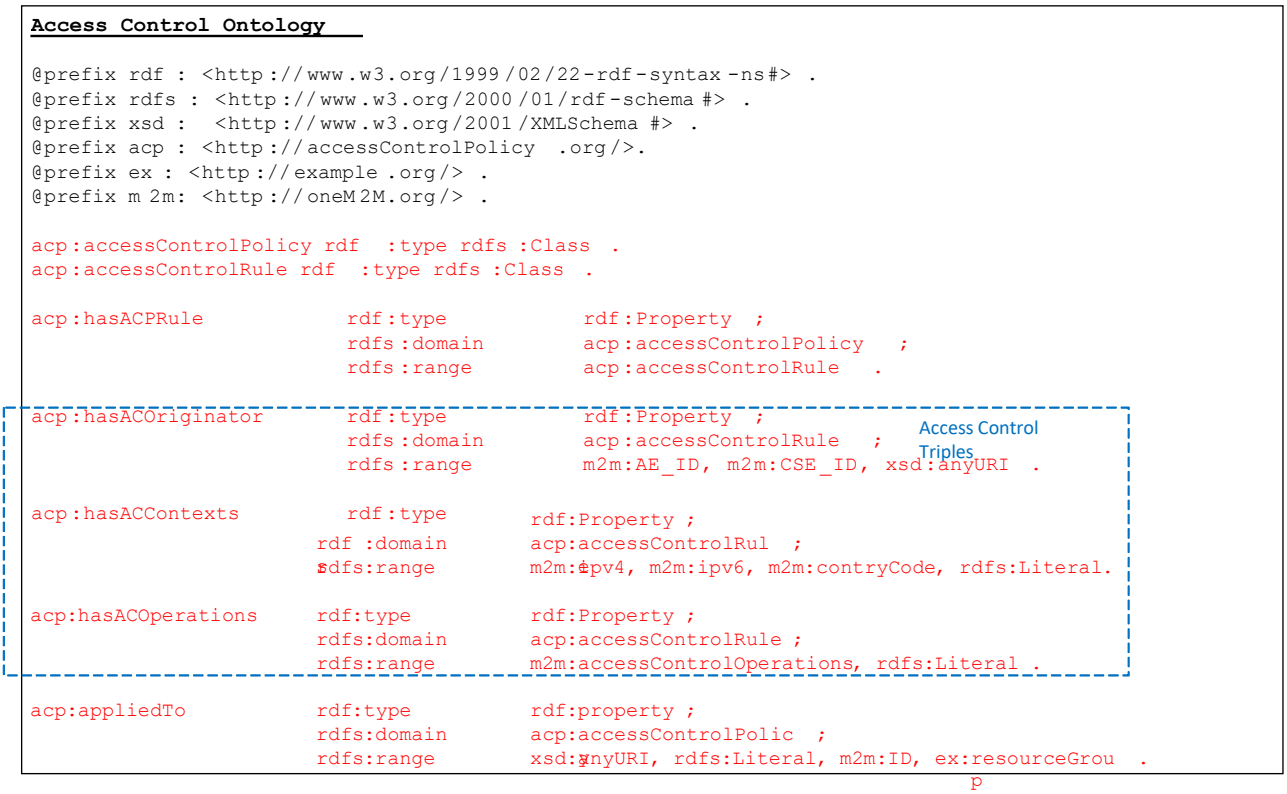

Figure 7.5.2.2-1: Access control ontology model

Figure 7.5.2.2-2 shows an example of the eHealth Ontology Reference Model, which will be used to develop the Semantic Graph Store example in figure 7.5.2.2-3 and SPARQL example in figure 7.5.2.3-1.

eHealthcare Ontology Reference Model

```
@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#> .
@prefix ex: <http://example.org/> .
@prefix acp: <http://accessControlPolicy.org/>.

ex:Person         a rdfs:Class .

ex:dateOfBirth    a rdf:Property ; rdfs:domain ex:Person ; rdfs:range xsd:date ; rdfs:comment "Date of Birth" .

ex:name           a rdf:Property ; rdfs:domain ex:Person ; rdfs:range rdfs:Literal ; rdfs:comment "name of the person" .

ex:Patient  rdfs:subClassOf ex:Person .
ex:Doctor   rdfs:subClassOf ex:Person .

ex:takeCareOf     a rdf:Property ; rdfs:domain ex:Doctor ; rdfs:range rdfs:Patient ;
                  rdfs:comment "doctor take care of (relation) patient" .

ex:MeasurementSample a rdfs:Class .

ex:measureOn      a rdf:Property ;  rdfs:domain ex:MeasurementSample ; rdfs:range xsd:date ;
                  rdfs:comment "the date of measurement" .

ex:measureFor     a rdf:Property ;  rdfs:domain ex:MeasurementSample ; rdfs:range ex:Patient ;
                  rdfs:comment "sample is measure for which patient" .

ex:unit           a rdf:Property ;  rdfs:domain ex:MeasurementSample ; rdfs:range rdfs:Literal ;
                  rdfs:comment "unit of the value" .

ex:BPMeasurementSample rdfs:subClassOf ex:MeasurementSample .

ex:dValue         a rdf:Property ;  rdfs:domain ex:BPMeasurementSample ; rdfs:range xsd:integer ;
                  rdfs:comment "value of the diastolic" .

ex:sValue         a rdf:Property ;  rdfs:domain ex:BPMeasurementSample ; rdfs:range xsd:integer ;
                  rdfs:comment "value of the systolic" .

ex:resourceGroup  a rdf:Class ;
                  rdfs:comment "contain a list of resources in resource tree" .

ex:containMeasurement   a rdf:Property ; rdfs:domain ex:resourceGroup ; rdfs:range ex:MeasurementSample ;
                        rdfs:comment "resourceGroup contains one or more measurement samples" .
```

Figure 7.5.2.2-2: eHealth ontology reference model

Figure 7.5.2.2-3 describes an example of RDF triples in the Semantic Graph Store based on the example shown in figure 7.5.2.1-1 and the Access Control Ontology defined in figure 7.5.2.2-1. In this use case, there are two patients Jack and Alice; their doctors are John and Steve, respectively. There are three blood pressure meansurement samples (i.e. Sample1 for Jack, Sample2 and Sample3 for another patient3). Corresponding triples are shown in black text in figure 7.5.2.2-3, which are generated based on the eHealth Ontology Reference Model in figure 7.5.2.2-2.

The triples in red text in figure 7.5.2.2-3 are added for access control purpose according to the proposed Access Control Ontology model in figure 7.5.2.2-1, when new ACPs are created or updated. In this example, it is assumed two access control polices be created. First, two <semanticDescriptor> are described (i.e. semanticDescriptor1 contains Sample1 and Sample2, while semanticDescriptor2 contains Sample3. Then, two access control policies are defined (i.e. accessControlPolicy1 is applied to semanticDescriptor1, while accessControlPolicy2 is applied to both semanticDescriptor1 and semanticDescriptor2). Next, the detail Access Control Rules for accessControlPolicy1 and accessControlPolicy2 are described:

- AccessControlPolicy1 has two accessControlRules, which states that:

  1) AE-ID-1, AE-ID-2, and AE-ID-3 can RETRIEVE and DISCOVER triples in the semanticDescriptor which accessControlPolicy1 is applied to (i.e. semanticDescriptor1);

  2) AE-ID-1 and AE-ID-3 can CREATE, UPDATE, or DELETE triples in the semanticDescriptor which accessControlPolicy1 is applied to (i.e. semanticDescriptor1).

- For accessControlPolicy2, only one accessControlRule is defined; this accessControlRule states that AE-ID-1 and AE-ID-2 can DISCOVER triples in the semanticDescriptor which accessControlPolicy2 is applied to (i.e. semanticDescriptor1 and semanticDescriptor2).

eHealth Semantic Graph Store

```
@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#> .
@prefix ex: <http://example.org/> .
@prefix acp: <http://accessControlPolicy.org/>.

ex:Patient1 a ex:Patient ; ex:name "Jack" ; ex:dateOfBirth  "2000-08-03"^^xsd:date .
ex:Patient2 a ex:Patient ; ex:name "Alice" ; ex:dateOfBirth  "1998-06-03"^^xsd:date .

ex:Doctor1 a ex:Doctor ; ex:name "John" ; ex:dateOfBirth  "1944-08-21"^^xsd:date ; ex:takeCareOf
ex:Patient1 .
ex:Doctor2 a ex:Doctor ; ex:name "Steve" ; ex:dateOfBirth  "1947-02-11"^^xsd:date ; ex:takeCareOf
ex:Patient2 .

ex:Sample1 a ex:BPMeasurementSample ;
      ex:measureOn "2014-08-21"^^xsd:date ; ex:measureFor ex:Patient1 ;
      ex:unit "mmHg" ; ex:sValue "150"^^xsd:integer ; ex:dValue "100"^^xsd:integer .
ex:Sample2 a ex:BPMeasurementSample ;
      ex:measureOn "2014-07-24"^^xsd:date ; ex:measureFor ex:Patient3 ;
      ex:unit "mmHg" ; ex:sValue "140"^^xsd:integer ; ex:dValue "96"^^xsd:integer .
ex:Sample3 a ex:BPMeasurementSample ;
      ex:measureOn "2012-11-24"^^xsd:date ; ex:measureFor ex:Patient3 ;
      ex:unit "mmHg" ; ex:sValue "130"^^xsd:integer ; ex:dValue "57"^^xsd:integer .

## Below are triples associating measurement samples with corresponding access control policy
ex:semanticsDescriptor1 a ex:resourceGroup ; ex:containMeasurement ex:Sample1, ex:Sample2 .
ex:semanticsDescriptor2 a ex:resourceGroup ; ex:containMeasurement ex:Sample3 .

acp:accessControlPolicy1 acp:appliedTo ex:semanticsDescriptor1 .
acp:accessControlPolicy2 acp:appliedTo ex:semanticsDescriptor1 .
acp:accessControlPolicy2 acp:appliedTo ex:semanticsDescriptor2 .

## Below are triples created for access control policy 1 resource based on access control ontology
acp:accessControlRule1_1 rdf:type acp:accessControlRule.
acp:accessControlRule1_2 rdf:type acp:accessControlRule.
acp:accessControlPolicy1 rdf:type acp:accessControlPolicy.
acp:accessControlPolicy1 acp:hasACPRule acp:accessControlRule1_1, acp:accessControlRule1_2 .

acp:accessControlRule1_1 acp:hasACOriginator "AE-ID-1", "AE-ID-2", "AE-ID-3" .
acp:accessControlRule1_1 acp:hasACOperations "RETRIEVE", "DISCOVERY" .

acp:accessControlRule1_2 acp:hasACOriginator "AE-ID-1", "AE-ID-3" .
acp:accessControlRule1_2 acp:hasACOperations "CREATE", "UPDATE", "DELETE" .

## Below are triples created for access control policy 2 resource based on access control ontology
acp:accessControlRule2_1 rdf:type acp:accessControlRule.
acp:accessControlPolicy2 rdf:type acp:accessControlPolicy.
acp:accessControlPolicy2 acp:hasACPRule acp:accessControlRule2_1 .

acp:accessControlRule2_1 acp:hasACOriginator "AE-ID-1", "AE-ID-2" .
acp:accessControlRule2_1 acp:hasACOperations "DISCOVERY" .
```

Figure 7.5.2.2-3: eHealth triples in semantic graph store

7.5.2.3 Examples of SPARQL query procedure

When the CSE receives SPARQL query from Originator, it will add the access control related patterns according to the ID of the Originator and the request operation of the query into the received SPARQL statement, and use the revised SPARQL statement to make query on the semantic graph store.

For example, in the scenario of the example in figure 7.5.2.2-3, when AE-ID-3 sends the following SPARQL query request to the CSE:

```
select distinct ?sample ?sValue ?dValue
where
{
   ?sample rdf:type ex:BPMeasurementSample .
   ?sample ex:sValue ?sValue .
   ?sample ex:dValue ?dValue .
}
```

The CSE will add some access control related statements according to the ID (i.e. AE-ID-3) of the Originator and the request operation (i.e. DISCOVERY) of the query, the revised SPARQL query can be given as:

```
select distinct ?sample ?sValue ?dValue
where
{
   ?accessControlRule acp:hasACOriginator "AE-ID-3" .
   ?accessControlRule acp:hasACOperations "DISCOVERY" .
   ?accessControlPolicy acp:hasACPRule ?accessControlRule .
   ?accessControlPolicy acp:appliedTo ?semanticDescriptor .
   ?semanticDescriptor ex:containResource ?sample .
?sample rdf:type ex:BPMeasurementSample .
   ?sample ex:sValue ?sValue .
   ?sample ex:dValue ?dValue .
}
```

Alternatively, using the approach which is described in clause 7.5.1 for the association between the semantic descriptor and semantic triples, the revised SPARQL query can be given as:

```
select distinct ?sample ?sValue ?dValue
where
{
   ?accessControlRule acp:hasACOriginator "AE-ID-3" .
   ?accessControlRule acp:hasACOperations "DISCOVERY" .
   ?accessControlPolicy acp:hasACPRule ?accessControlRule .
   ?accessControlPolicy acp:appliedTo ?semanticDescriptor .
   ?atom1 temp:describedIn ?semanticDescriptor.
 ?atom1 temp:hasSubject ?sample.
 ?atom1 temp:hasObject ?sValue.
 ?atom1 temp:hasProperty ex:sValue.
 ?atom2 temp:describedIn  ?semanticDescriptor.
 ?atom2 temp:hasSubject ?sample.
 ?atom2 temp:hasObject ?dValue.
 ?atom2 temp:hasProperty ex:dValue.
 ?atom3 temp:describedIn  ?semanticDescriptor.
 ?atom3 temp:hasSubject ?sample.
 ?atom3 temp:hasObject ex:BPMeasurementSample.
 ?atom3 temp:hasProperty rdf:type.
}
```

Figure 7.5.2.3-1 shows the SPARQL query result in the above example over the eHealth Semantic Graph Store in figure 7.5.2.2-3. According to the access control triples added to the Semantic Graph Store (i.e. red text in figure 7.5.2.2-3), AE-ID-3 is only allowed to DISCOVER samples included in semanticDescriptor1 (i.e. Sample1 and Sample2). As a result, the returned result for SPARQL query in figure 7.5.2.3-1 presents the selected content of Sample1 and Sample2.

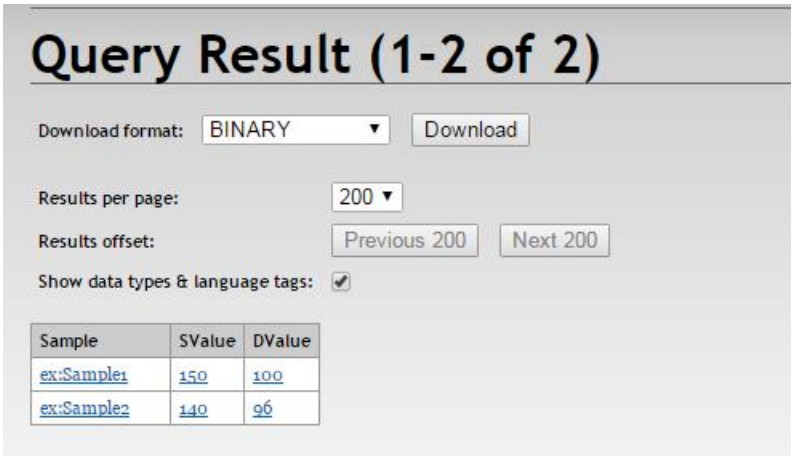

Figure 7.5.2.3-1: Example for eHealth semantic query result with access control

7.5.3 Access control using temporary semantic graph stores

This clause describes an implementation approach that can be used for implementing the semantic features supported in Release 2 of oneM2M. <semanticDescriptor> resources have been introduced to be able to semantically annotate oneM2M resources of certain resource types, including AE, Container, ContentInstance, FlexContainer and more. Semantic functionalities have been added for filtering and selectively updating the semantic information stored in the descriptor attribute of the <semanticDescriptor> which is represented in the form of RDF triples. The SPARQL language has been selected as the most suitable for specifying the filter and selective update operations.

For implementing the SPARQL-based semantic functionality, a SPARQL-engine is needed and such engines are typically provided on top of semantic graph stores. Thus, it is a straight-forward implementation choice to use such a semantic graph store.

<semanticDescriptor> resources as any other oneM2M resources have associated access policies which determine whether a requester is allowed to access the content of the resource. These access policies also apply for the semantic description stored in a <semanticDescriptor> resource and have to be adhered to when executing SPARQL requests on and across semantic information contained in these <semanticDescriptor> resources. So even when storing the semantic information in semantic graph stores and accessing them using SPARQL requests, the access control policies have to be applied. Approaches for translating and applying access control policies within the semantic graph store are described in clause 7.5.2.

In this clause, we show how the SPARQL-based functionality needed for oneM2M Release 2 can be implemented using temporary semantic graph stores. For an incoming SPARQL request, the access control policies are applied when accessing the relevant <semanticDescriptor> resource(s) for populating the temporary semantic graph store. Once this has happened, the SPARQL query can be executed without further access control checks.

Figure 7.5.3-1 shows the different steps in the case of a single <semanticDescriptor> resource being accessed to enable the execution of a SPARQL request on its semantic content.

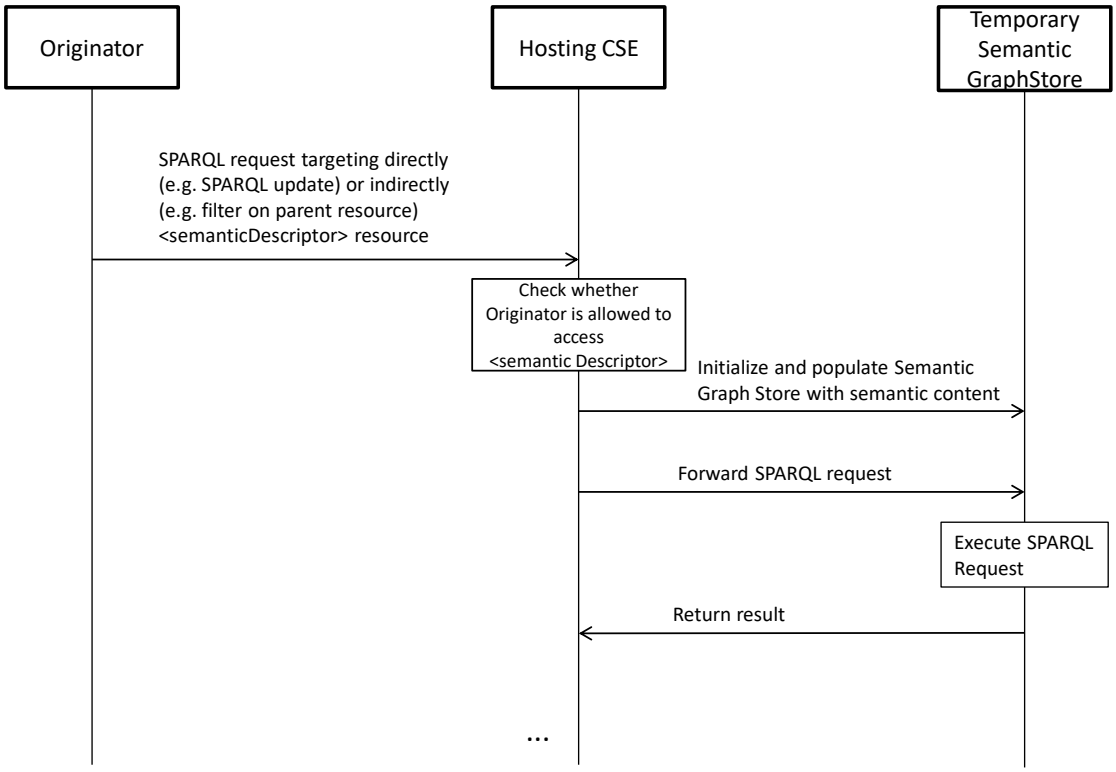

Figure 7.5.3-1: Execute SPARQL request on single <semanticDescriptor>

When receiving the request with the SPARQL content, the Hosting CSE checks the access control policy applying to the <semanticDescriptor> resource. If the originator of the request is allowed to access it, it retrieves the semantic information from its *descriptor* attribute, initializes the Temporary Semantic Graph Store, and populates it with the semantic information. Then it forwards the SPARQL request to the Temporary Semantic Graph Store to be executed. Note that the Temporary Semantic Graph Store in this context is seen as an implementation component and not an architectural component according to the oneM2M architecture.

Figure 7.5.3-2 shows the case that, in addition to the targeted <semanticDescriptor> resource, a set of related <semanticDescriptor> resources needs to be included before executing the SPARQL request. This is described in oneM2M TS-0034 [i.12] Caluse 7.4.3 and oneM2M TS-0001 [i.3], clause 10.2.35.2.2. The relevant <semanticDescriptor> resources are identified through the *relatedSemantics* attribute.

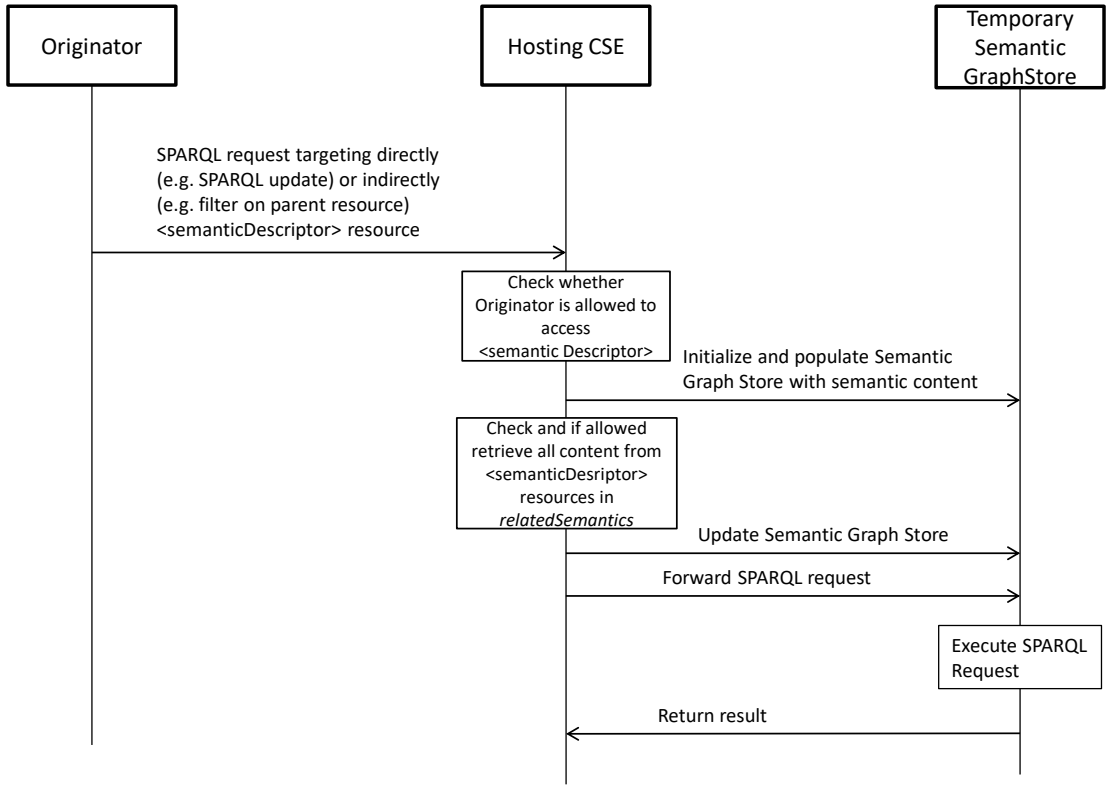

Figure 7.5.3-2: Execute SPARQL request on <semanticDescriptor> plus those identified through *relatedSemantics* attribute

In addition to the steps shown in figure 7.5.3-1, the Hosting CSE attempts to retrieve the semantic content of the <semanticDescriptor> resources identified by the *relatedSemantics* attribute and updates the Temporary Semantic Graph Store accordingly. The respective access control policies are checked as part of the retrieval as in any other resource access. Only once all the identified (and accessible) content has been added is complete, the SPARQL request is forwarded for execution.

Figure 7.5.3-3 shows the case that the semantic content of the targeted <semanticDescriptor> resource contains links to further <semanticDescriptor> resoruces identified by *resourceDescriptorLink* properties. This annotation-based approach is described in oneM2M TS-0034 [i.12] clause 7.4.2 and in oneM2M TS-0001 [i.3], clause 10.2.35.2.1.

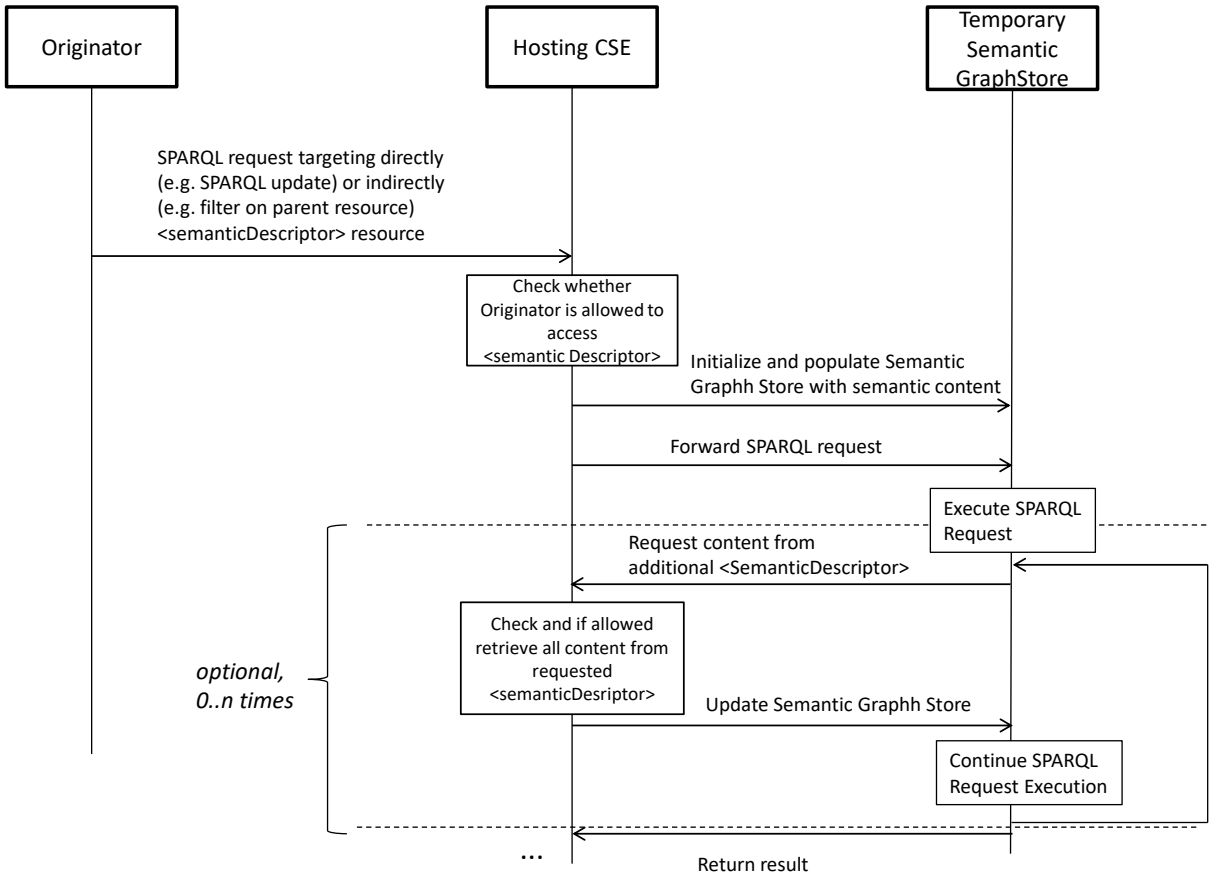

Figure 7.5.3-3: Execute SPARQL request on <semanticDescriptor> plus those encountered during the execution in form of *resourceDescriptorLink* properties

In this case, the modified SPARQL engine of the Temporary Semantic Graph Store may encounter elements with *resourceDescriptorLink* properties when matching variables as part of the execution of the SPARQL request. In this case the execution will be halted and the engine requests the content of the <semanticDescriptor> resource identified by the *resourceDescriptorLink* property before continuing execution on the merged content. This can happen 0 to n times during the execution. Note that the Temporary Semantic Graph Store is not an architectural oneM2M component and logically is to be seen as part of the Hosting CSE. The semantic content is requested on behalf of the original Originator with its access rights. The advantage of the implementation with the Temporary Semantic Graph Store is that no special care has to be taken with respect to access control policies. Access control policies are evaluated in the usual way when accessing the semantic content on which the SPARQL request is to be executed. The disadvantage is of course that a Temporary Semantic Graph store has to be initialized and populated each time a SPARQL request is to be executed.

8 Semantic functionalities

8.1 Introduction

The clauses 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 8.10 and 8.11introduce the semantic functionalities that a oneM2M system requires to address the identified semantics-related requirements. Semantic annotations are at the basis for all semantic functionalities. For correct operation, the validation of semantic information is important and it is required to have the respective ontologies according to which the semantic information is modelled. Thus the oneM2M system needs to be able to manage ontologies. Semantic resource discovery and semantic query are key functionalities to find and utilize information stored in the system. Semantic mashups are an advanced functionality that enables combining and processing existing semantic information according to a mashup function to dynamically create new semantic information. Further advanced functionalities like semantic reasoning and semantics-based analytics have been identified, but have not been covered in Release 3.

8.2 Semantic annotations

8.2.1 Overview

As identified in the requirements, Semantic Annotation within the oneM2M platform is of key importance for supporting semantic functionalities within the oneM2M platform.

8.2.2 Semantic instance management

8.2.2.1 Overview

The management of semantic instances stored in the <*semanticDescriptor*> resource is one of key semantic functionalities, including the create, update and delete operations against semantic instances. A simple way to deal with the update of semantic instances is to overwrite the whole <*semanticDescriptor*> resource which might lead to data redundancy problems, but a more efficient approach is needed to handle the management of semantic instances. An approach using functions e.g SPARQL HTTP POST, SPARQL HTTP PUT etc provided by SPARQL to update or create semantic instances is proposed.

8.2.2.2 Concrete example of managing semantic instance

In the oneM2M system, in order to change a semantic instances in the <*semanticDescriptor*> resources, the system has to change the whole semantic descriptor.

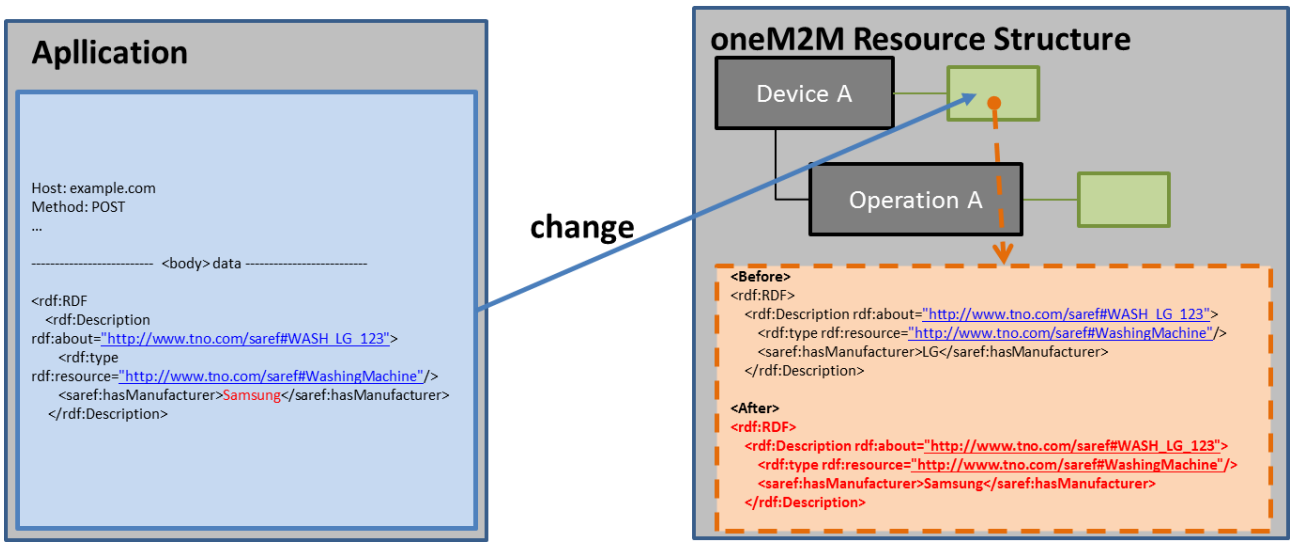

Figure 8.2.2.2-1: Concrete example of managing semantic instance in the oneM2M system

8.2.2.3 Managing semantic instances using SPARQL update operation

A semantic description consisting of semantic instances is contained in the <*semanticDescriptor*> resource. In order to manage semantic instances using the SPARQL update operation, it is assumed that the semantic description is stored in the <*semanticDescriptor*> resource as depicted in figure 8.2.2.3-1.

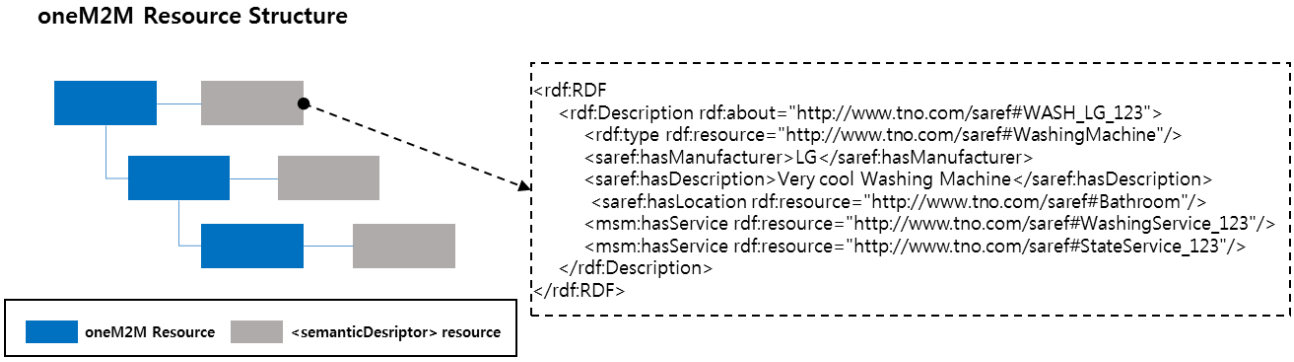

Figure 8.2.2.3-1: Representation of a semantic description in a <semanticDescripor> resource

To manage the semantic instances, the SPARQL update operation has to be mapped to the oneM2M UPDATE (U) procedure. The UPDATE procedure can be used by an AE Originator to manage the semantic instances stored in the <*semanticDescriptor*> resource on a Receiver CSE (also called the Hosting CSE).

Originator is responsible for sending requests to update semantic instances stored in the <*semanticDescriptor*> resource by using the UPDATE method.

Hosting CSE processes the update procedures against the requested semantic instances if the originator is allowed to do the update operation. Figure 8.2.2.3-2 shows the interaction between Originator and Receiver and the procedures are processed as follows.

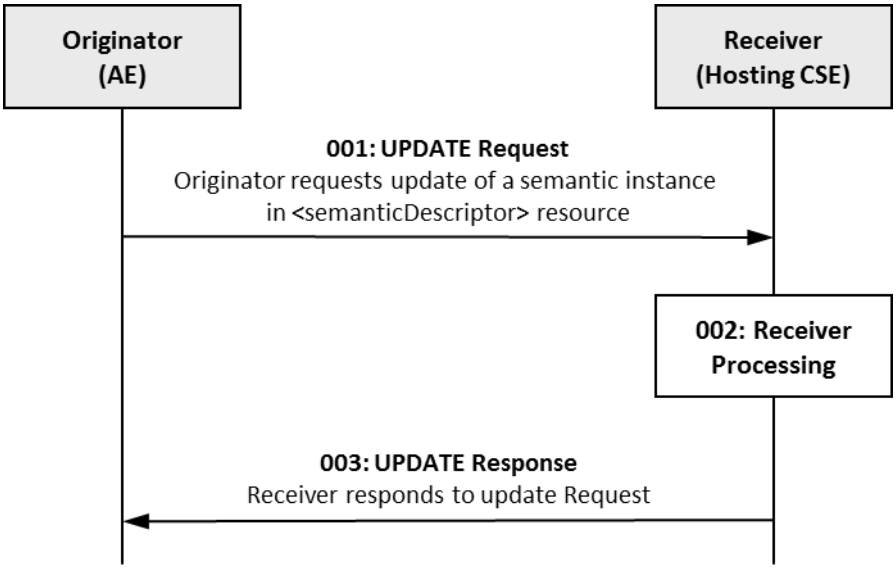

Figure 8.2.2.3-2: Procedure for managing the semantic instances using oneM2M UPDATE operation

Step 001: The Originator can include mandatory parameters and partial or whole optional parameters in the Request message for UPDATE operation. In **Step 001**, different SPARQL statements can be used in order to update (i.e. add, delete, and modify) an existing semantic description in a target <*semanticDescriptior*> resource at the Receiver as shown in the following cases.

- **Case 1:** This case is to add semantic instances (e.g. RDF, triples) to an existing semantic description in a target <*semanticDescriptor*> resource in the Receiver. In this case, the Originator can include `INSERT DATA` or `INSERT` SPARQL statements in a Request as shown in the block below. The `INSERT DATA` statement can add semantic instances using the `RDF PAYLOAD` to the semantic description in a target <*semanticDescriptor*> resource. Thus, `RDF PAYLOAD` in an `INSERT DATA` SPARQL statement disallows blank nodes or variables. On the other hand, an `INSERT` SPARQL statement can add semantic instances corresponding to a `template` by copying semantic instances from a source <*semanticDescriptor*> resource to a target <*semanticDescriptor*> resource based on a `pattern`. Accordingly, a `template` in the `INSERT` statement allows blank nodes or

variables with conditional SPARQL statements. Examples 1 and 2 give examples using `INSERT DATA` and `INSERT` SPARQL statements, respectively.

```
======================== INSERT DATA statement ========================
INSERT DATA
{GRAPH <Target URI of <semanticDescriptor> resource> {RDF PAYLOAD} }
========================== INSERT statement ===========================
INSERT {
GRAPH <Target URI of <semanticDescriptor> resource 1> {template} }
WHERE {GRPAH <Target URI of <semanticDescriptor> resource 2> {pattern} }
```

EXAMPLE 1: Add semantic instance to a <*semanticDescriptor*> resource using INSERT DATA statement:

```
INSERT DATA
{
GRAPH <http://<Hosting CSE address>/<CSEBase>/<AE>/<semanticDescriptor>>
{saref:WASH_LG_123 msm:hasOperation saref:WashingOperation_123}
}
```

EXAMPLE 2: Add semantic instance to a <*semanticDescriptor*> resource using INSERT statement:

```
INSERT {
GRAPH <http://<Hosting CSE address>/<CSEBase>/<AE>/<semanticDescriptor1>>
{?a saref:hasManufacturer ?c}
}
WHERE {
GRAPH <http://<Hosting CSE address>/<CSEBase>/<AE>/<semanticDescriptor2>>
{?a saref:hasManufacturer ?c}
}
```

- **Case 2:** This case is to remove any of the semantic instances from an existing semantic description in a target <*semanticDescriptor*> resource in the Receiver. In this case, the Originator can include two different SPARQL statements including `DELETE DATA` statement or `DELETE` statement in the Request as shown in the block below.
  The `DELETE DATA` statement can remove specific semantic instances using `RDF PAYLOAD` from an existing semantic description in a target <*semanticDescriptor*> resource. Hence, the `RDF PAYLOAD` in the `DELETE DATA` SPARQL statement is not allowed to contain blank nodes or variables. However, the `DELETE` SPARQL statement can remove semantic instances corresponding to the `template` through matching semantic instances based on a `pattern`. Accordingly, the `template` in the `DELETE` SPARQL statement is allowed to contain blank nodes or variables with conditional SPARQL statements. Examples 3 and 4 give examples using `DELETE DATA` and `DELETE` SPARQL statements, respectively.

```
======================== DELETE DATA statement ========================
DELETE DATA
{GRAPH <Target URI of <semanticDescriptor> resource> {RDF PAYLOAD} }
========================== DELETE statement ===========================
WITH <Target URI of <semanticDescriptor> resource>
DELETE {template }
WHERE {pattern}
```

EXAMPLE 3: Remove semantic instances in the <*semanticDescriptor*> resource using `DELETE DATA` statement:

```
DELETE DATA
{
GRAPH <http://<Hosting CSE address/<CSEBase>/<AE>/<semanticDescriptor>>
{saref:WASH_LG_123 msm:hasService saref:StateService_123}
}
```

EXAMPLE 4: Remove semantic instance in the <*semanticDescriptor*> resource using `DELETE` statement:

```
WITH <http://<Hosting CSE address/<CSEBase>/<AE>/<semanticDescriptor>>
DELETE {?a msm:hasService saref:StateService_123}
WHERE {?a saref:hasManufacturer 'LG'}
```

- **Case 3:** This case is to modify any of the semantic instances from the semantic description in a target <*semanticDescriptor*> resource in the Receiver. In this case, the Originator can include `DELETE/INSERT SPARQL` statements with `template1` and `template2` in the Request as shown in the block below. At this time, blank nodes or variables are allowed in each `template`.

```
WITH <Target URI of <semanticDescriptor> resource>
DELETE {template1} INSERT {template2}
WHERE {pattern}
```

EXAMPLE 5: Modify semantic instances in a <*semanticDescriptor*> resource using DELETE/INSERT operation:

```
WITH http://<Hosting CSE address/<AE>/<semanticDescriptor>
DELETE {?a saref:hasManufacturer 'LG'}
INSERT {?a saref:hasManufacturer 'SAMSUNG'}
WHERE {?a saref:hasManufacturer 'LG'}
```

Step 002: The Receiver will verify the existence (including ***Filter Criteria*** checking, if it is given) of the requested resource first and whether the Originator has the appropriate privilege to update the requested resource. On successful validation, the Receiver can update the semantic instances according to the SPARQL statements in the Request message. The update procedures are processed as follows:

- **According to Case 1:** If the `INSERT DATA` SPARQL statement is included in Request message, the `RDF PAYLOAD` in the statement will be added to the target <*semanticDescriptor*> resource. However, if the `RDF PAYLOAD` already exists in the target <*semanticDescriptor*> resource, then the Receiver will return a failure request status with additional error information through **Step 003**. If the `INSERT` SPARQL statement included in the Request message, the Receiver adds semantic instances corresponding to `template` by copying semantic instances from the source <*semanticDescripor*> resource to the target <*semanticDescriptor*> resource based on the `pattern`. At this time, if there are no existing semantic instances corresponding to `template` or matched semantic instances based on `pattern`, the Receiver will return a failure request status with additional error information through **Step 003.** The blocks below show the processing result of Example 1 and Example 2 presented in **Case 1**.

Result of Examples 1 and 2 presented in Case 1 using semantic description in figure 8.2.2.3-1.

Description before:

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>LG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
        <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
    </rdf:Description>
</rdf:RDF>
```

Description after:

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>LG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
         <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
        <msm:hasOperationrdf:resource="http://ontology.tno.nl/saref#WashingOperation_123"/>
    </rdf:Description>
</rdf:RDF>
```

- **According to Case 2:** If the `DELETE DATA` SPARQL statement is included in Request message, then the `RDF PAYLOAD` in the statement will be removed from the target <*semanticDescriptor*> resource. At this time, if the `RDF PALOAD` does not exist in the resource, then the Receiver can return a failure request status with additional error information through **Step 003**. If the `DELETE` SPARQL statement is included in the Request message, the Receiver will remove all of semantic instances corresponding to the template based on a pattern. Accordingly, if there are no existing semantic instances corresponding to `template` or matched semantic instances base on `pattern`, then the Receiver will return a failure request status with additional error information through **Step 003**. The block below shows the processing result of Example 3 and Example 4 presented in **Case 2**.

Result of Example 2 presented in Case 2 using semantic description in figure 8.2.2.3-1.

Description before:

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>LG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
         <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
    </rdf:Description>
</rdf:RDF>
```

Description after:

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>LG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
         <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
    </rdf:Description>
</rdf:RDF>
```

- **According to Case 3:** If the target semantic instance included in the Request message exists in the <*semanticDescriptor*> resource, then the Receiver will remove all semantic instances corresponding to `template1` from the <*semanticDescriptor*> resources, and then the Receiver will create new semantic instances corresponding to `template2` in the <*semanticDescriptor*> resource base on `pattern`. In case that either the target semantic instance corresponding to `template1` requested to be removed does not exist in the <*semanticDescriptor*> resource or the target semantic instance corresponding to `template2` requested to be added to the <*semanticDescriptor*> resource already exists, the Receiver will return a failure request status with additional error information through **Step 003**. The block below shows the processing result of Example 3 presented in **Case 3**.

Result of Example 3 presented in Case 3 using semantic description in figure 8.2.2.3-1.

Description before

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>LG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
         <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
    </rdf:Description>
</rdf:RDF>
```

Description after:

```
<rdf:RDF
    <rdf:Description rdf:about="http://ontology.tno.nl/saref#WASH_LG_123">
        <rdf:type rdf:resource="http://ontology.tno.nl/saref#WashingMachine"/>
        <saref:hasManufacturer>SAMSUNG</saref:hasManufacturer>
        <saref:hasDescription>Very cool Washing Machine</saref:hasDescription>
         <saref:hasLocation rdf:resource="http://ontology.tno.nl/saref#Bathroom"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#WashingService_123"/>
        <msm:hasService rdf:resource="http://ontology.tno.nl/saref#StateService_123"/>
    </rdf:Description>
</rdf:RDF>
```

Step 003: The Receiver will include all mandatory parameters and partial or whole optional parameters in the Response message for the UPDATE operation.

8.3 Semantic validation

8.3.1 Problem statement

The <semanticDescriptor> resource contains a *descriptor* attribute which can store any RDF triples as the semantic description (i.e. annotation) of the associated resource (usually the parent resource of the <semanticDescriptor>). In the same time, <semanticDescriptor> resource contains a *ontologyRef* attribute, which is a reference (URI) of the ontology used to represent the information that is stored in the *descriptor* attribute.

Normally, the triples stored in the *descriptor* attribute should be compliant with the ontology referenced by the *ontologyRef* attribute. However, there is no guarantee that an issuer (e.g. a 3rd party AE) which creates or updates the <semanticDescriptor> will always provide the consistent information. In case the semantic description (as triples in *descriptor* attribute) is not compliant with the referneced ontology, it basically means the <semanticDescriptor> is not valid and cannot be used by the AE and/or CSE properly. For example, it may cause a consumer application to mis-interpret the data semantics and types (e.g. termperature vs. humidity) so that lead to a wrong re-action (turn on/off a heater vs. a humidifier). This will undermine the benefits of semantic annotation for the IoT data and will hinder the usage of more advanced features of semantic enablement e.g. reasoning and mash-up.

Following are two detailed examples showing the potential inconsistency between the <semanticDescriptor> resources and the referenced ontology. Common assumption is described below.

The referenced ontology is called 'OntologyH', which represent the ontology of humidity sensor of domain X. The graphical representation of 'OntologyH' is show in figure 8.3.1-1. Besides that, OntologyH holds the following statement: " *controllingFunction ow:disjointwith measuringFunction*", meaning a 'controllingFunction' cannot be a 'measuringFunction' instance at the same time.

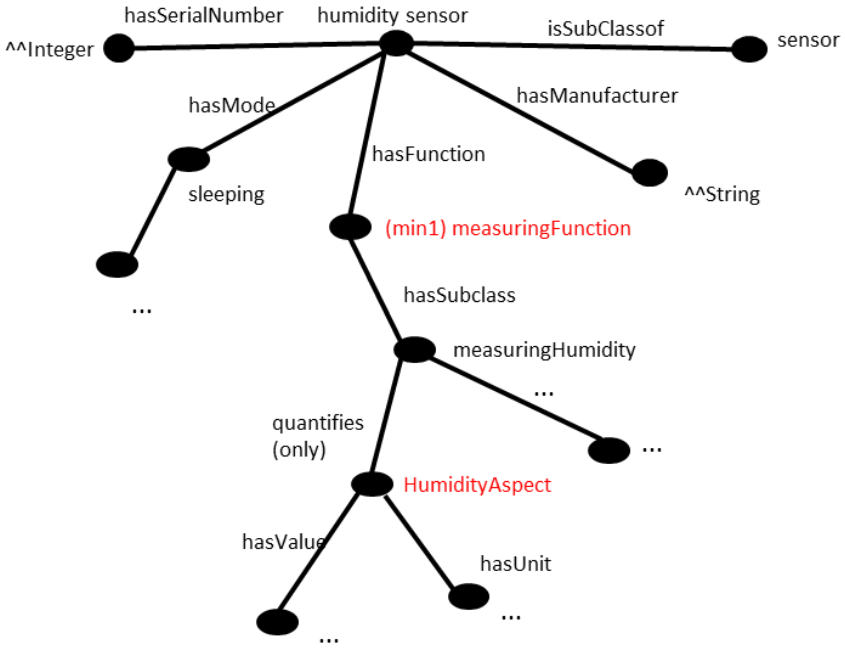

Figure 8.3.1-1: OntologyH (humidity sensor ontology of domain X)

A humidity sensor named 'sensor12' is reprented as an AE resource with its child resources as shown in figure 8.3.1-2. The semantic description of 'sensor12' is represented in the <semanticDescrptorX> child resource, where the *ontologyRef* attribute points to 'OntologyH'.

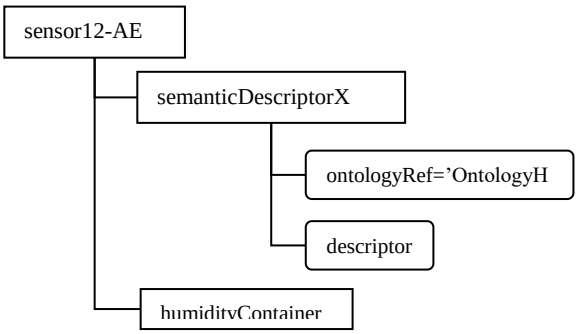

Figure 8.3.1-2: oneM2M resource representation of humidity sensor 'sensor12'

Case 1: stand-alone <semanticDescriptor>

In this case, all semantic description about 'sensor12' is self-contained in the <semanticDescriptorX> resource. Figure 8.3.1-3 further illustrate the RDF triples of the *descriptor* attribute of <semanticDescriptorX>.

```
…
<rdf:Description rdf:about= "…/sensor12"
    <rdf: type ont:humiditySensor/>
    < ont:hasfunction rdf:resource="…/humidityContainer" />
…
</rdf:Description>
…
< rdf:Description rdf:about="…/humidityContainer"
    <rdf:type ont:controllingFunction/>
…
</rdf:Description>
…
```

Figure 8.3.1-3 RDF triples of <semanticDescriptorX> - case 1

Note that in the *descriptor*, it claims that 'sensor12' has the function 'humidityContainer' as an instance of class 'controllingFunction', while according to the renferece 'OntologyH', 'humidityContainer' can only be the instance of class 'measuringFunction'. This causes the inconsistency issue.

Case 2: linked <semanticDescriptors>

Case 2 has a similar setup as Case 1, the difference is that in Case 2, the *descriptor* attribute of <semanticDescriptorX> resource contains *m2m:resourceDescriptorLink* pointing to another <semanticDescriptorR> resource, which is the semantic description of a remote resource 'roomA'. A similar case could be that the <semanticDescriptorR> resource is linked by the *relatedSemantics* attribute of <semanticDescriptorX> resource. The RDF triples of both resources are shown in figures 8.3.1-4 and 8.3.1-5 respectively.

```
<rdf:Description rdf:about= "…/sensor12"
<rdf: type ont:humiditySensor/>
< ont:hasfunction rdf:resource="…/humidityContainer" />
…
</rdf:Description>
…
< rdf:Description rdf:about="…/humidityContainer"
        <on:quantifies rdf:resource="…/roomA/tempAspect"/>
…
</rdf:Description>
< rdf:Description rdf:about ="…/roomA/tempAspect"/>"
        < m2m:resourceDescriptorLink rdf:resource="…/semanticDescriptorR"/>
…
</rdf:Description>
```

Figure 8.3.1-4: RDF triples of <semanticDescriptorX> - case 2

```
…
< rdf:Description rdf:about ="…/tempAspect"/>"
    <rdf:type rdf:resource="ont:TemperatureAspect"/>
…
</rdf:Description>
```

Figure 8.3.1-5: RDF triples of <semanticDescriptorR> - case 2

Note that in <semanticDescriptorX>, 'humidityContainer' has 'roomA/tempAspect' as the quantified aspect and in <semanticDescriptorR> 'roomA/tempAspect' is claimed as an instance of class 'TemperatureAspect'. However, according to 'OntologyH', the measurement of 'sensor12' can only quantify instances of class 'HumidityAspect'. This also causes the inconsistency issue.

8.3.2 Proposed solution

To solve the issues mentioned above, a semantic description validation common services function (CSF) is proposed to validate a <semanticDescriptor> resource against the referenced ontology. The end-to-end flow for semantic description validation is shown in figure 8.3.2-1. The extention to the resource structure is defined in clauses 7.3.1 and 7.3.2.

Message flow

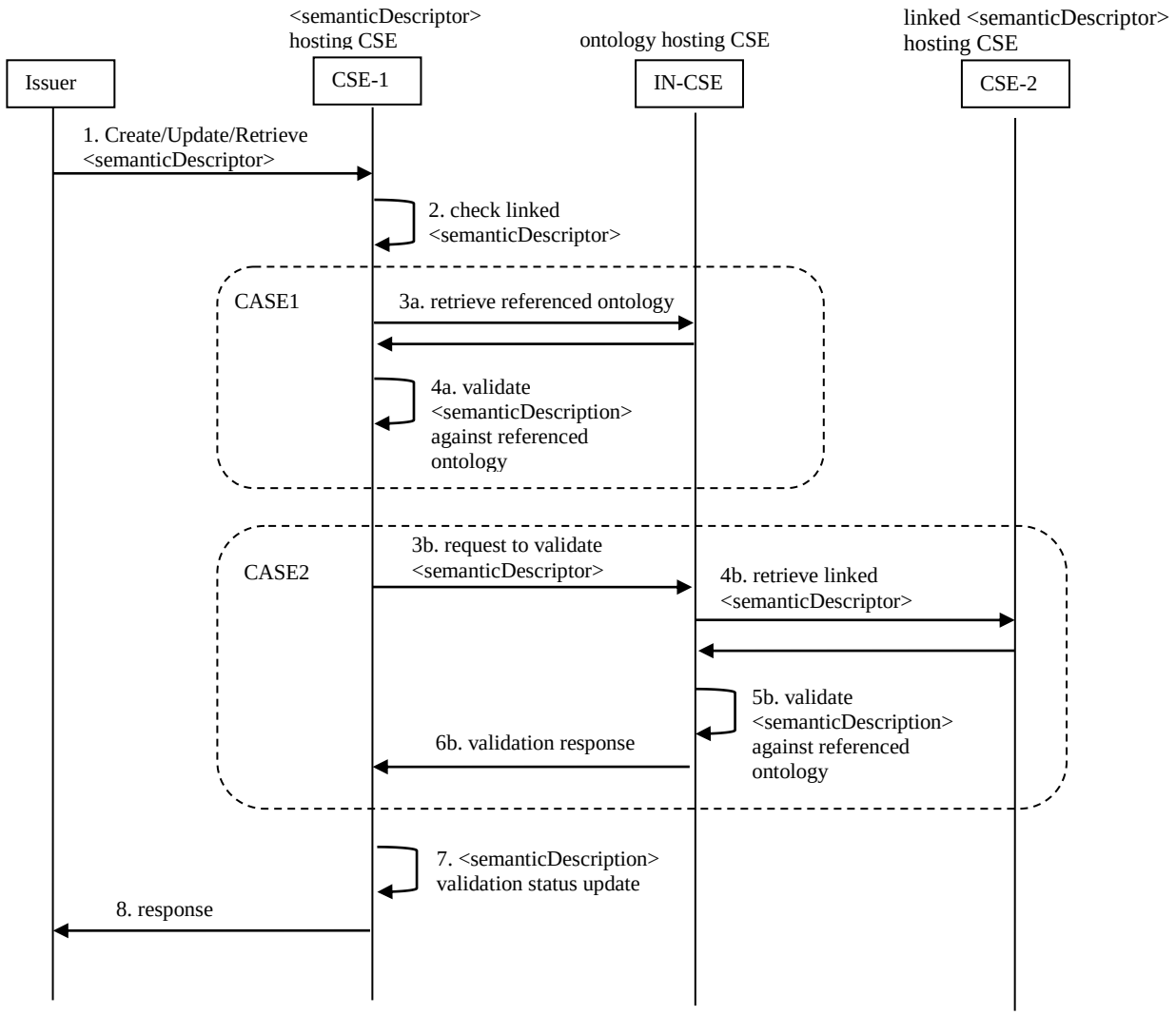

Figure 8.3.2-1: Message flow for semantic description validation

Step 1. The issuer (e.g. an AE or a CSE) sends to CSE-1 an operation request (CREATE, UPATE or RETRIEVE) pertaining to the <semanticDescriptor> resource hosted by CSE-1. In the case of CREATE or UPDATE, the request contains the <semanticDescriptor> resource representation, which includes the semantic description (triples in *descriptor* attribute), the URI to the referenced ontology (i.e. *ontologyRef*) and potential links to other <semanticDescriptor> resources (as in *relatedSemantics* attribute, or in the triples with *m2m:resourceDescriptorLink* in *descriptor* attribute). In the case of RETRIEVE, the request contains the URI of the <semanticDescriptor> resource.

Optionally, the request may contain a parameter (e.g. *needValidation*, which is to be specified) to indicate that the addressed <semanticDescriptor> needs to be validated before creation, update or retreival. Such indication may

alternatively (e.g. in the case of CREAET and UPATE) be indicated by setting an extended attribute *validationEnable* of the <semanticDescriptor> resource as defined in clause 7.3.1.

Step 2. CSE-1 receives the operation request and check if the addressed <semanticDescriptor> resource is linked to any other remote <semanticDescriptor> resources (i.e. check if the addressed <semanticDescriptor> resource contains *relatedSemantics* attribute, or triples with *m2m:resourceDescriptorLink* in *descriptor* attribute). If no, the procedure goes to Case 1 (step 3a to 4a), otherwise, goes to Case 2 (step 3b to 6b). In the case of RETRIEVE, CSE-1 may only trigger the following steps if the addressed <semanticDescritpor> resource is not yet validated (by checking the *semanticValidated* attribute as defined in clause 7.3.1).

Case 1: stand-alone <semanticDescritpor> (see also clause 8.3.1 Case 1)

Step 3a. CSE-1 retrieves the ontology reprentation according to the URI in the *ontologyRef* attribute of the addressed <semanticDescriptor> resource from IN-CSE (which hosts the referenced ontology) . In case CSE-1 cannot retrieve the ontology reprensentation (due to access right control or other exceptional reasons), skip **Step 4a.**

Step 4a. CSE-1 performs semantic validation of the semantic description (the triples in *descriptor* attribute) of the addressed <semanticDescriptor> resource against the retrieved ontology.

Case 2: linked <semanticDescritpor> (see also clause 8.3.1 Case 2)

Step 3b. CSE-1 sends a semantic validation request to IN-CSE (the hosting CSE of the referenced ontology according to *ontologyRef*) with the addressed <semanticDescriptor> resource representation (which includes *descriptor*, *ontologyRef* and URIs to other linked <semanticDescriptor> resources) to validate the semantic description of the addressed <semanticDescriptor> resource (together with the linked <semanticDescriptor> resources) against the referenced ontologies of the addressed and the linked <semanticDescriptor> resources.

The Ontology Repository resource on the IN-CSE may need to be extended with an additional attribute or sub-resource (e.g. <*semanticValidation*> as defined in clause 7.3.2) as the RESTful interface to accept the semantic validation request.

Step 4b. After receiving the semantic validation request from CSE-1, IN-CSE retrieves the linked <semanticDescriptor> resources (including the semantic description in *descriptor* attribute and the URI of the referenced Ontology in *ontologyRef* attribute). In case the linked <semanticDescriptor> resources also contain URIs linking to other <semanticDescriptor> resources, IN-CSE may repeat this step iteratively to retrieve all linked <semanticDescriptor> resources. In case IN-CSE cannot retrieve the linked <semanticDescriptor> resources (due to access right control or other exceptional reasons), skip **Step 5b.**

Step 5b. IN-CSE validates the semantic description of the addressed <semanticDescriptor> resource and the retrieved linked <semanticDescriptor> resources against the referenced ontologies of the addressed and the linked <semanticDescriptor> resources.

Step 6b. IN-CSE returns the validation response to CSE-1. In case **Step 5b** succeeds, the response code is successful, otherwise (including **Step 5b** is skipped due to **Step 4b** fails), the response code is failed.

Step 7. CSE-1 performs normal operation (CREATE, UPDATE or RETRIEVE) on the addressed <semanticDescriptor> resource according to the original request from the issuer following existing procedures. In addition, based on the validation result of **Step 4a** (in **Case 1**) or the validation response received in **Step 6b** (in **Case 2**), CSE-1 updates the validation status (validated or not) of the addressed <semanticDescriptor> resource accordingly. If **Step 4a** is skipped due to **Step 3a** fails, it's also considered as not validated.

To do this, <semanticDescriptor> resource needs be extended with an additional attribute *semanticValidated* to represent the validation status as defined in clause 7.3.1.

Step 8. CSE-1 returns operation (CREATE, UPDATE or RETRIEVE) response to the issuer.

NOTE: Current assumption is that all referenced ontologies are hosted by the IN-CSE, no matter how the ontololgies are physicaly stored (in the IN-CSE, in a seperate database, or on an external website).

8.3.3 Aspects to be checked in semantic validation

Several aspects need to be checked in order to make sure that the semantic description consists of valid RDF triples and they are indeed capable of interoperating semantically with other oneM2M resources. Taking into account the nature of semantically annotated data, we can distinguish four levels of validation:

1) **Lexical check.** This level of check consists of verifying the correctness of RDF serialization regarding to the declared type. For example, the <*semanticDescriptor*> resource is marked in XML representation (according to the *descriptorRepresentation* attribute) whereas the semantic annotation (in the *descriptor* attribute) is indeed serialized in JSON, or the XML document contains some error that the parser cannot accomplish its job, the lexical check fails.

2) **Syntactic checks.** After the basic lexical checks, the syntactic check consists of verifying the correctness of the "syntax" of the RDF triples, more specifically:

   a) **Untyped of resources and literals**. Here resource refers to instances of a class, and literal refers to a textual or numerical value. The type of resource or literal is the link of an annotation to the ontology which enables the semantic capabilities. Any un-typed element presented in an annotation is problematic towards the semantic interoperability.

   b) **Ill-formed URIs**. URI is essential and critical for identification of a resource. They should be checked against RFC3968 which defines the generic syntax of URI.

   c) **Problematic prefix and namespaces**. Namespaces play the role of linking the annotation to the reference ontologies and vocabularies. If the URI of the namespace is problematic (e.g. wrong URI, URI contains illegal character), it may cause others to mis-interpret the data semantics and types. Prefix is a unique reference to replace the namespaces in the local file. A one-to-one mapping between the prefix and namespace is essential to ensure a correct reference.

   d) **Unknown classes and properties**. A prerequisite of semantic interoperability is that all the resources use a common and agreed vocabulary. As consequence, if any resource uses in its annotation a class or property that is not defined in the reference ontology(ies), other resources would have no way to understand it, so that the semantic interoperability is impossible.

3) **Semantic checks.** Following a successful syntactic validation, the semantic check consists of verifying the logical consistence of the semantic annotation regarding to the reference ontology(ies):

   a) **Cardinality inconsistency**. If the ontology defines that class A can have one and only one instance of child class B, and in the annotation, there are two instances of B related to one instance of A, there is a problem.

   b) **Problematic relationship or inheritance**. Following the relationship defined in the reference ontology, if an instance of a class A is wrongly annotated to be at same time an instance of class B which is disjoint from class A, there is a conflict and the instance cannot be resolved by the semantic engine. A concrete example in detailed in clause 8.3.1.

The validation response returned to the issuer depends on the result of each of the above tests. To conclude that an annotation is validated, ideally all the above checks need to be performed and passed. However, as several tests are independent from others (for example: 3.a and 3.b do not have an impact on each other), several "validated profiles" can be defined by choosing to include different aspects to be checked.

8.4 Semantic filtering and discovery

Semantic filtering is about using a semantic filter expressed in SPARQL [i.14] on the semantic annotation of a resource. The semantic annotation is contained in a <*semanticDescriptor*> child resource. The goal of applying the filter is to determine whether the (parent) resource fits the criteria specified in the filter. It is used for the purpose of discovery, i.e. if the SPARQL query applied to the semantic content of the <*semanticDescriptor*> resource returns a non-empty result, its parent resource is part of the discovery result, otherwise not. In some cases, not the complete semantic annotation is contained in a single <*semanticDescriptor*>resource, e.g. to avoid having to replicate the same information in many different places. In this case, the <*semanticDescriptor*> resource can contain links to related <*semanticDescriptor*> resource whose content is also to be taken into account when applying the filter. Two approaches are supported, an annotation-based method, where the links are embedded in the semantic information itself, and a resource link-based, where the links are included in an attribute of the <*semanticDescriptor*> resource, which may also include a group resource that groups a number of <*semanticDescriptor*> resources.

Semantic discovery is part of the general request procedure description in oneM2M TS-0001 [i.3], clause 8.1.2, using the discovery option with a *semanticsFilter* set. The details of how the linking to other <*semanticDescriptor*> resources is used for discovery is described in oneM2M TS-0034 [i.12], clause 7.4.

8.5 General semantic queries

8.5.1 Introduction

In Release 2, the only semantic operation supported by oneM2M is semantic resource discovery. As a result of a semantic resource discovery on a resource, the addresses of the child resources are returned that have a semantic descriptor resource ,which (together with possibly linked other semantic descriptor resources) fits the provided semantic filter.The semantic filter is specified in form of a SPARQL query. Below an example for a semantic resource discovery is provided:

- Semantic discovery (R2): Give me the resources that can provide occupancy information for meeting rooms.

Once the addresses of the resources are known, their semantic descriptor resources and the related content information can be retrieved by the application. The drawback is that this requires a potentially huge number of interactions and the application has to retrieve a lot of resource content which may be irrelevant in the end. If only the information is required, it would be much easier to directly ask for the information:

- Semantic query: What are the meeting rooms for which I can retrieve the current occupancy information.

It would even be more useful to not just be able to access the (meta) information contained in the <semanticDescriptor> resource, but to also access the actual content - in this case the occupancy information - at the same time:

- Semantic query including content: Give me the current occupancy information for meeting rooms.

If not all information is relevant, but it depends on the actual content, it would be even more useful to allow restrictions, which could be specified in SPARQL:

- Semantic query including content with restriction: Give me the meeting rooms whose current occupancy is "empty".

To keep the discussion about semantic queries independent of the discussion of what kinds of resources containing semantic information should exist in oneM2M, we will use the term *semantic resource* in the remainder of clause 8.5. Semantic resources currently include <smanticDescriptor> resources, but could in the future contain other semantic resources, e.g. semantic content resources, semantic container resources, etc. The discussion here should cover all supported semantic resources.

8.5.2 Semantic query vs. semantic resource discovery

In general, semantic queries enable the retrieval of both explicitly and implicitly derived information based on syntactic, semantic and structural information contained in data (such as RDF data). Currently oneM2M already supports semantic resource discovery through semantic filter. However, some essential differences between semantic query and semantic resource discovery need to be taken into consideration when realizing the semantic query feature, which is listed in table 8.5.2-1 from several aspects.

Table 8.5.2-1: Comparison semantic query and semantic resource discovery

Aspects	Semantic Query	Semantic Resource Discovery
Objective	The objective of Semantic Query is to extract "useful knowledge" over a set of "RDF data basis".	Semantic resource discovery is mainly targeted for discovering resources for further resource accessing (e.g. CRUD operations).
Technical Focus	Semantic Query is a more advanced semantic-centric feature. For example, the query result can include any derived knowledge based on the query statement.	Semantic resource discovery is a more resource-oriented feature to leverage semantics to enable sophisticated resource discovery.
Processed Result	The semantic query result (in term of "knowledge") can be returned as semantic information to answer the query, i.e. not just resources URIs.	The processed result of a semantic resource discovery is mainly to include a list of identified resource URIs.

Below is an example from W3C (see https://www.w3.org/2009/Talks/0615-qbe/), which illustrates the above perspectives:

Query Example: Find me all the people in Tim Berners-Lee's FOAF (Friend of a friend) file that have names and email addresses. Return each person's URI, name, and email address.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?person foaf:name ?name .
    ?person foaf:mbox ?email .
}
```

Query Expected Results:

Person	Name	Email
<http://www.w3.org/People/karl/karl-foaf.xrdf#me>	"Karl Dubost"	<mailto:karl@w3.org>
<http://www.w3.org/People/Berners-Lee/card#amy>	"Amy van der Hiel"	<mailto:amy@w3.org>
<http://www.w3.org/People/Berners-Lee/card#edd>	"Edd Dumbill"	<mailto:edd@xmlhack.com>

8.5.3 Introduction to semantic graph scoping (SGS)

To complete a given semantic query operation, two parts have to be in place:

1) **The Query Statement**: A semantic query is normally initialized by a semantic query user based on his/her needs, which can be specified as a SPARQL query.

2) **The RDF Data Basis**: This is the RDF triples where the semantic query is to be executed on, in order to yield the query result.

Given the oneM2M resource-oriented architecture, semantic query may need to be done directly over distributed semantic descriptors when there is no centralized Triple Store available. In particular, a unique situation is that the RDF data basis needed by a given semantic query may be dispersed in the resource tree and stored in different <semanticDescriptor> resources. Note that, those <semanticDescriptor> resources are often the child resources of normal oneM2M resources, and their major purpose currently is for semantic annotation and supporting semantic resource discovery, not directly for semantic query.

More importantly, a desirable query result for a given semantic query can be efficiently yielded only if the query is conducted over an "appropriate" RDF data basis, such a fundamental process for forming a RDF data basis is called a "**Semantic Graph Scoping** (**SGS**)" process. For example, as an illustration shown in figure 8.5.3-1., a given SPARQL query can be executed on a RDF data basis, which is formed through a SGS process and is constituted by the RDF triples from three <semanticDescriptor> resources, i.e. <SD_1>, <SD_2> and <SD_3>. It is worth noting that the query result in terms of derived knowledge of this query will be limited to the information included in those three <semanticDescriptor> resources.

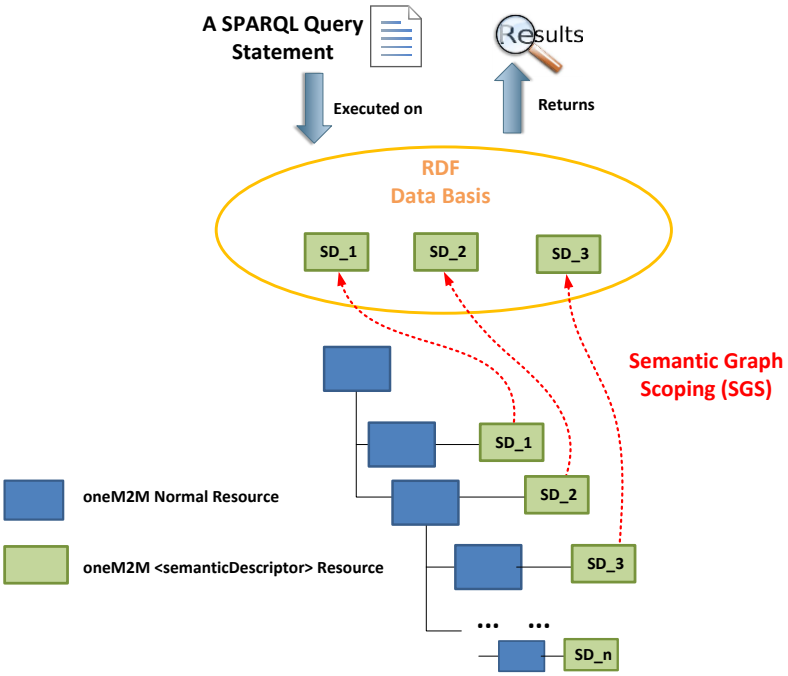

Figure 8.5.3-1: Semantic descriptor discovery needed by semantic query over distributed semantic descriptors

Note that, SGS is an essential process for supporting semantic query feature over distributed semantic descriptors. For example, in the semantic resource discovery, semantic descriptors are mainly leveraged as "utilities" in order to identify normal oneM2M resources. In comparison, SGS is to discover the semantic descriptor themselves such that they can form a RDF data basis for a given semantic query.

In addition, semantic query has more stringent requirements compared to the semantic resource discovery. For example, the originator of a semantic resource discovery can decide where the resource discovery starts by specifying a start point using the "To" parameter in an oneM2M request message, and the worst case is that the resource discovery cannot return any satisfied resource URI. In comparison, for a given semantic query, the quality of the query result sometimes may be limited if it is not able to form an appropriate RDF data basis for this query by employing appropriate SGS.

For example, consider an example shown in figure 8.5.3-2 and the semantic query is formulated as "return me how many devices that their OperationA have the operation status larger than 60". To answer this query (the desired answer is 3 since the operation status of all the Device12, Device34 and Device56 are larger than 60 now), ideally the SGS process should collect all of three involved <semanticDescriptor> child resources of Device12, Device34 and Device56 (assuming that the access control is not an issue), which forms the appropriate RDF data basis for answering this query. Otherwise, the quality of the query result may be limited to the RDF data basis formed by a SGS process if it does not include all of the three <semanticDescriptor> resources. Therefore, it can be seen that how SGS could potentially collect the involved <semanticDescriptor> resources needs to be specified, which is introduced in a later clause.

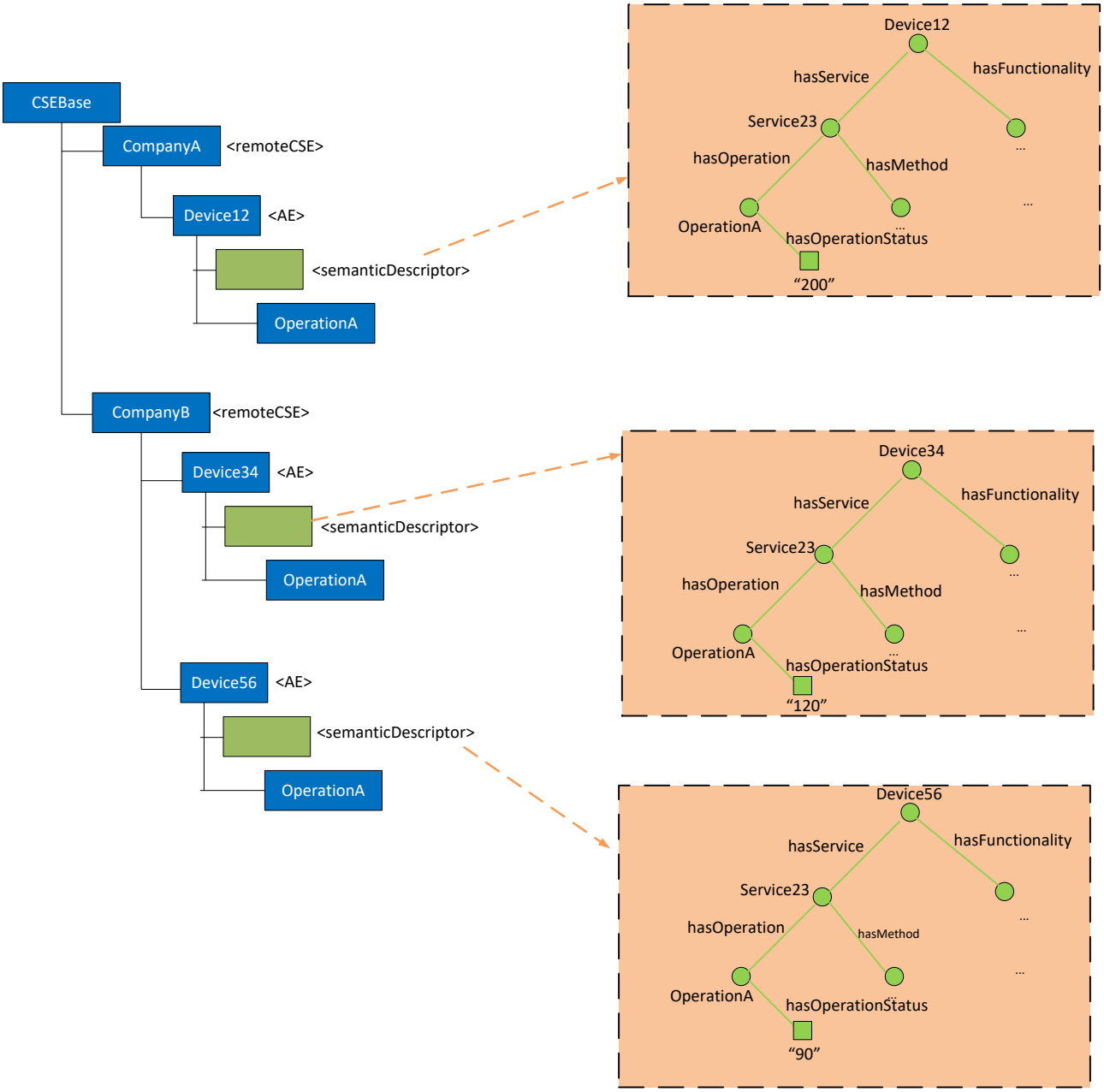

Figure 8.5.3-2: A semantic query example over distributed semantic descriptors

8.5.4 Solutions to semantic graph scoping

In this clause, several potential solutions are specified for the SGS process, and each of them has their own methodology, as well as pros/cons, which are discussed in details. Two major entities are involved, i.e. the originator (i.e. a semantic query user sending a query request message), and the receiver that is to conduct the semantic query processing and yield the query result.

Solution-1: Solution-1 is a straightforward solution in that the SGS is conducted at a local CSE. In particular, when an originator sends a semantic query to a receiver, it directly targets to the <CSEBase> resource. Accordingly, the SGS process at the receiver side will discover all the <semanticDescriptor> resources under <CSEBase> as the RDF data basis for this query as long as access control policy is allowed. Note that, this is a best-effort approach in the sense that the SGS will try to form a data basis as wide as possible, in which it is also possible that the triples from many <semanticDescriptor> resources may not be relevant to the query. In particular, such an SGS may also incur considerable processing cost when trying to traverse the whole resource tree.

By examining the working procedures of existing semantic resource discovery over distributed descriptors as specified in clause 8.5.5, the following two enhanced solutions are introduced.

Solution-2: Solution-2 highly follows the existing practice on semantic resource discovery in the sense that the SGS may still start with the URI as specified by the "**To**" parameter and how semantic descriptors can be collected is similar with the process as done in the semantic resource discovery as introduced in clause 8.5.5. During the SGS process, some existing practices adopted by the existing semantic resource discovery can also be leveraged. For example, when SGS discovers <semanticDescriptor> resources for forming a RDF data basis, the existing resourceDescriptorLink property (and other similar utilities) can also be leveraged. For example, similar to the semantic resource discovery, when a SGS discovers a specific <semanticDescriptor> resource, it is possible that this resource may be linked with other <semanticDescriptor> resources in the local or remote CSEs through the resourceDescriptorLink property. Accordingly, the SGS can also follow the resourceDescriptorLink in order to add other linked <semanticDescriptor> resources into the RDF data basis.

Regarding to another unique aspect related to query result quality, this solution allows for the case when the SGS cannot form an appropriate RDF data basis. Accordingly, the following mechanism can be adopted. For example, in order to enable the originator to tolerate or be aware of the quality of the query result, one thing at the receiver side could do is to not only return the query result, but also piggyback certain query summary information as a feedback in order to enable the originator to have some sense regarding how to interpret/evaluate the query result. For example, the summary information could explain how the query was processed, the overview of the RDF data basis used for this query, e.g. how many <semanticDescriptor> resources were involved in this query, etc.

Solution-3: In this approach, the semantic query process is decoupled with the SGS process. For example, without processing any semantic query, at the receiver side, it may proactively aggregate some related <semanticDescriptor> resources together by using existing <group> and/or < semanticGroup> resources (as defined in existing oneM2M TS-0001 [i.3] and oneM2M TR-0007 [i.13]). Still taking the same example used previously, as shown in figure 8.5.4-1, the receiver or any other 3rd party entity may proactively create a <Group-1> resource, in which the members are the <semanticDescriptor> child resources of all the devices. The <semanticDescriptor> resource of the <Group-1> itself can be used to give the description about this group.

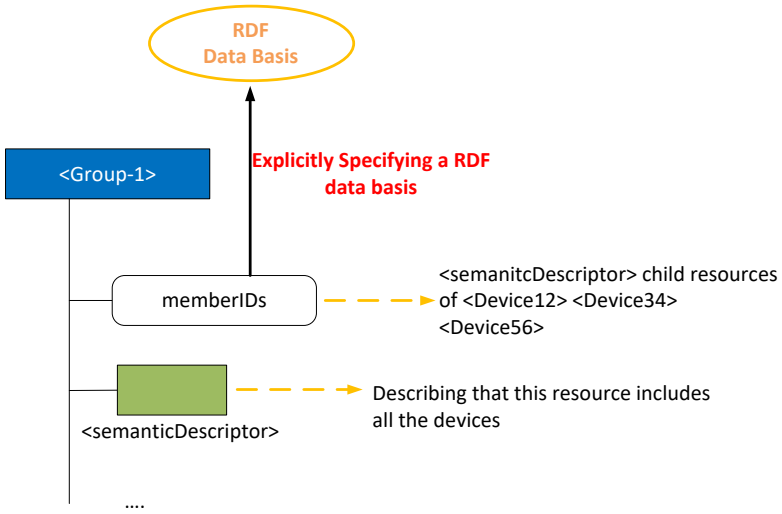

Figure 8.5.4-1: Use <group> resource for explicitly specifying an RDF data basis

For the originator, it may first discover various <group> resources to see which one have the most appropriate RDF data basis for the query to be launched. After such a <group> resource is selected, the originator may send its semantic query to the selected <group> resource. At the receiver side, the semantic query will be executed on the member resources of the selected <group> resource, which will be the data basis of this query (More general, it is also possible that the originator can form a RDF data basis by selecting more than one <group> resource). Last, the partial query results from each of member resources will be combined together as the final query result before being sent back to the originator. It is worth noting that the advantages for this approach include:

1) the receiver side can leverage its own knowledge on the resource tree to form some useful RDF data basis candidates;

2) the originator can have a clear understanding on the RDF data basis where the intended query is to be conducted. The disadvantage is that it is required that the receiver side needs to create different <group> or <semanticGroup> resources in order to form different RDF data basis candidates that may potentially be needed by various semantic queries.

It is worth noting that although this solution also proposes to leverage the <group> and/or <semanticGroup> resources for SGS, it has the fundamental differences compared to how those resources are used in the existing semantic resource discovery. For example, in semantic resource discovery, <group> resources are mainly used "**internally**" by the receiver when conducting a semantic resource discovery (i.e. the originator does not know any details regarding to how semantic descriptors are being organized/linked and collected, which is only known/used by the receiver side). By comparison, in Solution-3, the <group> resources and its contents can be directly exposed to the originator as RDF data basis candidates. In other words, those <group> resources are representing RDF data basis candidates and are visible to the originator (i.e. not just being used internally by the receiver) such that the originator now is also involved in the SGS process, which is a different working procedure compared to the existing semantic resource discovery.

Overall, the table 8.5.4-1 has summarized the pros/cons for each of the proposed solutions as discussed above, respectively. As can be seen from the table that, Solution-2 and Solution-3 have their own advantages, and both of them outperform the native Solution-1.

Table 8.5.4-1: Pros/cons of solutions

Solutions	Pros	Cons
Solution-1	• Simple implementation. • Best effort discovery with wide search scope.	• Considerable discovery cost due to full-traversal of whole resource tree. • Considerable query processing due to a large data basis.
Solution-2	• Follow the current practice on semantic resource discovery. • Less discovery cost due to a controllable discovery scope. • The originator may have some sense on the quality of the query result with the help of result summary.	• Relevantly more complex implementation is needed, compared to Solution-1.
Solution-3	• Fully decouple SGS with the query processing stage. • The RDF data basis candidates can be proactively created at the receiver side. • The originator has its own options to flexibly choose the appropriate RDF data basis candidate for its intended query, based on its subjective needs. • The originator may have clear understanding what data basis the query is to be executed on.	• Relevantly more complex implementation is needed, compared to Solution-1.

8.5.5 Implicitly scoped semantic queries

8.5.5.1 Introduction

Implicitly-scoped semantic queries are those that are based on a target resource in the oneM2M resource tree and the semantic query is executed relative to this target resource. This is the same as for other oneM2M requests, e.g. in the case of a discovery request, the child resources of the target resource which fit the filters specified in the request are returned. In the same way, a semantic query is assumed to be executed with respect to the semantic content that is available in the aggregation of the semantic information included in the semantic resources that are descendants of the target resource, e.g. <semanticDescriptor> resources of the target resource, its child resources and further descendants. Thus, by targeting a resource in the resource tree, the scope of the semantic query is implicitly defined as this aggregation of semantic information.This situation is depicted in figure 8.5.5.1-1.

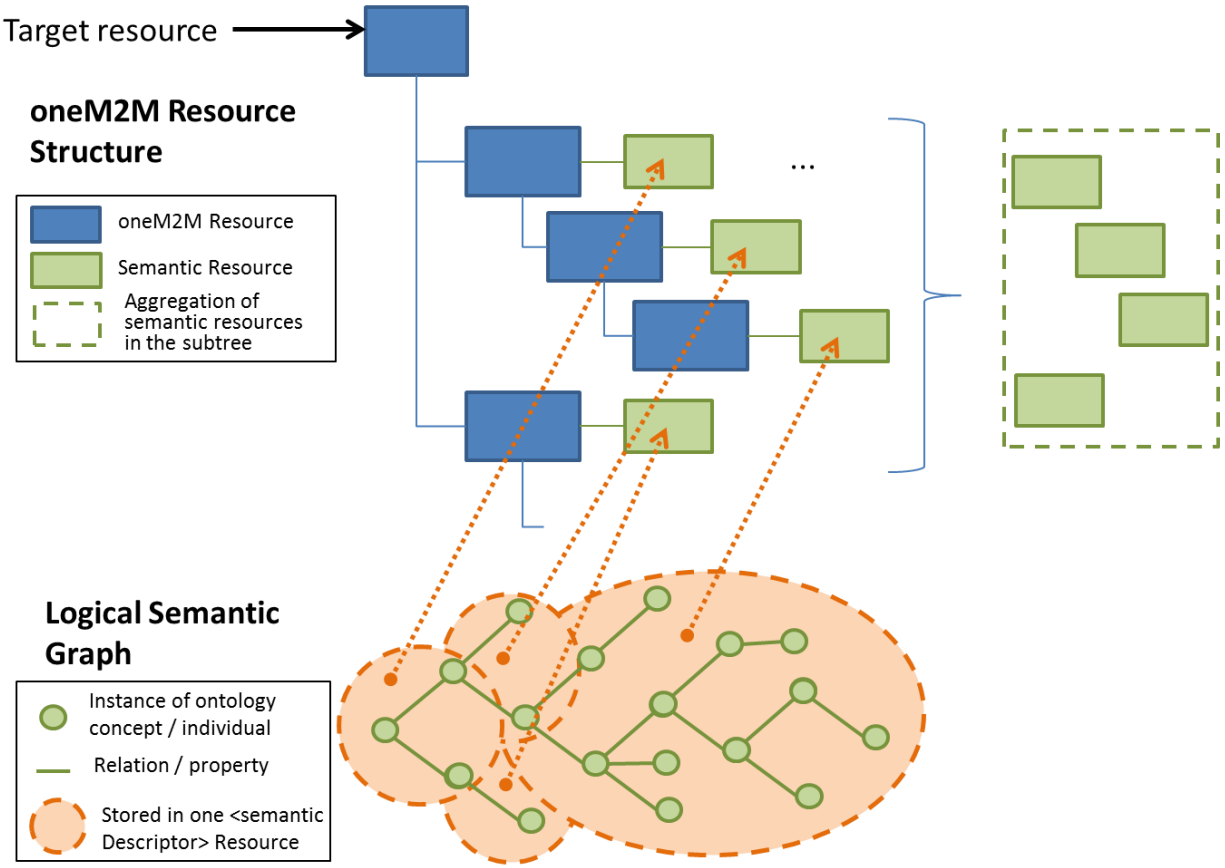

Figure 8.5.5.1-1: Implicitly scoped semantic queries

In Release 2 of oneM2M, there is no way of requesting the execution of a semantic query and directly getting the resulting semantic information back. There are two options of achieving these - either by extending the parameters of the general RETRIEVE requests and the corresponding results to enable specifying a semantic query - in SPARQL as it is done in the semantic filter for the discovery information, and the results, enabling the returning of SPARQL results instead of the usual resource content, or by introducing a speficic virtual resource (similar to the already existing <semanticFanOutPoint>) clause 8.5.5.2 presents the option proposing a new virtual resource type.

8.5.5.2 Semantic query based on virtual resource

Instead of executing a semantic query directly on the resource that is the root of the targeted subtree, a virtual resource can be introduced. The motivation is that, logically, it is not the content of the root resource and its children directly that are addressed, but the semantic information contained in all the semantic resources in the targeted subtree, including possible links to other resources. The result to be returned after executing the semantic query is then the subset of the total semantic information contained in the subtree and which is selected by the semantic query. This is depicted in figure 8.5.5.2-1.

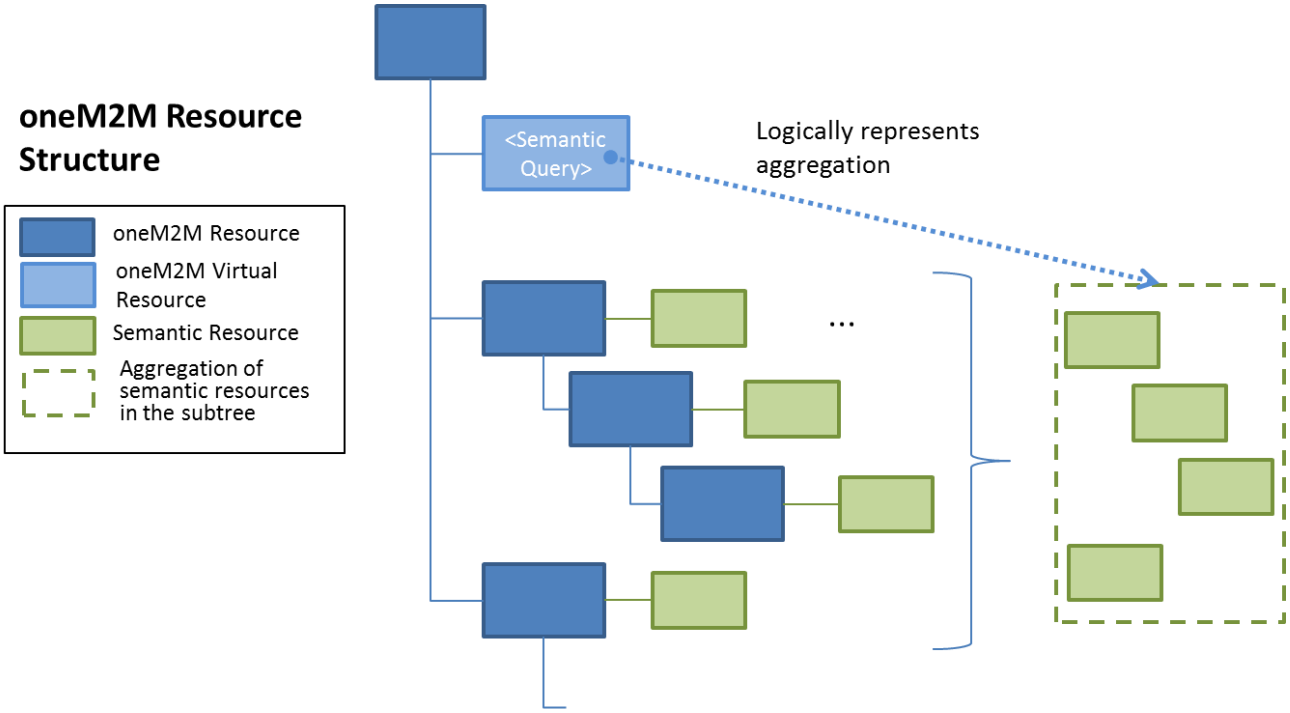

Figure 8.5.5.2-1 <semanticQuery> virtual resource

In Release 2 of oneM2M, the <semanticFanOutPoint> virtual resource was introduced as a child resource of the <group> resource that, given a semantic query in SPARQL, would first gather the semantic information from the semantic descriptors that are members of the group, then execute the SPARQL query on the aggregated result, and finally return the SPARQL result. The same general approach is proposed here, except that instead of having an explicit set of group members whose information is to be used as the basis for the query, the information is gathered from all semantic resources that are descendants of the parent resource or linked from there. This is the implicitly given scope of the query.

In the following the proposed Resource Type <semanticQuery> is described, in analogy to the <semanticFanOutPoint> already introduced:

- The <semanticQuery> resource is a virtual resource because it does not have a representation. It can be the child resource of any resource who can have semantic resources as child resources or whose descendants can have semantic resources as child resources.
- Whenever a semantic query request is sent to the <semanticQuery> resource, the host determines the semantic resources in the subtree rooted by its parent resources and retrieves the semantic information form them, taking into account explitly contained links, e.g. given in the *relatedSemantics* attribute. All semantic descriptors are accessed based on the respective access control policies.
- Once all of the related semantic information has been retrieved, it is added to the content on which the SPARQL request is being executed. The full content subject to the SPARQL request is provided to the SPARQL engine for processing.
- The <semanticQuery> resource uses the accessControlPolicyIDs attribute of the parent resource for access control policy validation.

The virtual resource <*semanticQuery*> is used for processing semantic query requests. As such, this virtual resource is the target of RETRIEVE requests only. The <*semanticQuery*> resource is created and deleted at the same time as the parent resource.

This RETRIEVE procedure is used for performing a semantic query using the semantic information from all the semantic resources that are descendants of the parent resource of the <semanticQuery> resource.

Table 8.5.5.2-1: *<semanticQuery>* RETRIEVE

***<semanticQuery>* RETRIEVE**	
Associated Reference Point	Mca, Mcc and Mcc'
Information in Request message	According to clause 10.1.2 of oneM2M TS-0001 [i.3]
Processing at Originator before sending Request	The Originator requests a semantic query to be performed using the semantic information from all the semantic resources that are descendants of the parent resource. The Originator is either an AE or CSE
Processing at Receiver	The Receiver does the following steps: • Check if the Originator has RETRIEVE privilege in the *<accessControlPolicy>* resource referenced by the *accessControlPolicyIDs* in the parent resource • Upon successful validation, obtain the URIs of all the semantic resources that are descendants of the parent resource and thosed linked through the *relatedSemantics* attribute • All semantic resources are accessed based on the respective access control policies • Once all the semantic information from the semantic resources has been retrieved, the SPARQL request is being executed on the combined content
Information in Response message	The result of the SPARQL request executed on the semantic information contained in/linked from the semantic resources that are descendants of the parent resource.
Processing at Originator after receiving Response	According to clause 10.1.2 oneM2M TS-0001 [i.3]
Exceptions	According to clause 10.1.2 oneM2M TS-0001 [i.3]

Adding a new resource type needs to be carefully considered, but alternatively if the support of semantic queries is added to the generic RETRIEVE operation, there are other issues to be taken into account. Semantic queries are rather specific RETRIEVE requests that require the possibility of adding a SPARQL request and being able to return SPARQL result content. A new parameter with the SPARQL request could only be used for RETRIEVE requests and in combination with requesting SPARQL result content (a new parameter value for result content). For neither the parameter containing the SPARQL request nor the parameter value requesting SPARQL result content it makes sense to use it on its own and they cannot be combined with any other parameters / parameter values. This restriction of use has to be carefully explained, making the already complex description of the general RETRIEVE request even more complex.

Also the general idea of REST is that (part of) the content of a resource is returned, possibly including descendant resources. In the case of using a semantic query as part of the general RETRIEVE this principle would be stretched as only some partial content of some descendant resources is going to be returned. With the virtual resource whose content is the aggregation of all relevant information, the RETRIEVE would simply return the part of the content that fits the SPARQL query, which is a kind of filter on the content of the <semanticQuery> resource, i.e. it is closer to the REST principle.

Finally, as the support of semantic queries on arbitrary subtrees may become expensive, the creation of <semanticQuery> resources could be restricted by policies, i.e. for a particular oneM2M instantiation it could be decided where to allow <semanticQuery> resources, providing flexibility.

A summary of the advantages and drawbacks of this approach are presented in table-8.5.5.2-2.

Table 8.5.5.2-2: *Advantages and drawbacks of <semanticQuery>resource*

Advantages	Drawbacks
General RETRIEVE operation does not have to be touched, as the needs for semantic queries are rather specific and need new request parameter(SPARQL) and result content (SPARQL result).	Introduces a new resource
Closer to REST as (subset) of resource content is returned	
Flexibility where to allow the creation of resource (e.g. policy based)	

8.5.5.3 Semantic query based on request parameter

Semantic query is a special request operation, which is different from but similar to the semantic resource discovery as described in the clause 8.5.2. According to oneM2M TS-0001, semantic resource discovery is triggered by a RETRIEVE operation containing a semanticsFilter condition tag included in Filter Criteria request parameter. Semantic query needs to be differentiated from semantic resource discovery, but using the same triggering principle; it is proposed that the following new parameter shall be included in a RETRIEVE request in order to retrieve a semantic query.

- **Semantic Query Indicator:** This parameter indicates that the request containing this parameter is to be processed as semantic query for semantically querying/retrieving the required information or knowledge based on some RDF triples.

When a CSE receives a RETRIEVE request which contains this new parameter, the CSE shall trigger a semantic query operation based on the SPARQL statement contained in the semanticsFilter condition tag.

Table 8.5.5.3-1 describes how this parameter relates to the request operation.

Table 8.5.5.3-1: New Request Message Parameters

Request message parameter		Operation				
		Create	**Retrieve**	**Update**	**Delete**	**Notify**
Optional	**Semantic Query Indicator** - A flag to indicate this request message is a semantic query request	N/A	O	N/A	N/A	N/A

Table 8.5.5.3-2 provides an assessment of this approach.

Table 8.5.5.3-2: Pros and Cons of Semantic Query based on Request Parameter

Pros	Cons
• No need to define any new resource/attribute or to create it prior to the query. • No impact on existing resources and/or attributes. • Follows the same principle for triggering a semantic resource discovery. • Query scope can be adjusted dynamically by changing the addressed resource. • No explicit query scope needs to be indicated before the operation, either by creating a new resource or otherwise.	• Extra overhead for processing a RETRIEVE operation due to the new request parameter (not high since the parameter is optional). • Need to specify the interaction with other request parameters applicable to RETRIEVE operations.

8.5.5.4 Conclusion

It was decided to enhance the request operation itself to support semantic queries. For this purpose, the existing *semanticsFilter* parameter is used to take the semantic query in SPARQL [i.14] and a new *semantic-content* result content was added to take the SPARQL query results. This is described in oneM2M TS-0001 [i.3], clause 8.1.2.

8.5.6 Explicitly scoped semantic queries

8.5.6.1 Using <semanticFanOutPoint> for semantic queries

The resource type semanticFanOutPoint has been introduced in Release 2 to support semantic discovery using semantic information that is contained in a set of different <semanticDescriptor> resources. The relevant <semanticDescriptor> resources in this case are the members of a group, which are represented as a <group> resource, which is the parent of the virtual <semanticFanOutPoint> resource. The proposal here is to extend the use of the <semanticFanOutPoint> resources to also support explicitly scoped semantic queries, i.e. the explicit scope of the query will be the semantic resources which are the members of the parent group. Also, we envision the introduction of additional semantic resources as mentioned in Section 8.5.1, so the description is generalized to cover semantic resources in general.

In the following minor modifications to the <semanticFanOutPoint> resource are proposed that enable this extension.

- The <semanticFanOutPoint> resource is a virtual resource because it does not have a representation. It is the child resource of a <group> resource ~~with members of type <semanticDescriptor>~~ whose members are semantic resources.

- Whenever a semantic ~~discovery~~ query request is sent to the <semanticFanOutPoint> resource the host uses the memberIDs attribute of the parent <group> resource to retrieve all the related ~~descriptors~~ semantic information. If there are descriptors stored on different CSEs, individual RETRIEVE requests are sent to each CSE for retrieving the semantic information from the external ~~descriptors~~ resources. All semantic ~~descriptors~~ resources are accessed based on the respective access control policies.

- Once all of the related semantic information~~<semanticDescriptor>(s) have~~ has been retrieved, ~~the content of each of the descriptor attributes it~~ is added to the content on which the SPARQL request is being executed. The full/enlarged content subject to the SPARQL request is provided to the SPARQL engine for processing.

- The <semanticFanOutPoint> resource uses membersAccessControlPolicyIDs attribute in the parent <group> resource for access control policy validation.

The virtual resource <*semanticFanOutPoint*> is used for processing semantic ~~discovery~~ query requests. As such, this virtual resource is the target of RETRIEVE requests only. The <*semanticFanOutPoint*> resource is created and deleted at the same time as the parent <*group*> resource.

This RETRIEVE procedure is used for performing a semantic discovery procedure using the descriptor content of all member ~~<*semanticDescriptor*>~~ semantic resources belonging to an existing <*group*> resource.

Table 8.5.6.1-1: *<semanticFanOutPoint>* RETRIEVE

***<semanticFanOutPoint>* RETRIEVE**	
Associated Reference Point	Mca, Mcc and Mcc'
Information in Request message	According to clause 10.1.2 oneM2M TS-0001 [i.3]
Processing at Originator before sending Request	The Originator requests a semantic ~~discovery~~ query request to be performed using the ~~content of the semantic descriptors~~ semantic information of all member resources belonging to an existing *<group>* resource. The Originator ~~may be~~ is either an AE or CSE
Processing at Receiver	The Receiver does the following steps: • Check if the Originator has RETRIEVE privilege in the *<accessControlPolicy>* resource referenced by the *membersAccessControlPolicyIDs* in the parent *<group>* resource. In the case *membersAccessControlPolicyIDs* is not provided, the access control policy defined for the parent *<group>* resource is be used • Upon successful validation, obtain the URIs of all the member ~~*<semanticDescriptor>*~~ semantic resources from the *memberIDs* attribute of the parent*<group>* resource • If there are ~~*<semanticDescriptor>*~~ semantic resources stored on different CSEs, individual RETRIEVE requests are sent to each CSE for retrieving the descriptors, otherwise the ~~*descriptor* attributes are~~ the semantic information is simply retrieved for all the ~~*<semanticDescriptor>*~~ semantic resources hosted locally. All semantic ~~descriptors~~ resources are accessed based on the respective access control policies • Once all of the related ~~*descriptor* attributes have~~ semantic information has been retrieved, the SPARQL request is being executed on the combined content
Information in Response message	The result of the SPARQL request executed on the combined content of the members' ~~*descriptors*~~ semantic information
Processing at Originator after receiving Response	According to clause 10.1.2 oneM2M TS-0001 [i.3]
Exceptions	According to clause 10.1.2 oneM2M TS-0001 [i.3]

8.6 Query scopes

The scoping of semantic queries has been discussed as part of the discussion on semantic queries in clause 8.5.

8.7 Semantic reasoning

Semantic reasoning has not been covered in Release 3.

8.8 Semantic mash-up

8.8.1 Introduction

8.8.1.1 Semantic mashup definition

Existing semantic resource discovery in oneM2M can help in discovering various IoT devices and their data. However, in many application scenarios, the discovered data needs to be further processed (e.g. integrated/orchestrated/combined) based on a certain application business logic. For example, users may just be interested in a metric called "weather comfortability index", which cannot be directly provided by physical sensors, and in fact can be calculated based on the original sensory data collected from multiple types of physical sensors (e.g. temperature and humility sensors).

In general, the above process is called "**Semantic Mashup**", which is defined as a process to discover and collect data from **more than one source** as inputs, conduct a kind of **business logic-related mashup function** over the collected data, and eventually generate meaningful **mashup results**. In particular, semantic mashup emphasizes on **leveraging semantic-related technologies** during the entire mashup process. For example, in the oneM2M context, an normal resource (e.g. a <AE> resource representing a temperature sensor) may be annotated by semantic descriptions and then they could be discovered and identified as a potential data source for a specific mashup application through the semantic resource discovery.

The above definition also indicates a fact that a complete semantic mashup process may involve multiple stages and multiple entities for each stage, which will be illustrated and discussed in details through the following sections.

8.8.1.2 Semantic mashup example: smart parking application

Smart parking is one of the major applications in a smart city. In the smart city, each parking spot (e.g. inside parking buildings and any street parking spot) is equipped with a parking sensor. The parking sensor provides real-time parking spot information including the real-time status about its associated parking spot (i.e. occupied or not), the geographic location of the associated parking spot, as well as the real-time parking rate of the associated parking spot. All those information can be described in a semantic form, such as RDF triples. As shown in figure 8.8.1.2-1, there are some parking buildings (whose parking sensors are registered to Server-A) as well as street parking spots (whose parking sensors are registered to Server-B) around Building-A.

For a given user/client who wants to find a suitable parking spot near her/his destination (e.g. Building-A), she/he may send a parking request (indicating e.g. her/his destination and/or parking preference, etc.) to Server-C, which provides real-time parking assistance. In order to process this parking request, a mashup process will be triggered in Server-C. For example, by leveraging semantic-related technologies Server-C will first collect the needed information from multiple places, which include:

1) Real-time parking spot information from Server-A for the parking buildings;

2) Real-time parking spot information from Server B for the street parking spots.

Then, Server-C will process and mashup all these collected information in order to find the most suitable parking spot according to a certain mashup function, which could have been provisioned to Server-C. For instance, the mashup function can define the most suitable parking spot as the one having the minimum parking rate or having the most convenient location (e.g. having the shortest walking distance to the elevator). Finally, the mashup result (i.e. the most suitable parking spot) will be returned to the user.

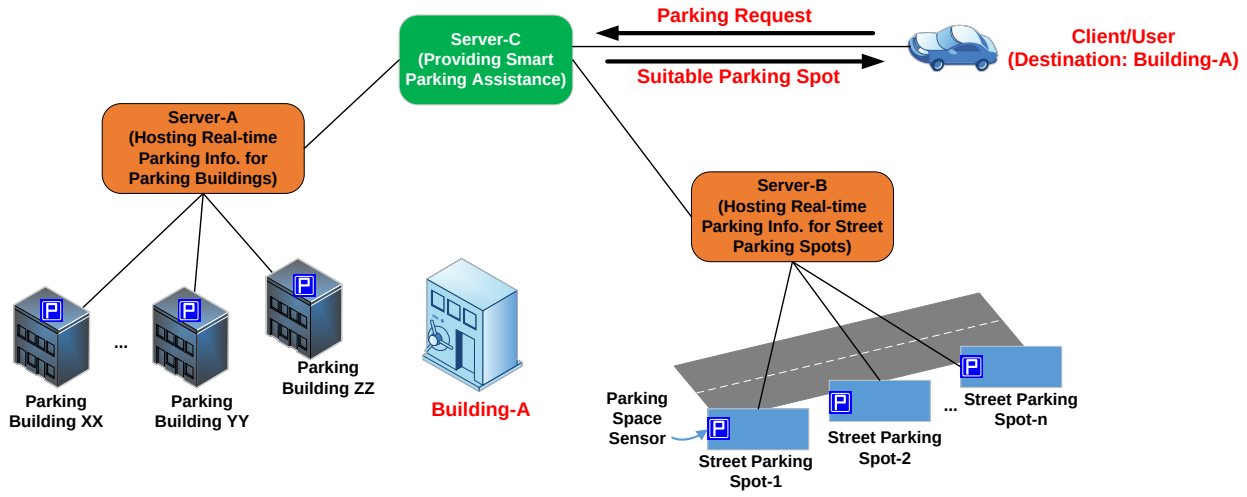

Figure 8.8.1.2-1: Smart Parking Assistance Realized Through Semantic Mashup

8.8.2 Problem statement

8.8.2.1 Entities involved in sematic mashup

From the above smart parking example, it can be seen that multiple entities are involved in the mashup process, which include:

- **Mashup Requestor (MR):** The entity which initiates a mashup request to Semantic Mashup Function (as introduced next) for a certain need. For example, a client/user looking for the suitable parking spot is the mashup requestor in the smart parking use case. In the context of oneM2M, an AE or a CSE can be an MR.

- **Resource Host (RH):** The entity which hosts data source(s) for a given mashup process. For example, in the smart parking use case, Server-A and Server-B are RHs, which host data sources in terms of many parking sensors. In the context of oneM2M, a data source is typically represented by a oneM2M resource (e.g. a temperature <AE> resource) and a RH will be a CSE that hosts oneM2M resources.

- **Semantic Mashup Function (SMF):** The entity which is responsible for collecting the data inputs from data sources hosted on RHs and mashing them up to generate the mashup result based on a certain business logic. For example, in the smart parking use case, Server-C provides an SMF and one of mashup applications run by the SMF is the smart parking assistance. In the context of oneM2M, SMF is a Common Service Function.

8.8.2.2 Technical analysis of semantic mashup

It is worth noting that there are many-to-many relationships between different MRs, RHs and SMFs in reality. Typically, a RH may provide the same data inputs for different mashup applications. For example, the same outdoor temperature sensors can not only provide inputs for a weather reporting mashup application, but also for a travel planning mashup application. Similarly, a single SMF may provide and support various mashup applications. For example, in the smart parking use case, the SMF hosted by Server-C may also support the shopping guidance mashup application besides the smart parking assistance; accordingly, an MR may resort to Server-C in order to not only find the most suitable parking spot near Building-A (supported by smart parking assistance mashup application) but also get the most recent coupons provided by the shops in Building-A (supported by the shopping guidance mashup application). In addition, for a given mashup application, different MRs may trigger individual mashup processes/instances to be conducted by SMF. For example, in the smart parking use case, different users as MRs may resort to Server-C for finding their respective parking spots given their different destinations, and each of their requests may trigger an individual mashup process for the same smart parking assistance mashup application.

Considering the multiple entities involved in a mashup process as well as their potential many-to-many relationships as discussed above, a common SMF needs to be included in a service layer in order to provide **generic** semantic mashup support for various application scenarios. In particular, in the context of oneM2M service layer, this SMF could be a CSF since data inputs mainly come from oneM2M resources and MRs mainly refer to service layer entities such as AEs/CSEs. In comparison, when there is no such a common SMF in the system, alternative ways to realize semantic mashup likely will be inefficient/non-optimal. For example, all mashup-related tasks could be handled by MRs themselves (i.e. MRs need to fully understand business logics, identify the qualified data sources, collect data inputs, and mashup those data inputs in order to derive the mashup result), which may have the following potential issues:

- An MR has to collect data inputs from original resources. However, in most of cases, an MR does not really care about original resources and it is only interested in the mashup result. Furthermore, the MR needs to have various domain knowledge if data inputs to be collected are from different domains. Accordingly, all the mashup-related processing needs to be handled by the MR itself, which increases the implementation complexity for the MR.

- Some resources (as data sources) hosted by RHs may be sensitive due to various access control policies. In other words, certain resources may not be exposable to an MR or not allowed to be retrieved directly by an MR. In comparison, a common SMF may have more privileges for collecting and accessing data inputs.

- Different MRs may conduct mashup operations individually without any collaboration, which may lead to considerable/unnecessary overhead (e.g. repetitive data retrieval from the same data source). In comparison, a common SMF could retrieve and feed a data input to multiple mashup processes if they need the same data input(s), which can significantly reduce the related overhead.

8.8.3 Semantic Mashup Function (SMF)

8.8.3.1 High-level architecture

This section describes the high-level architecture of Semantic Mashup Function (SMF) and the high-level operations of SMF that are needed during the execution of a semantic mashup process. In the meantime, this section also introduces various semantic mashup paradigms in the real world and the details on how this SMF can be tailored in order to support those different paradigms, which demonstrates the improved flexibility, reusability, and system efficiency of SMF.

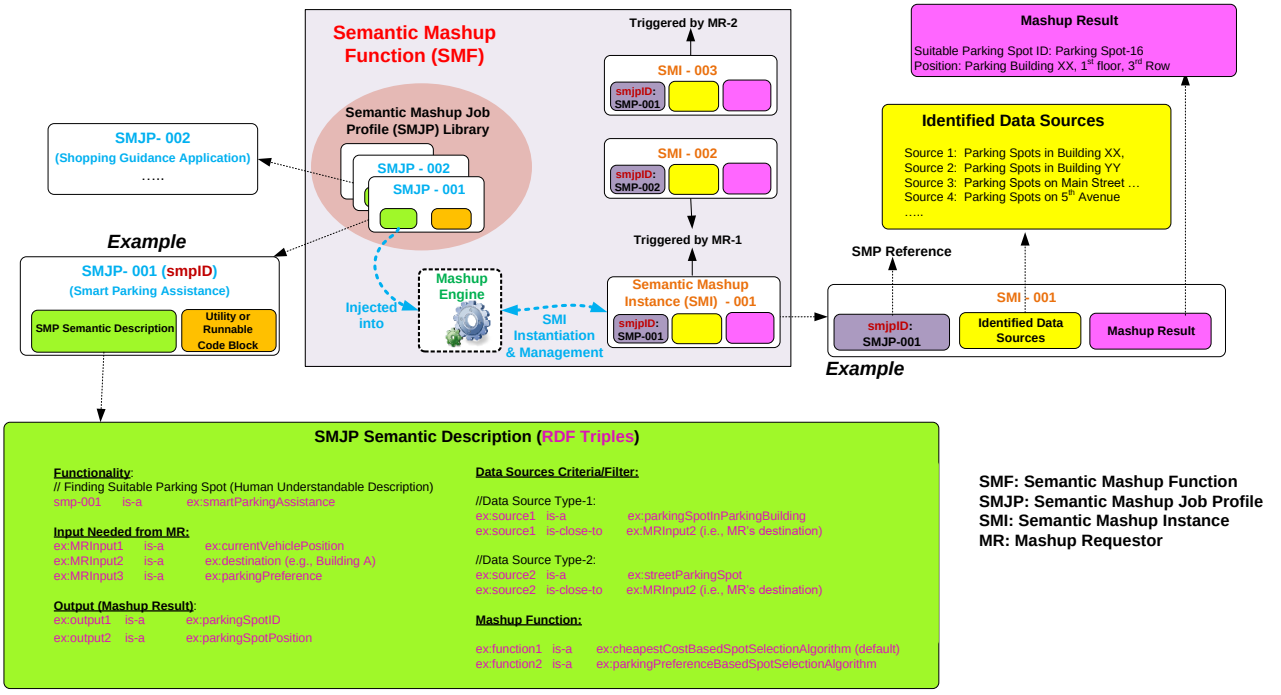

Figure 8.8.3.1-1: High-level architecture of Semantic Mashup Function

The high-level architecture of an SMF is shown in figure 8.8.3.1-1, In general, the SMF is designed in a modular way and mainly includes the following components:

- **Semantic Mashup Job Profile (SMJP):** Each specific semantic mashup application has a corresponding SMJP, which not only provides functionality/interaction details for external entities to discover (e.g. MRs), but also defines the internal working details regarding how to realize this mashup application (e.g. the criteria of how to select the qualified data sources as well as the definition of mashup function). To be more specifically, an SMJP includes the following parts:

  - **SMJP Semantic Description:** In general, an MR first needs to discover a desired SMJP based on its need before sending a request to an SMF for triggering a real mashup process for this SMJP. Since there could be various mashup applications supported by an SMF, the mashup applications' corresponding SMJPs could be stored in a repository called SMJP Library for MRs to browse and discover. An MR can evaluate the semantic descriptions (e.g. RDF triples) of an SMJP in order to identify whether an SMJP meets its need.

    The semantic descriptions can include the information regarding to what is the functionality that can be realized by executing this SMJP, how to interact with or run this SMJP, e.g. what inputs are needed from MR side and what is the expected mashup result (as output) of this SMJP, etc. Taking the example shown in figure 8.8.3.1-1, the semantic descriptions of the SMJP (which has the smjpID of SMJP-001) indicate that:

    1) this is a "smart parking assistance mashup application";

    2) an MR needs to provide its current location, destination and/or parking preference as user inputs in order to trigger a real mashup process for applying this SMJP; and

3) the mashup result will be a suitable parking spot (including the spot's ID/name and location).

The semantic descriptions can also specifies the "internal" operation details regarding to how to execute and realize a semantic mashup process, which defines the following aspects:

- **Data Sources Criteria:** The potential data sources for an SMJP are described through some criteria filter(s), which defines what types of data sources are eligible or qualified to provide data inputs for this specific mashup application. Taking the smart parking example shown in figure 8.8.3.1-1, Two types of data sources are desired, i.e. the parking spots in any parking buildings near the destination, and the parking spots in any street parking near the destination. Note that, an SMJP only defines the "criteria" for identifying the qualified data sources, which means that it does not refer to any specific data sources. Typically, when a real mashup process of an SMJP is triggered in SMF, SMF will then identify the specific qualified data sources according to the criteria as defined in the SMJP (the details will be discussed later).

- **Mashup Function:** For a given semantic mashup process, after the SMF collects data inputs from the qualified data sources, the next step is to mashup those inputs and derive the mashup result, which is described by the mashup function. Taking the smart parking example shown in figure 8.8.3.1-1, after identifying all the parking spots near the destination, various spot selection algorithms can be used as mashup functions in order to determine the best suitable one. For example, the default one is to find the parking spot having the minimum cost.

- **Utility or Runnable Code Block:** This part mainly stores the related utilities or code blocks for realizing the SMJP. For example, it could include specific mathematical formulas to be used by the parking spot selection algorithms, and the runnable code blocks for realizing the mashup functions as mentioned earlier. Note that, this part is optional in the sense that the implementations of SMJP could be stored separately or just internally hosted by SMF.

- **Semantic Mashup Instance (SMI):** Once an MR identifies a desired SMJP (which can be analogous to a "job description", but not a real job), it can ask SMF to initialize a real mashup process (which corresponds to a "working instance" of this SMJP and is called Semantic Mashup Instance, or SMI). In order to do so, the SMF will inject the corresponding SMJP into the Mashup Engine of SMF for the SMI instantiation, during which the engine may be involved in:

  - Identifying the qualified data sources according to the data source criteria as defined in the SMJP.

  - Collecting data inputs from those identified data sources.

  - Mashing up the collected inputs by applying mashup functions as defined in the SMJP, and finally deriving the mashup result.

- Typically, an SMI has the following three parts as shown in figure 8.8.3.1-1:

  1) the smjpID, which is the reference to the applied SMJP.

  2) The identified data sources, which indicate where the data inputs can be collected and optionally can also contain the data inputs collected from those data sources; To consider an SMI as a whole, the identified data sources in a SMI can be regarded as the "**Mashup Members**" of this SMI and accordingly the data source criteria as defined in SMJP is in fact a member filter for determining how to select qualified mashup members for a given SMI. The term "data source" and "mashup member" will be used interchangeably in the rest of sections.

  3) The mashup result derived by this SMI.

- Taking the smart parking example shown in figure 8.8.3.1-1, SMI-1 is an SMI for smart parking assistance mashup application, which is created by MR-1. As can be seen, based on MR-1's inputs (e.g. current location and destination, etc.), the parking spots in parking buildings, i.e. XX and YY, as well as street parking spots on Main Street and 5th Avenue are identified as "qualified data sources". By applying the parking slot selection algorithm, the final mashup result in terms of the suitable parking spot is parking spot -16, which is located in 1st floor, 3rd row of the parking building XX. It can also be seen that, besides the smart parking assistance mashup application, MR-1 may need a shopping guidance mashup result as well, which will trigger another SMI-2 referring to a different SMJP (i.e. SMJP-002, which specifies shopping guidance mashup application). In addition, another MR-2 (having a different destination) may also ask the SMF for its own parking need, which will trigger another SMI-3 (note that, SMI-3 applies the same SMJP as SMI-1, i.e. SMJP-001).

8.8.3.2 High-level operations

With the high-level architecture of an SMF as introduced in clause 8.8.3.2-1, the next question is how those components should work together and how an SMF should interact with other entities, such as RHs (providing potential data sources), and MRs (having various semantic mashup needs). It is worth noting that an SMF may involve in different tasks/operations for realizing a complete semantic mashup process. This clause is to introduce those major SMF operations, and particularly discuss the possible solutions for implementing those operations in the context of oneM2M.

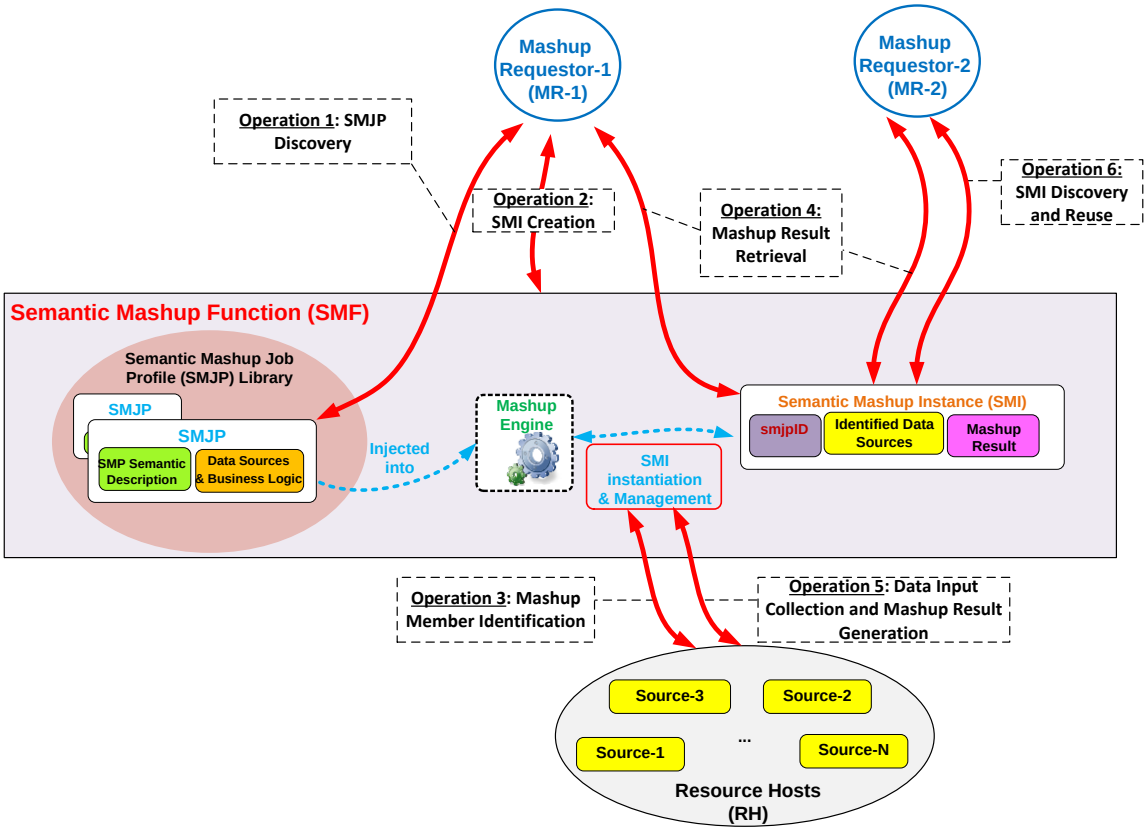

Figure 8.8.3.2-1: High-level operations for Semantic Mashup Function

The high-level SMF operations are shown in figure 8.8.3.2-1, which is discussed in details as follows:

- **Operation 1 - SMJP Discovery:** This process is needed when an MR (e.g. MR-1 in figure 8.8.3.2-1) tries to discover a desired SMJP for its need. Since operations in oneM2M context is resource-oriented, an SMJP can be exposed as a oneM2M resource (e.g. <SMJP>) and the semantic descriptions of the SMJP can be put into the <semanticDescriptor> child resource of <SMJP>. Accordingly, the SMJP Discovery can be implemented by leveraging the existing semantic resource discovery mechanism.

- **Operation 2 - SMI Creation:** This process is needed when an MR already identified a desired SMJP, but there is no corresponding SMI available for use. To implement this operation, an MR can send an SMI creation request targeted to the CSE hosting SMF in order to instantiate a new SMI for the desired SMJP. Alternatively, the SMF can also create a new SMI by itself instead of being triggered by the SMI creation request from MR.

- **Operation 3 - Mashup Member Identification:** This process is needed when an SMF tries to identify the qualified mashup members (i.e. data sources) for a given SMI, by referring to the criteria as defined in the corresponding SMJP of this SMI. Since in the oneM2M context, data sources (such as sensors) are normally represented as oneM2M resources hosted by RHs, this operation can also be implemented through the existing semantic resource discovery mechanism.

- **Operation 4 - Mashup Result Retrieval:** This process is needed when an MR tries to retrieve the mashup result from a specific SMI. It is worth noting that for a given SMI, it may involve in multiple rounds for mashup result generation especially when the mashup result needs to be refreshed periodically. For example, an SMI of a weather reporting mashup application for a given area (e.g. New York City Area) may need to re-

calculate the mashup result every certain minutes due to real-time weather changes. In order to do so, this SMI first needs to re-collect new data inputs from its identified mashup members (note that, the SMI is not necessary to re-identify the qualified mashup members for every round of mashup result generation if the existing identified mashup members are still qualified). Accordingly, there could be several alternative mechaniSMF for the mashup result generation, e.g.:

- **Option 1:** The SMF proactively and periodically runs the mashup result generation and as long as the newly mashup result becomes available, it will be exposed as a oneM2M resource for MRs to access. In this approach, Operation 4 is only involved with a normal oneM2M resource retrieval. The benefit of this approach is that the MRs can obtain the mashup results with minimum delay but the overhead is that the SMF needs to proactively and periodically run the mashup result generation no matter whether there is a need.

- **Option 2:** The SMF will work in a reactive way to generate the mashup result. In this approach, only after receiving a mashup result retrieval request from an MR, SMF will just start to re-collect new data inputs and re-calculate the mashup result (which corresponds to Operation 5 as introduced next). In other words, Operation 4 in this approach will further trigger Operation 5. The benefit of this approach is that SMF will work in an on-demand way, which may reduce overhead as compared to Option 1. However, the downside is that it leads to longer waiting time for an MR before the up-to-date mashup result becomes available because data re-collection and mashup result generation will not be triggered until the SMF receives a request from the MR.

- **Operation 5 - Data Input Collection and Mashup Result Generation:** This process is needed when an SMF tries to generate a mashup result for a given SMI. Note that, Operation 3 focuses on how to identify the mashup members while Operation 5 focuses on how to collect data inputs from those identified/qualified mashup members, which can be implemented through the normal resource retrieval. In addition, the working mechanism used for Operation 4 as mentioned above will affect how Operation 5 is used by the SMF.

- **Operation 6 - SMI Discovery and Re-use:** As mentioned earlier, in some scenarios, an SMI can be re-used and shared among different MRs. For example, the same SMI of a weather reporting mashup application for New York City Area can be shared by different users asking weather information for this area. Accordingly, this process is needed when an MR (e.g. MR-2 in figure 8.8.3.2-1) tries to discover whether there is already an available/desired SMI ready for use. Similarly, in the oneM2M context, a given SMI can also be exposed as a resource (i.e. each SMI has a corresponding **Semantic Mashup Resource,** or called <SMR>), in which the essential parts for describing this SMI can be included, such as the referred smjpID, the identified mashup member list, and/or the mashup result yielded by this SMI. Accordingly, taking the example shown in figure 8.8.3.2-1, MR-2 can find a desired SMI (which is available and was created by MR-1) through an SMI discovery process, which can also be realized by using the existing semantic resource discovery mechanism. After that, MR-2 can ask the discovered SMI for the mashup result, and the SMI will directly leverage the already-identified mashup members. This approach leads to less processing overhead, since MR-2 does not have to require the SMF to generate a new SMI (therefore Operation 2 and 3 are not needed).

8.8.3.3 Functional paradigms

It is worth noting that although the smart parking mashup application was used in the previous sections for illustrating the details of an SMF, the smart parking assistance is just one of the possible semantic mashup paradigms. This section is to explore the two major semantic mashup paradigms in real world, and to show how an SMF can be tailored in order to well support each of those paradigms.

Paradigm 1: Short-lived Semantic Mashup: Short-lived semantic mashup application basically refers to as the application in which a mashup process will be completed once the corresponding mashup result is produced. The previous smart parking assistance mashup application falls into this paradigm. For example, once a user gets a suitable parking spot from SMF, the whole mashup process is completed. In the meantime, since different users may be towards different destinations and have different parking preferences, the same SMI for serving user A may not be re-used for serving user B. Accordingly, every time when a user needs to find a suitable parking spot for its own use, the SMF may probably need to instantiate a new SMI. With the proposed SMF in clauses 8.8.3.1 and 8.8.3.2, the Operations 1-5 could be adopted as the solution to Paradigm 1.

Paradigm 2: Long-lived Semantic Mashup: Long-lived semantic mashup application basically refers to the application in which a mashup process will last for a relatively long time, during which the corresponding SMI may generate the mashup result for multiple times and the SMI can also be re-used/shared among different MRs. The previous weather reporting mashup application falls into this paradigm. For example, once an SMI has been created for

reporting weather information of New York City Area, the mashup result of this SMI will be refreshed by the SMF periodically due to real-time weather changes. In the meantime, the same SMI can be shared and re-used in this paradigm by different MRs, e.g. the users from New York City Area can ask the same SMI for the real-time weather information in the area. With the SMF, the Operations 1-6 could be adopted as the solution to Paradigm 2.

8.8.4 Semantic Mashup Procedure Details

8.8.4.0 Introduction

The clauses 8.8.4.1, 8.8.4.2, 8.8.4.3 and 8.8.4.4 describe procedure details for the mashup operations as listed in the clause 8.8.3.2:

- The clause 8.8.4.1 describes the procedure for semantic mashup job profile discovery and retrieval (i.e. corresponding to Operation 1 in figure 8.8.3.2-1).
- The clause 8.8.4.2 describes the procedure for semantic mashup instance creation (i.e. corresponding to Operation 2 and Operation 3 in figure 8.8.3.2-1).
- The clause 8.8.4.3 describes the procedure for semantic mashup result retrieval (i.e. corresponding to Operation 4 and Operation 5 in figure 8.8.3.2-1).
- The clause 8.8.4.4 describes the procedure for semantic mashup instance discovery (i.e. corresponding to Operation 6 in figure 8.8.3.2-1).

8.8.4.1 Semantic Mashup Job Profile Discovery and Retrieval

As described in the clause 8.8.3.2, an MR first discovers a desired SMJP from SMJP library which could be hosted by a CSE (i.e. Operation 1 in figure 8.8.3.2-1). Figure 8.8.4.1-1 illustrates the procedure for SMJP discovery and retrieval.

Step 1: The MR Requestor sends an SMJP discovery request to the SMF. The SMJP discovery request contains a parameter to indicate and specify the type of desired SMJPs. Remember that since operations in oneM2M context is resource-oriented, an SMJP can be exposed as an oneM2M resource and the semantic descriptions of the SMJP can be put into the <semanticDescriptor> child resource of the resource representing a SMJP. Accordingly, this SMJP discovery request can be implemented by leveraging the existing semantic resource discovery mechanism; as such, *semanticsFilter* condition tag of the *Filter Criteria* parameter in an oneM2M request message can be used to indicate the type of desired SMJPs.

Step 2: The SMF sends the response to the MR containing the *smjpID*s of the qualified SMJPs, which satisfies the criteria described by the parameter in Step 1.

Step 3: After receiving the *smjpID*s of the qualified SMPs, the MR can send an SMJP retrieval request to retrieve one of the discovered SMJPs in Step 2. This request contains the *smjpID* of the SMJPs to be retrieved.

Step 4: After receiving the SMJP retrieval request, the SMF sends a response, which contains the representations of the corresponding SMJP.

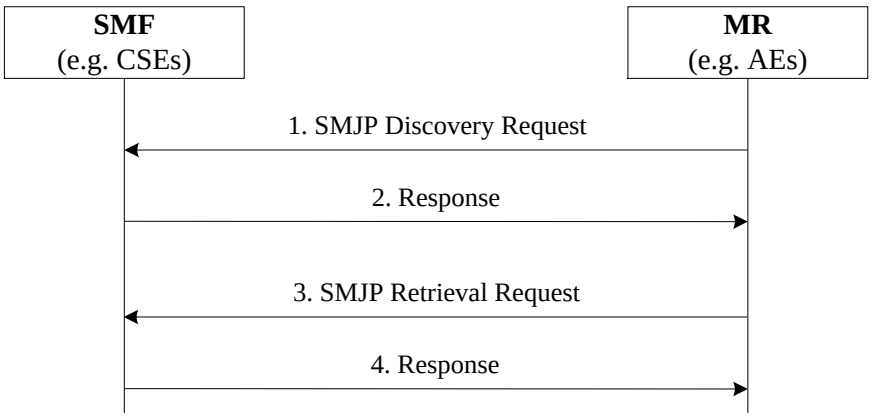

Figure 8.8.4.1-1: SMJP Discovery and Retrieval

8.8.4.2 Semantic Mashup Instance Creation

This procedure corresponds to Operation 2 and Operation 3 as described in the clause 8.8.3.2 and in figure 8.8.3.2-1. The steps illustrated in figure 8.8.4.2-1 are used by the MR to create an SMI (corresponding to Operation 2), during which mashup members will be discovered and identified (corresponding to Operation 3).

Step 1: The MR sends an SMI creation request to the SMF. This request contains the URI of the applied SMJP and some input parameters required to create an SMI based on the applied SMJP. In one example, the value of input parameters can be contained in the form of RDF triples. In addition, this creation request may also include a parameter to indicate the way for generating mashup result; for example, mashup result could be generated periodically (corresponding to Paradigm 2- Long-lived Semantic Mashup as described in the clause 8.8.3.3) or only one time when the SMI is created (corresponding to Paradigm 1- Short-lived Semantic Mashup as described in the clause 8.8.3.3).

Step 2: The SMF processes the received SMI creation request. Then, the SMF will conduct Step 3 to first find qualified original resources as mashup members. Here, the qualified resources are defined as the original resources, which meet the data source criteria of the corresponding SMJP.

Step 3: The SMF performs semantic resource discovery to find suitable original resources as mashup members of the SMI to be created in Step 4.

Step 4: The SMF creates a new SMI. Mashup members of this new SMI will be selected from the qualified resources from Step 3. In addition, this SMI will be associated with the corresponding SMJP as indicated in Step 1.

Step 5: After successfully creating the requested SMI, the SMF returns the URI of the SMI to the MR so that the MR can retrieve the content of the SMI at a later time.

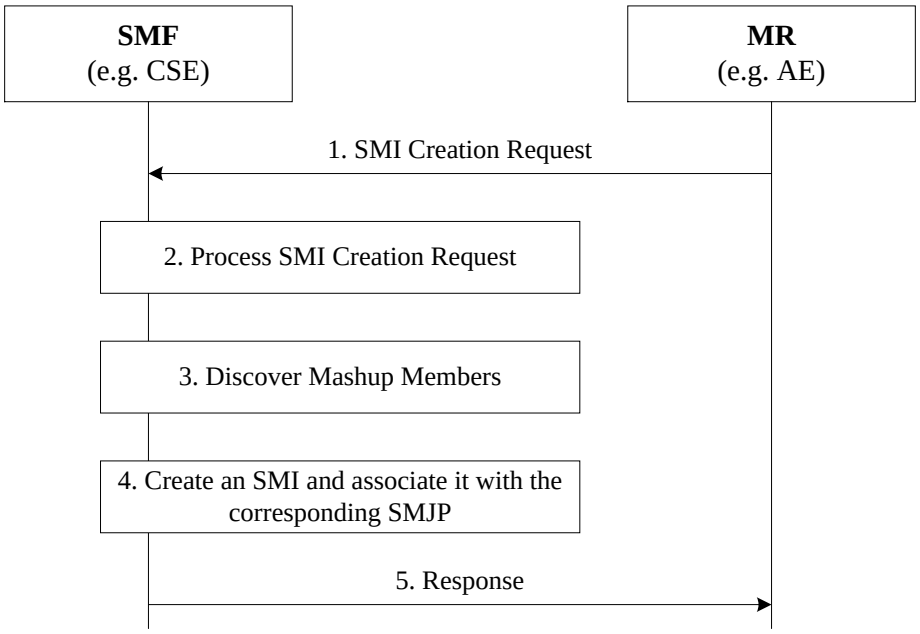

Figure 8.8.4.2-1: SMI Creation

8.8.4.3 Semantic Mashup Result Retrieval

This procedure corresponds to Operation 4 and Operation 5 as described in the clause 8.8.3.2 and in figure 8.8.3.2-1. The steps illustrated in figure 8.8.4.3-1 are used by the MR to retrieve mashup results from a created SMI (corresponding to Operation 4), during which data inputs from mashup members may be collected and mashup results may be re-generated (corresponding to Operation 5).

Step 1: Assume the MR knows the URI of the SMI resource. In order to retrieve its mashup result, the MR send a mashup result retrieval request to the SMF. This request targets the existing SMI (or the mashup result child resource of this SMI resource if there exists such a child resource to store the mashup result of this SMI).

Step 2: The SMF receives the request and locates the targeted SMI resource, from which the SMF knows how the mashup result should be generated for this SMI resource. For example, if the SMI resource indicates that "the mashup result should be re-generated when an MR requests", the SMF needs to retrieve the latest representation of each mashup member resources using Step 3 and Step 4 as described next; otherwise, if the SMI resource indicates that "the mashup result is re-generated periodically by the SMF itself", the SMF will skip Steps 3-5 and simply return the existing mashup result to the MR in Step 6 since the current result is up-to-date.

Step 3: The SMF retrieves the latest data value of each mashup member resource of the targeted SMI resource in Step1.

Step 4: The RH responds with the representations of the corresponding mashup member resource to the SMF.

Step 5: After receiving the representations of all mashup members, the SMF executes the mashup function of corresponding SMJP which is associated with the targeted SMI resource. The SMF stores the calculated mashup result in the SMI (e.g. as its attribute or child resource).

Step 6: The SMF sends the mashup result to the MR as a response to Step1.

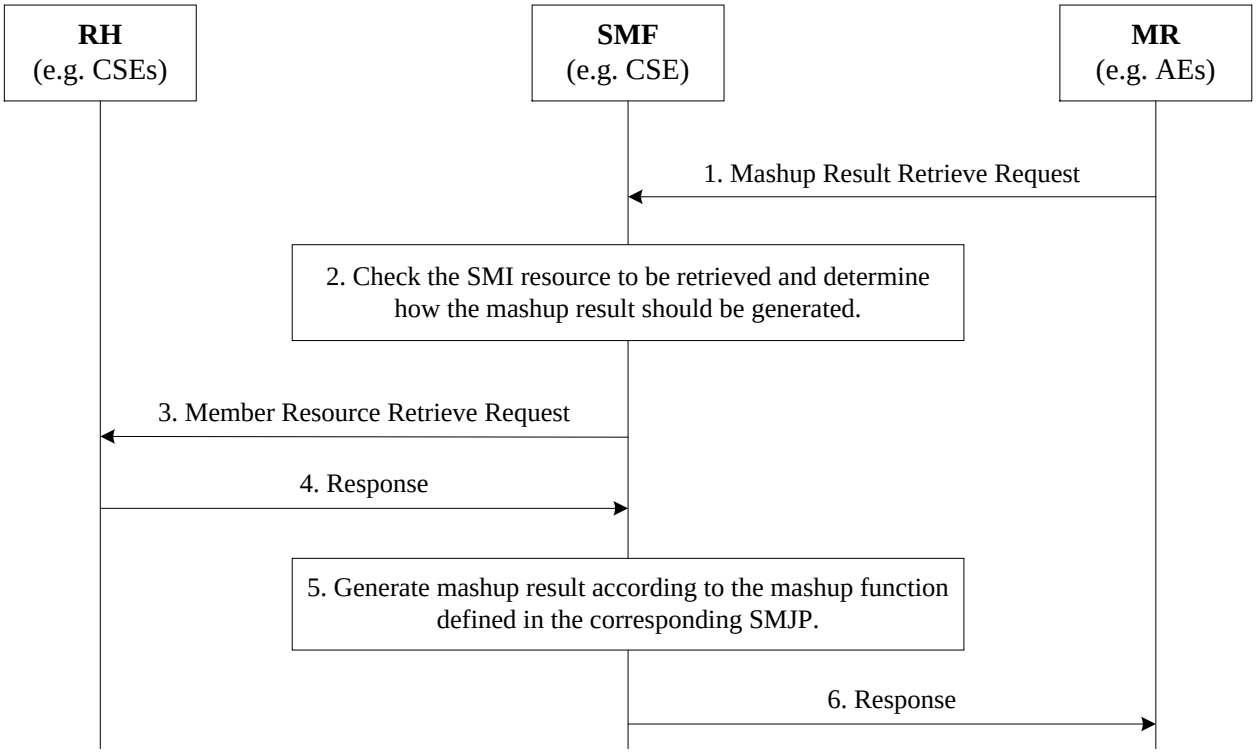

Figure 8.8.4.3-1: Semantic Mashup Result Generation and Retrieval

8.8.4.4 Semantic Mashup Instance Discovery

The created SMI can be discovered by other MRs, who are interested in the service (i.e. mashup result) provided by the SMI. Thus, existing SMI can be reused by other MRs to improve the system efficiency. This corresponds to Operation 6 as described in the clause 8.8.3.2 and in figure 8.8.3.2-1. Once an MR discovers an existing SMI, it can reuse this SMI without creating a new SMI; for example, it can directly employ the procedure in the clause 8.8.4.3 to retrieve mashup result.

The following steps are illustrated in figure 8.8.4.4-1 for SMI discovery.

Step 1: An MR, who is interested in a specific SMI, sends an SMI discovery request to an SMF. This discovery request includes a parameter to indicate the types of desired SMIs to be discovered. This request can be implemented by leveraging the existing semantic resource discovery mechanism since SMIs normally also have its own resource representation; as such, the *semanticsFilter* condition tag of the *Filter Criteria* parameter in an oneM2M request message can be used to indicate the types of desired SMIs.

Step 2: After receiving the SMI discovery request, the SMF may build a local semantic graph store, search the RDF triples over the local semantic graph store, and return the list of existing SMI resources which match the types of SMIs as indicated in Step 1. If there is a central semantic graph store at the service layer, the SMF will contact the central semantic graph store, which can search the RDF triples and return the list of the matched SMI resources' URIs to the SMF. Consequently, the SMF sends the list of the matched SMI resources' URIs to the MR.

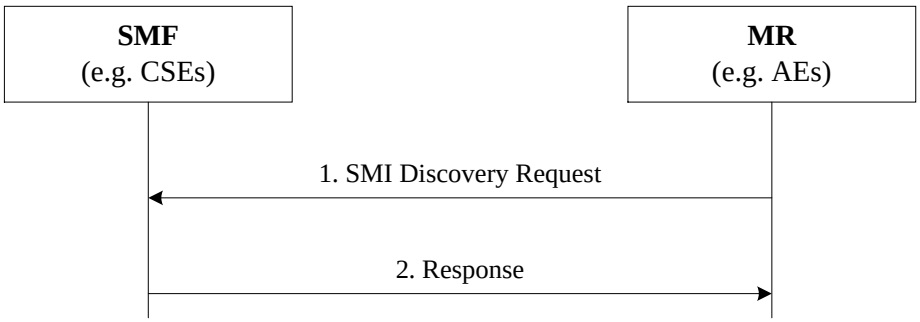

Figure 8.8.4.4-1: SMI Discovery

8.9 Semantics-based data analytics

Semantics-based data analytics has not been covered in Release 3.

8.10 Ontology management

Ontology management has been introduced in Release 3 with the <*ontologyRepository*> and <*ontology*> resources that have been introduced in clauses 7.3.5 and 7.3.6 respectively.

8.11 Semantic Access Control

8.11.1 Synchronizing ACP information between the Resource Tree and the Semantic Graph Store

As described earlier, the Direct Access Control of Semantic Graph Store approach requires synchronization between <accessControlPolicy> resources at the Hosting CSE and ACP Triples at the Semantic Graph Store. In order to maintain such synchronization, the Hosting CSE needs to perform the following tasks:

- When a new <accessControlPolicy> resource is created, the Hosting CSE will generate new ACP Triples according to the ACP ontology and stores these new ACP Triples in the Semantic Graph Store
- When the *privileges* attribute of an existing <accessControlPolicy> resource is updated, the Hosting CSE will also generate new ACP Triples and update corresponding old ACP Triples at the Semantic Graph Store accordingly
- When an existing <accessControlPolicy> resource is deleted, the Hosting CSE will need to remove the corresponding ACP Triples at the Semantic Graph Store.

Figure 8.11.1-1 illustrates the procedure for creating ACP Triples in Semantic Graph Store, which is triggered when an Originator requests to create an <accessControlPolicy> resource at the Hosting CSE.

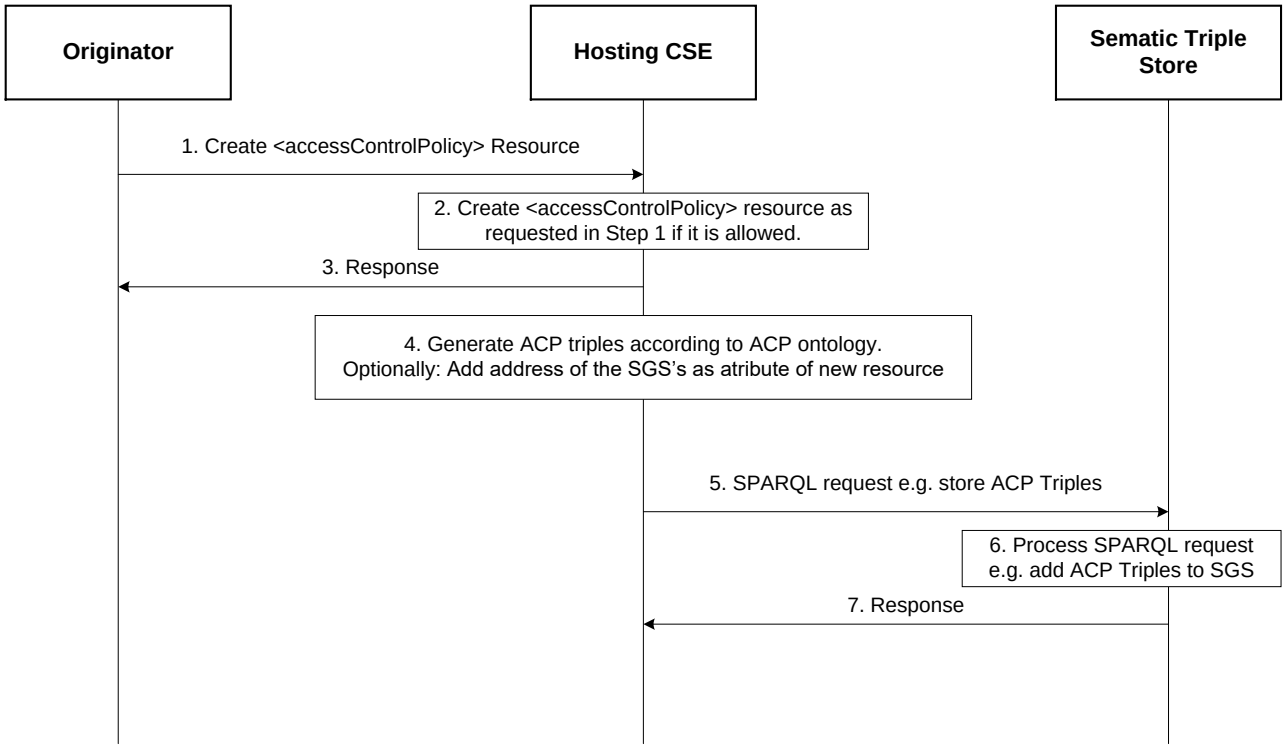

Figure 8.11.1-1: Procedure for creating ACP triples in the Semantic Graph Store

- **Step 1:** The Originator sends a request to create an <accessControlPolicy> resource to the Hosting CSE. This message contains the representation of <accessControlPolicy> to be created (e.g. the value of privileges attribute).

- **Step 2:** The Hosting CSE receives the request in Step 1 and, subject the Originator access rights verification, will create the requested <accessControlPolicy> resource.

EXAMPLE 1: Assume <acp1> to be the newly created ACP resource and its URI "acp1URI". Assuming <acp1> has one access control rule (e.g. acr11) and the URI of the corresponding privileges attribute is "acr11URI". For exemplification, assume also that acr11 allows an AE ("AE-ID-1") to perform DISCOVERY operations.

- **Step 3:** The Hosting CSE sends a response to the Originator.

EXAMPLE 2: If Step 1 was successful, "acp1URI" will be contained in this response message.

- **Step 4:** The Hosting CSE generates corresponding ACP Triples based on the content of <acp1> and the ACP ontology.

EXAMPLE 3: An example of ACP Triples for <acp1> resource created in Step 1 is illustrated in figure 8.11.1-2.

Line#1	@PREFIX	acp:	<http://accessControlPolicy.org> .
Line#2	acp:acp1	rdf:type	acp:accessControlPolicy .
Line#3	acp:acp1	acp:hasACPRule	acp:acr11 .
Line#4	acp:acr11	rdf:type	acp:accessControlRule .
Line#5	acp:acr11	acp:hasACOriginator	"AE-ID-1" .
Line#6	acp:acr11	acp:hasACOperations	"DISCOVERY" .

Figure 8.11.1-2: Example ACP triples corresponding to <acp1> resource

- In figure 8.11.1-2:
  - line#1 defines prefix "acp" which will used in lines #2-#6.

- line#2 defines a new acp:accessControlPolicy class instance for <acp1> resource. The subject value of this triple (i.e. acp:acp1) is "acp1URI", therefore the subject value of this triple makes it possible to locate the corresponding resource <acp1>. The Hosting CSE can also use "acp1URI" to locate corresponding triples in the Semantic Graph Store (e.g. when updating existing ACP Triples).

- line#3 defines that acp:acp1 instance has an associated access control rule acr11. The object value of this triple (i.e. acp:acr11) is "acr11URI", therefore the object value of this triple, makes it possible to locate the corresponding *privileges* attribute of <acp1> resource. The Hosting CSE can use "acr11URI" to locate the corresponding triples in the Semantic Graph Store (e.g. when updating existing ACP Triples).

- line#4 defines that acp:acr11 (i.e. the object on line#3) is an acp:accessControlRule class instance.

- line#5 and line#6 give the values of two properties of acp:acr11 based on the assumptions in this example.

NOTE 1: The triples on lines #4-#6 define the access control rule acr11. If <acp1> has more access control rules, additional access control rules will be defined similarly to those on lines #4-#6.

- Optionally: The Hosting CSE may add the address of the Semantic Graph Store to the <accessControlPolicy> resource created in Step 2 in a new attribute, to enable direct addressing of the triples.

- **Step 5:** The Hosting CSE sends a SPARQL request to store the ACP Triples created in Step 4 to the selected Semantic Graph Store.

EXAMPLE 4: The ACP Triples shown in figure 8.11.1-2 will be contained in the SPARQL request.

- **Step 6:** The Semantic Graph Store receives the SPARQL request, processes it and saves the ACP Triples into its graph store.

- **Step 7:** The Semantic Graph Store sends a response back to the Hosting CSE to confirm the request in Step 6 is successfully executed.

The procedure for updating ACP Triples in a Semantic Graph Store follows a similar flow to the procedure used when a new <accessControlPolicy> resource is created. In this case the Originator requests to update the *privileges* attribute of an existing <accessControlPolicy> resource, as shown in figure 8.11.1-3.

NOTE 2: This procedure applies also for updates of the *accessControlPolicyIDs* attribute of the <semanticDecriptor> resource.

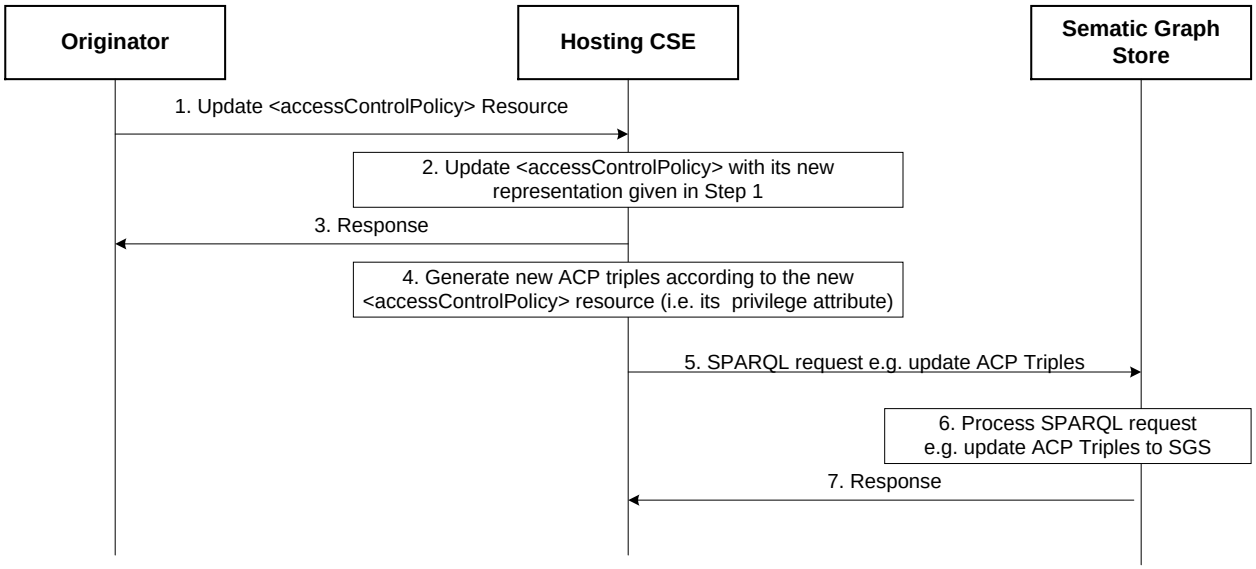

Figure 8.11.1-3: Procedure for updating ACP triples in Semantic Graph Store

- **Steps 1 - 3:** Similar to those describing figure 8.11.1-1, but reflecting normal processing of an UPDATE operation. In this case the Originator triggers an update of the *privileges* attribute of an existing <accessControlPolicy>.

- **Step 4:** Based on the new value of the *privileges* attribute the Hosting CSE generates new ACP Triples.

EXAMPLE 5: Assume the Originator aims to update the *privileges* attribute of <acp1> resource from "DISCOVERY" to "DISCOVERY" and "RETRIEVE" as the new *accessControlOperations*. To implement these changes in figure 8.11.1-3 the Hosting CSE can simply add a new triple e.g. "acp:acr11 acp:hasACOperations "RETRIEVE". Alternatively, the Hosting CSE can replace the triple on Line#6 to the new triple "acp:acr11 acp:hasACOperations "DISCOVERY", "RETRIEVE".".

- **Step 5:** The Hosting CSE sends a SPARQL request to the Semantic Graph Store to update existing ACP Triples related to <acp1> resource to reflect the update being requested.

EXAMPLE 6: As described in Step 4, there are two options to implement this.

- The Hosting CSE adds a new triple with the following SPARQL request:

```
@PREFIX acp: <http://accessControlPolicy.org> .
INSERT DATA
{ acp:acr11 acp:hasACOperations "RETRIEVE" . }
```

- The Hosting CSE replaces Line#6 in figure 8.11.1-3 with the SPARQL request:

```
@PREFIX acp: <http://accessControlPolicy.org> .
DELETE
{ ?acr acp:hasACOperations ?operation}
WHERE
{
?acr acp:hasACOperations ?operation .
FILTER( ?acr = acp:acr11 )
}
INSERT DATA
{ acp:acr11 acp:hasACOperations"DISCOVERY", "RETRIEVE" .
}
```

- **Step 6:** The Semantic Graph Store processes the received SPARQL request and updates the corresponding ACP Triples.

- **Step 7:** The Semantic Graph Store sends a response to the Hosting CSE to inform it whether the request has successfully executed.

The procedure for deleting ACP Triples in a Semantic Graph Store follows a similar flow to the procedure. In this case the Originator requests to delete an existing <accessControlPolicy> resource, as shown in figure 8.11.1-4.

- **Steps 1 - 3:** Similar to those describing figure 8.11.1-1, but reflecting normal processing of a DELETE operation. In this case the Originator triggers the deletion of an existing <accessControlPolicy> resource.

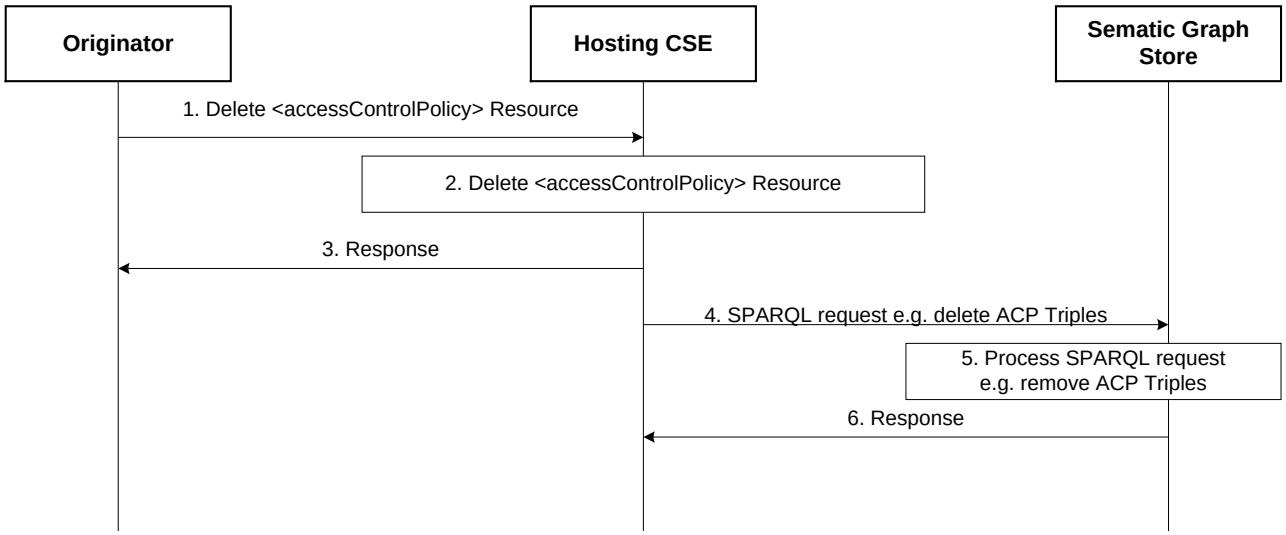

Figure 8.11.1-4: Procedure for Deleting ACP Triples in the Semantic Graph Store

- **Step 4:** The Hosting CSE sends a SPARQL request to the Semantic Graph Store to delete existing ACP Triples.

EXAMPLE 7: The following SPARQL request implements this request:

```
@PREFIX   acp:  <http://accessControlPolicy.org> .

DELETE
{ ?acp    ?p        ?o
  ?s      ?p2       ?acp
  ?acr    ?p1       ?o1
}
WHERE
{
?acp      ?p        ?o
?s        ?p2       ?acp
?acp      acp:hasACPRule  ?acr
?acr      ?p1       ?o1
FILTER ( ?acp = acp:acp1)
}
```

- **Step 5:** The Semantic Graph Store processes the received SPARQL request and removes all requested ACP Triples.

- **Step 6:** The Semantic Graph Store sends a response to the Hosting CSE to inform it whether the request was successfully executed.

8.11.2 Synchronizing SD-related triples between the Resource Tree and the Semantic Graph Store

8.11.2.1 Introduction

Functionality such as Direct Access Control of the Semantic Graph Store require synchronization between information the resources at the Hosting CSE and the Semantic Graph Store. In addition to ACP Triples, others such as SD Original Triples, ACP-SD Binding Triples and SD relationship triples need to be synchronized.

Several use cases are envisioned:

- When a <semanticDescriptor> is created:
  - In this case the Hosting CSE will generate SD Relationship Triples and ACP-SD Binding Triples and then store them in the SGS.
- When the *accessControlPolicyIDs* attribute of a <semanticDescriptor> resource changes:
  - In this case the Hosting CSE will generate new ACP-SD Binding Triples and use them to update corresponding old ACP-SD Binding Triples in the SGS. This case may apply also when the *accessControlPolicyIDs* attribute of the parent changes.
- When the descriptor attribute of a <semanticDescriptor> resource changes:
  - In this case the Hosting CSE will generate new SD Relationship Triples and use them to update old SD Relationship Triples in the SGS.
- When a <semanticDescriptor> resource is deleted:
  - In this case the Hosting CSE will delete all corresponding SD Original Triples, SD Relationship Triples and ACP-SD Binding Triples from the SGS.

8.11.2.2 Procedure for Creating ACP-SD binding triples and SD relationship triples in SGS

Figure 8.11.2.2-1 illustrates the procedure for creating ACP-SD Binding Triples and SD Relationship Triples in SGS, which is triggered when an Originator requests to create a new <semanticDescriptor> resource.

After checking the access rights and other related security functions, the Hosting CSE creates the <semanticDescriptor> resource locally (referred to as SD1 and its URI assumed to be sd1URI). Then, the Hosting CSE will store all semantic triples as described in the descriptor attribute of SD1 resource to the SGS. More importantly, the Hosting CSE will generate new SD Relationship Triples and ACP-SD Binding Triples and store them to the SGS as well. Note that if SD1 has no accessControlPolicyIDs attribute, ACP-SD Binding Triples will not be generated.

The following steps are performed:

- **Step 1:** The Originator sends "Create <semanticDescriptor> Resource" request to the Hosting CSE. It is assumed that the value of descriptor attribute and accessControlPolicyIDs attribute of <semanticDescriptor> resource will be given in this request message:
  - Assume the descriptor attribute contains only one SD Original Triple "S1 P1 O1".
  - Assume the value of accessControlPolicyIDs is "acp1URI" i.e. the access control policy acp1 will be applied.
- **Step 2:** The Hosting CSE accordingly creates the <semanticDescriptor> resource (referred to as sd1):
  - Assume its URI is sd1URI.
- **Step 3:** The Hosting CSE sends a response to Originator to inform it if Step 2 is successfully completed.
- **Step 4:** Bases don the SD Original Triple contained in the descriptor attribute of sd1, the Hosting CSE generates SD Relationship Triples:

- In our example, there is only one SD Original Triple, as shown below:

```
Line#1  @PREFIX     sd:   <http://semanticDescriptor.org> .

Line#2  sd:sd1              rdf:type              sd:semanticDescriptor .
Line#3  sd:tripleInstance11 rdf:type              sd:sdOriginalTriple .
Line#4  sd:tripleInstance11 sd:describedIn        sd:sd1 .
Line#5  sd:tripleInstance11 sd:hasSubject         sd:S1 .
Line#6  sd:tripleInstance11 sd:hasProperty        sd:P1 .
Line#7  sd:tripleInstance11 sd:hasObject          sd:O1 .
```

- **Step 5:** The Hosting CSE will generate the ACP-SD Binding Triples:
  - In our example, since sd1's accessControlPolicyIDs attribute points to acp1 resource as shown below:

```
Line#1  @PREFIX     acp:  <http://accessControlPolicy.org> .
Line#2  @PREFIX     sd:   <http://semanticDescriptor.org> .

Line#3  acp:acp1    rdf:type              acp:accessControlPolicy .
Line#4  sd:sd1      rdf:type              sd:semanticDescriptor .
Line#5  acp:acp1    acp:appliedTo         sd:sd1 .
```

- **Step 6:** The Hosting CSE sends a SPARQL request to the SGS to store these SD Relationship Triples and ACP-SD Binding Triples to the SGS.
- **Step 7:** The SGS processes the SPARQL request and store corresponding SD Relationship Triples and ACP-SD Binding Triples in the SGS.
- **Step 8:** The SGS sends a response message to the Hosting CSE to inform it if the SPARQL request in Step 6 is successfully executed.

NOTE: If the <semanticDescriptor> resource being created in Step 2 does not have accessControlPolicyIDs attribute, the accessControlPolicyIDs attribute of the parent resource may be used or system default access privileges may be applied. The new ACP-SD Binding Triples will also be generated using either the parent resource's accessControlPolicyIDs attribute or based on the default privileges.

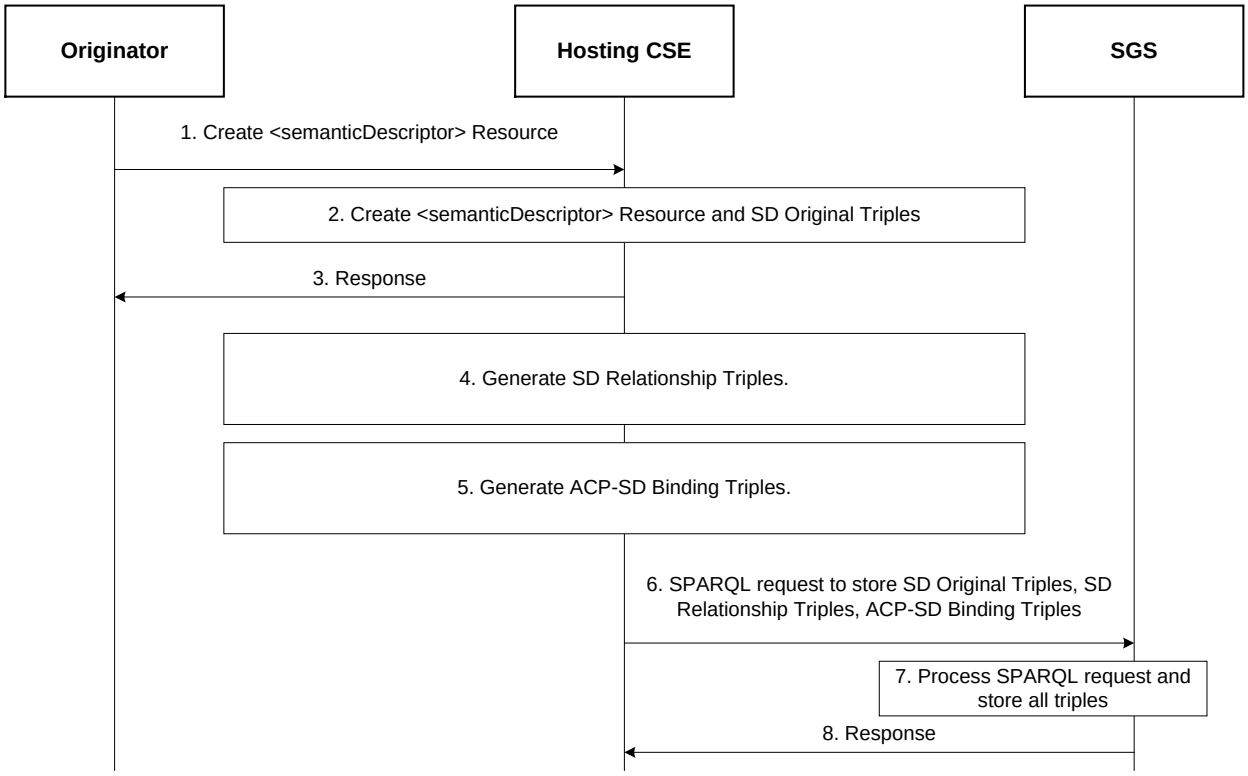

Figure 8.11.2.2-1: Procedure for creating SD relationship triples and ACP-SD binding triples

8.11.2.3 Procedure for updating ACP-SD binding triples in SGS

Figure 8.11.2.3-1 shows the procedure for updating ACP-SD Binding Triples when the accessControlPolicyIDs attribute of a <semanticDescriptor> resource is updated.

For example, the sd1 resource created earlier has accessControlPolicyIDs changed from acp1 to acp2; with the ACP Triples for the resource acp2 as follows:

```
@PREFIX acp:                        <http://accessControlPolicy.org>.
acp:acp2     rdf:type               acp:accessControlPolicy .
acp:acp2     acp:hasACPRule         acp:acr21 .
acp:acr21    rdf:type               acp:accessControlRule .
acp:acr21    acp:hasACOriginator    "AE-ID-2" .
acp:acr21    acp:hasACOperations    "RETRIEVE" .
```

The following steps are performed:

- **Step 1:** The Originator sends a request to update the resource sd1's accessControlPolicyIDs from the URI of the resource acp1 to the URI of the resource acp2. The URI of the resource acp2 (i.e. acp2URI) is contained in this request. The URI of the resource sd1 (i.e. sd1URI) is also contained in this request.
- **Step 2:** The Hosting CSE checks access rights. If it is allowed, the Hosting CSE updates sd1's accessControlPolicyIDs with acp2's URI given in Step 1.
- **Step 3:** The Hosting CSE sends a response back to the Originator to inform it if the request in Step 1 is successful or not.
- **Step 4:** Since the sd1's accessControlPolicyIDs is changed, the Hosting CSE generates a new ACP-SD Binding Triple ("acp:acp2 acp:appliedTo sd:sd1") to reflect this change. This new ACP-SD Binding Triple will replace the old ACP-SD Binding Triple (i.e. "acp:acp1 acp:appliedTo sd:sd1"):
  - (new ACP-SD Binding Triple) acp:acp2 acp:appliedTo sd:sd1
  - (old ACP-SD Binding Triple) acp:acp1 acp:appliedTo sd:sd1

- **Step 5:** The Hosting CSE sends an SPARQL request to replace the old ACP-SD Binding Triple in the SGS with the new ACP-SD Binding Triple as shown in above Step 4. This SPARQL request for this example is shown below:

```
@PREFIX    acp:    <http://accessControlPolicy.org> .
@PREFIX    sd: <http:semanticDescriptor.org> .

DELETE
{ ?acp acp:appliedTo   sd:sd1 }
WHERE
{
?acp   acp:appliedTo   sd:sd1

}
INSERT DATA
{ acp:acp2    acp:appliedTo    sd:sd1 . }
```

- **Step 6:** The SGS processes the SPARQL request and updates the specified ACP-SD Binding Triples in Step 5.
- **Step 7:** The SGS sends a response to the Hosting CSE to inform it if the SPARQL request in Step 5 is successfully performed.

NOTE: If the accessControlPolicyIDs attribute of the <semanticDescriptor> resource was empty to start with, its parent resource's accessControlPolicyIDs may be enforced. The hosting CSE will apply this step based on updates to the accessControlPolicyIDs attribute of the parent resource.

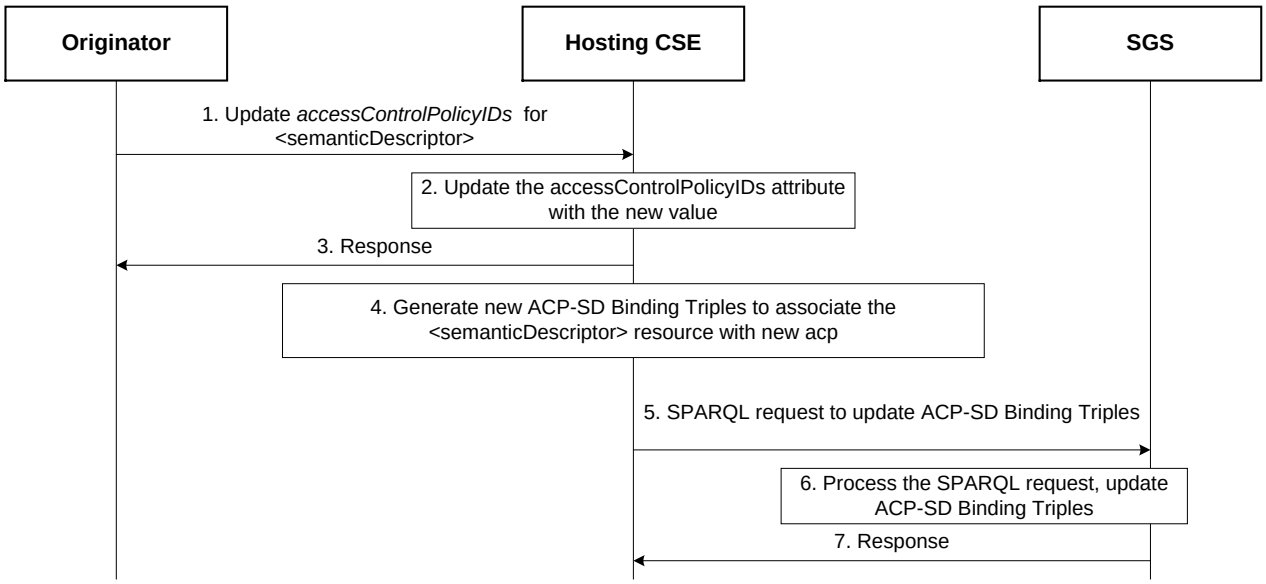

Figure 8.11.2.3-1: Procedure for updating ACP-SD binding triples in the SGS

8.11.2.4 Procedure for updating SD relationship triples in SGS

Figure 8.11.2.4-1 shows the procedure for updating SD Relationship Triples when the descriptor attribute of a <semanticDescriptor> resource is changed.

For example, the descriptor of the sd1 resource created earlier is changed to have two SD Original Triples (Old one - "S1 P1 O1"; New one - "S2 P2 O2").

The following steps are performed:

- **Step 1:** The Originator sends a request to update the resource sd1's descriptor to include one new SD Original Triple (i.e. "S2 P2 O2"). The URI of the resource sd1 (i.e. sd1URI) is also contained in this request.
- **Step 2:** The Hosting CSE checks access rights. If it is allowed, the Hosting CSE updates sd1's descriptor attribute by adding one new SD Original Triple (i.e. "S2 P2 O2").

- **Step 3:** The Hosting CSE sends a response back to the Originator to inform it if the request in Step 1 is successful or not.
- **Step 4:** The Hosting CSE generates new SD Relationship Triples below to reflect this change:
  - In our example:

sd:tripleInstance12	rdf:type	sd:sdOriginalTriple .
sd:tripleInstance12	sd:describedIn	sd:sd1 .
sd:tripleInstance12	sd:hasSubject	sd:S2 .
sd:tripleInstance12	sd:hasProperty	sd:P2 .
sd:tripleInstance12	sd:hasObject	sd:O2 .

- **Step 5:** The Hosting CSE sends an SPARQL request to replace old SD Relationship Triples and/or add new SD Relationship Triple in the SGS with the new SD Relationship Triple generated in above Step 4. This SPARQL request for this example is shown below:

```
@PREFIX   acp:    <http://accessControlPolicy.org> .
@PREFIX   sd:     <http://semanticDescriptor.org> .

INSERT DATA
{ sd:tripleInstance12 rdf:type        sd:sdOriginalTriple .
  sd:tripleInstance12 sd:describedIn      sd:sd1 .
  sd:tripleInstance12 sd:hasSubject       sd:S2 .
  sd:tripleInstance12 sd:hasProperty  sd:P2 .
  sd:tripleInstance12 sd:hasObject        sd:O2 .
}
```

- **Step 6:** The SGS processes the SPARQL request and adds new SD Relationship Triples.
- **Step 7:** The SGS sends a response to the Hosting CSE to inform it if the SPARQL request in Step 5 is successfully performed.

NOTE 1: If an old SD Original Triple is removed or updated by a new SD Original Triple, the corresponding SD Relationship Triples related to this old SD Original Triple will be removed from the SGS.

NOTE 2: The update of triples in the descriptor attribute may be performed also by targeting the *semanticOpExec* attribute of the <semanticDescriptor> parent resource with a SPARQL query; when this SPARQL query is executed, new SD Original Triples may be added to the *descriptor* attribute of the <semanticDescriptor> resource. In this case Steps 4 - 7 will be performed. More specifically, SPARQL Update consists of DELETE and ADD operations, so the SD relationship triples associated with the old original triples will be deleted, and the new ones stored.

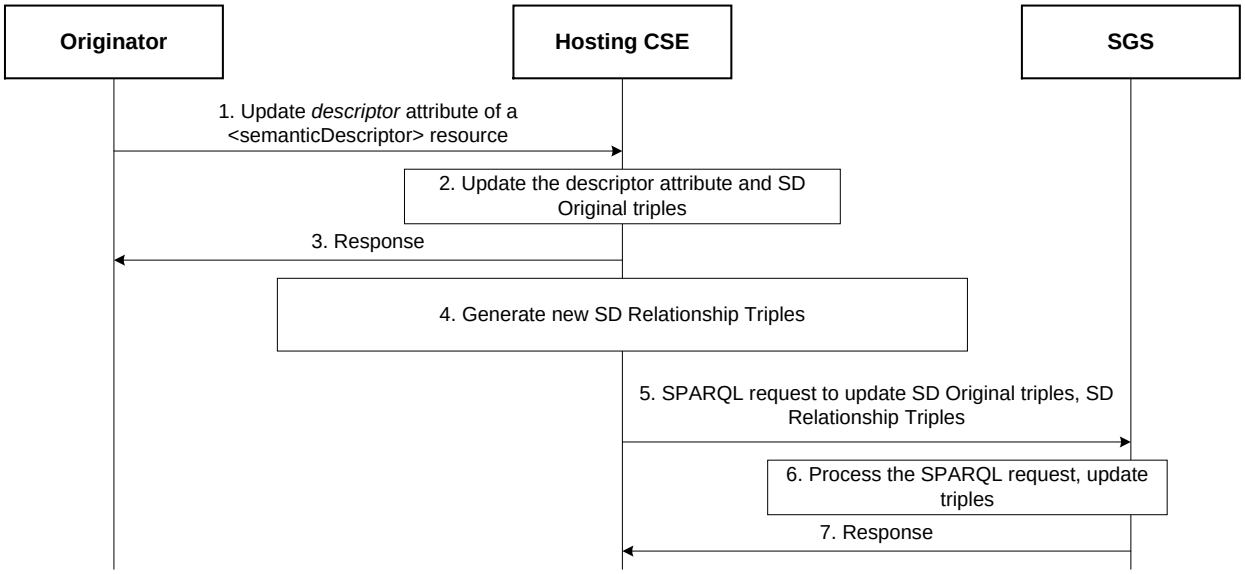

Figure 8.11.2.4-1: Procedure for updating SD relationship triples in the SGS

8.11.2.5 Procedure for deleting SD relationship triples and ACP-SD binding triples in SGS

Figure 8.11.2.5-1 shows a procedure for deleting SD Relationship Triples and ACP-SD Binding Triples from the SGS, which could be triggered by the Initiating AE/CSE or the Hosting CSE to delete a <semanticDescriptor> resource.

For example, the sd1 resource created earlier is removed.

The following steps are performed:

- **Step 1:** The Originator sends "Delete <semanticDescriptor> Resource" to the Hosting CSE to delete sd1 resource. The URI of sd1 resource (i.e. sd1URI) is contained in this request.
- **Step 2:** The Hosting CSE deletes sd1 resource locally.
- **Step 3:** The Hosting CSE sends a response to the Originator to inform it if the deletion request in Step 1 is successful.
- **Step 4:** The Hosting CSE sends an SPARQL request to the SGS to remove SD Relationship Triples and ACP-SD Binding Triples related to sd1 resource. The SPARQL will look like:

```
@PREFIX     acp:    <http://accessControlPolicy.org>.
@PREFIX sd: <http:semanticDescriptor.org> .

DELETE
{ ?sd   ?p  ?o
  ?tripleInstance     ?p1   ?o1
  ?acp  acp:AppliedTo   ?sd
}
WHERE
{
?sd ?p  ?o.
?tripleInstance   ?p1   ?o1.
?tripleInstance sd:describedIn  ?sd .
?acpacp:AppliedTo   ?sd
FILTER ( ?sd = sd:sd1)
}
```

- **Step 5:** The SGS processes the SPARQL request in Step 4 and removes corresponding SD Relationship Triples and ACP-SD Binding Triples.
- **Step 6:** The SGS sends a response to the Hosting CSE to inform it if the SPARQL request in Step 4 is successfully performed.

NOTE: Steps 4 - 6 will also be performed if a SPARQL query targeting the semanticOpExec attribute of a <semanticDescriptor> resource results in the deletion of existing SD Original Triples.

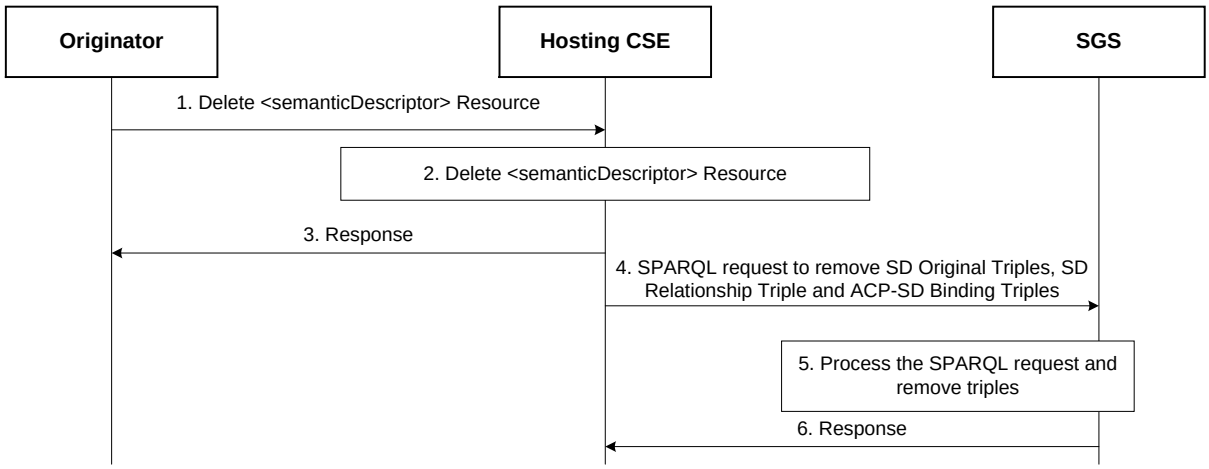

Figure 8.11.2.5-1: Procedure for deleting SD relationship triples and ACP-SD binding triples from the SGS

9 Conclusions

The semantic functionality supported by oneM2M has been significantly enhanced in Release 3 - now also supporting semantic queries and semantic mashups. In this document the different options have been developed that helped in deciding what approach is most suitable to be adopted in the oneM2M specifications, i.e. oneM2M TS-0001 [i.3], oneM2M TS-0004 [i.4] and oneM2M TS-0034 [i.12]. In addition to the approaches to be specified, some implementation aspects and options have been discussed. These are not specified by oneM2M in a normative way and are thus up to the implementer, but are important for understanding how the semantic features can be implemented.

History

Publication history		
V3.0.0	May 2019	Release 3 - Publication