UPPSALA
UNIVERSITET

IT 14 019
Examensarbete 30 hp
Mars 2014

Integrating SciSPARQL and MATLAB

Xueming He

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Integrating SciSPARQL and MATLAB

Xueming He

Nowadays many scientific experiment results involve multi-dimensional arrays. It is desirable to store these results in a persistent way and make queries against not only well-structured data objects like arrays but also the metadata that describe the experiments. SPARQL is a Semantic Web standard query language for data and metadata stored in terms of RDF. SciSPARQL is an extended version of SPARQL designed for scientific applications. It includes numeric multi-dimensional array operations and user-defined functions. The SciSPARQL Database Manager (SSDM) is a query processing engine for SciSPARQL. MATLAB is a popular and powerful scientific computing application programming language. We implemented an interface between MATLAB and SciSPARQL called MATLAB SciSPARQL Link (MSL). MSL provides SciSPARQL queries in MATLAB through a client/server interface. It optionally also provides an interface to enable calls to MATLAB in SciSPARQL queries. With MSL MATLAB users can populate, update, and query SSDM databases it in terms of SciSPARQL queries. For the implementation we use C interfaces of MATLAB and SSDM, and the networking capabilities of SSDM. The DLL we made extends MATLAB with MSL interface functions.

Key words: SPARQL; MATLAB; SciSPARQL; RDF; client/server interface

Handledare: Andrej Andrejev
Ämnesgranskare: Tore Risch
Examinator: Ivan Christoff
IT 14 019
Tryckt av: Reprocentralen ITC

Content

1. Introduction

Scientific experiments often involve numeric multi-dimensional arrays. It is desirable to store these results in a scalable way and make flexible queries against not only well-structured data objects like arrays but also the metadata that describe the experiments.

SPARQL is a query language for finding combinations of data and metadata defined in terms of RDF as 'triples of knowledge' [8]. RDF aims at improving the maintainability and flexibility of Linked Data [6] by allowing anyone to make statements about any resource on the web. However, RDF has weak support for representing arrays. In RDF arrays have to be broken down into triples, which is both unnatural and inefficient. SSDM (Scientific SPARQL Data Manager) [4] is a database management system (DBMS) for storing and querying data used in scientific experiments. SSDM provides scalable storage representation of arrays for RDF. SSDM supports the query language SciSPARQL, which is an extension of the W3C standard Semantic Web query language SPARQL [1]. SciSPARQL extends SPARQL with primitives to search not only metadata about scientific experiments, but also experimental data represented as arrays. SciSPARQL supports scientific applications with a number of features, in particular:

- Numeric multidimensional array operations
- User-defined functions (e.g. Java or Python)
- User-defined aggregate functions
- Views of data defined as functions

The MATLAB language is a popular high-level language for scientific and engineering computing whose basic elements are arrays. The goal of this project is to enable MATLAB application programs to generate and store scientific data in a scalable storage, which can be queried using the query language SciSPARQL. For this purpose, we implemented an interface between the MATLAB runtime system and SSDM called MATLAB-SSDM Link (MSL).

Table.1 below compares the capabilities of SPARQL, SciSPARQL, and MATLAB for processing metadata and scientific data including arrays.

	MATLAB	**SPARQL**	**SciSPARQL**
Metadata		√	√
Scientific data including Arrays	√		√

Table.1 Comparison of data processing capabilities of MATLAB, SPARQL, and SciSPARQL

MSL provides the an interface between MATLAB and SSDM called the *MATLAB Client-Server Interface* (MCSI) that allows MATLAB application programs to send SciSPARQL statements to an SSDM database in order to populate, query, and update the database. Fig.1 below shows the overall architecture of the system with the place of MCSI in the software stack.

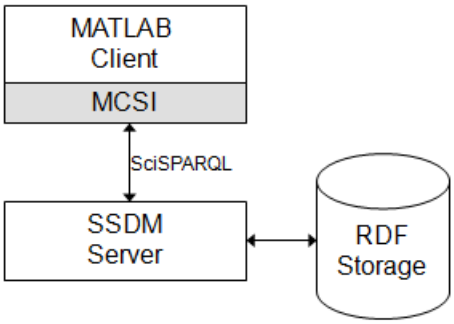

Fig.1 The MATLAB-SSDM Link

MCSI is implemented as a dynamic link library (DLL) that is loaded into MATLAB. The MCSI interface functions enable connecting to an SSDM server, sending SciSPARQL queries, updates, and directives (e.g. function definitions) to the server for execution, iteratively retrieving query results from the server, and bulk loading RDF datasets into an SSDM database. MCSI provides MATLAB functions and classes for MATLAB users/programmers so that they can conveniently use SciSPARQL for storing and querying large collections of numerical arrays managed by SSDM.

In summary, MSL provides MATLAB and SSDM with the following features:

- A consistent storage for scientific data including arrays as objects of RDF triples.
- A MATLAB Client/Server Interface (MCSI) to query SSDM databases from MATLAB in terms of SciSPARQL.
- Shipping commands to the SSDM server for execution, e.g. to define functions used in SciSPARQL queries.
- Bulk-loading of MATLAB data into SSDM databases.

MSL is implemented using the MATLAB application program interfaces [10], which are a set of libraries that allow programmers to write programs in other language like C or Fortran that interacts with MATLAB. MSL is delivered as a C dynamic-link library (DLL). In particular MSL implements a mapping between MATLAB arrays and the SSDM data type *NMA* [4] that represents multi-dimensional arrays in SSDM. It also provides mappings between RDF data types through MATLAB's object-oriented programming features. Other RDF data types, like numbers are mapped to native MATLAB types.

2. Background

This chapter introduces the relevant technologies to this project, including the RDF data model, the SPARQL query language, the SciSPARQL extensions, SSDM, the Amos II system [16] on top of which SSDM is implemented, and the MATLAB language.

2.1 RDF and SPARQL

RDF is a data model developed by W3C to represent mainly web-based metadata. It has an abstract syntax that reflects a simple labeled graph-based data model [2]. RDF uses an extensible URI-based representation of entities [15] and XML Schema data typess [14]. RDF is used to represent metadata, e.g. to describe properties of resources on the web [8].

RDF expression of simple facts
Generally, RDF represents simple assertions (facts) called *statements* [1]. A statement consists of three parts: a *subject*, a *predicate* (also called property), and an *object*, denoted as a triple *(subject, predicate, object)*. An RDF triple represents a relationship, indicating that the *predicate* holds between the entities *subject* and *object*. For example, the triple *(bistab:Experiment1, bistab:Subtask, bistab:Task1)* states that he experiment represented by the URI *bistab:Experiment* has a property *bistab:Subtask* being the URI *bistab:Task1*. The triple is lustrated by the graph in Fig. 2.

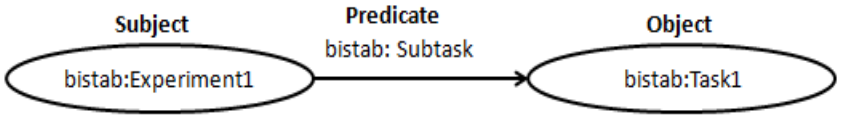

Fig.2: An RDF statement about a subtask of an experiment

As another example, the triple *(bistab:Experiment1, bistab:Responsible, "Emily")* illustrated by Fig.3 expresses that the experiment *bistab:Experiment1* has a property *bistab:Responsible* being "Emily".

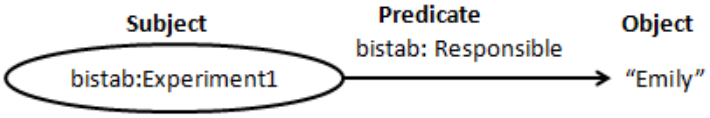

Fig.3: An RDF statement expressing the responsible person of an experiment

The triple *(bistab:Experiment1, bistab:StartDate, "2013-09-10")* in fig.4 states that an experiment, *bistab:Experiment*, has a property, *bistab:StartDate*, whose value is *2013-09-10*.

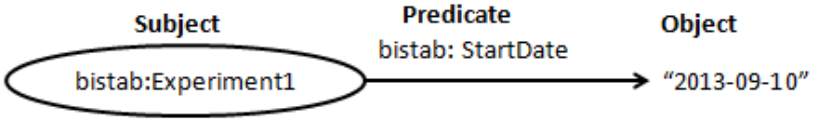

Fig.4 RDF statement expressing the start date of an experiment

In the examples above, a predicate denotes a relationship between a subject and an object. The direction of the arc is significant since it always points from a subject toward the object. The colon notation abbreviates

*URI*s with common prefixes, e.g. *bistab:Experiment1,stands for <http://udbl.it.uu.se/bistab#Experiment1>*, given a prefix

```
bistab : <http://udbl.it.uu.se/bistab#>
```

is defined both in the dataset and in the queries.

RDF Databases

A set of RDF triples can be seen as an RDF database. For example, by combining the above three RDF triples we get the RDF database consisting of the triples:
(bistab:Experiment1, bistab:Subtask, bistab:Task1)
(bistab:Experiment1, bistab:StartDate, "2013-09-10")
(bistab:Experiment1, bistab:Responsible, "Emily")

The example database is illustrated by Fig.5.

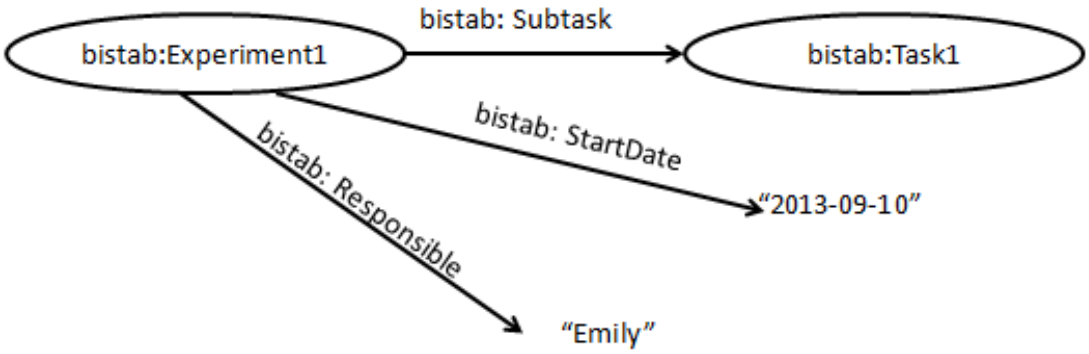

Fig.5 RDF statements about an experiment

```
@prefix bistab: <http://udbl.it.uu.se/bistab#> .
bistab:Experiment1 bistab:Subtask bistab:Task1 .
bistab:Experiment1 bistab:StartDate "2013-09-10".
bistab:Experiment1 bistab:Responsible "Emily" .
```

Making simple queries against an RDF database

Once an RDF database has been populated, we can query it in SPARQL, which provides a general way to search RDF databases where data and metadata are represented as triples. SPARQL queries search for the relevant RDF data by matching *triple patterns* against the database. Triples patterns are similar to RDF triples with the difference that any element in a triple pattern (*subject, predicate,* or *object)* may also be a variable. An example of a triple pattern is, ?*s :p ?o,* in which *?s* and *?o* are variables, and :p is a URI representing a predicate. The following example shows how to query the database above to get the value of the responsible person for *bistab:Experiment1* together with the starting date of the experiment:

```
SELECT  ?responsible ?date
    WHERE {
         Bistab:Experiment1 bistab:Responsible ?responsible .
         Bistab:Experiment1 bistab:StartDate ?date }
```

SELECT returns sets of tuples based on matching triple patterns against the RDF database. A SPARQL query consists of two parts, the *SELECT* clause and the *WHERE* clause. The *SELECT* clause defines the result tuples of the query. In the example the result tuples have two elements defined by the variables *?responsible* and *?date*. The WHERE clause defines the triple patterns. In the example there are two triple patterns. The possible variable bindings in the result tuples are obtained by matching the triple patterns against the RDF database. In the example the variable *?responsible* matches *"Emily"* and *?date* matches *"2013-09-10"*.

The advantage of using RDF for scientific computing is that it provides a powerful data model for representing properties (metadata) about scientific and engineering experiments, while SPARQL provides a powerful high level query language to search the metadata. However, RDF does not support storing the scientific results themselves as numerical arrays. To enable storing both data and metadata using RDF, the data representation of RDF and the query capabilities of SPARQL were extended with ability to store and query numerical arrays, for which purpose SSDM and SciSPARQL were developed. Since MATLAB is commonly used for processing scientific data there is a need to interface MATLAB with SSDM through the query language SciSPARQL, which is the purpose of this project.

2.3 MATLAB

MATLAB is a popular high-level programming language in scientific computing as it has support for processing and visualizing large numeric matricies. The MATLAB system consists of five main parts: the MATLAB language, the MATLAB working environment, a graphic system, the MATLAB mathematical function library, and the MATLAB application program interfaces. External programming language interfaces support the extensibility of MATLAB to call functions implemented in other languages e.g. C or Java. Furthermore, functions in MATLAB can also be called from other languages. MATLAB has high-level functions for high-dimensional data processing and calculation. The basic data type of MATLAB is a matrix, which is one reason why MATLAB is widely used and popular in scientific computing experiments.

The core issue in MSL is providing a way to access SSDM servers from MATLAB programs so that SSDM databases storing experimental data can be searched and updated from MATLAB programs. In addition, MSL also allows MATLAB functions to be called in SciSPARQL queries to utilize the vast MATLAB function library over matrices. The implementation of MSL includes the mapping between the multi-dimensional numeric array representations of MATLAB and SSDM.

The rest of this section describes some basic MATLAB functionality for readers not familiar with it.

Accessing single element

A particular element in a matrix can be referenced by specifying the row and column of the matrix in the following way, *A(row, column),* in which *A* is the matrix. The example below gets the element at the second row and second column of a 3-by-3 magic square *A*.

```
A=magic(3)
A =  8 1 6
     3 5 7
     4 9 2
```

A(2,2) is used to get the element at row 2, column 2 and 5 is the returned element.

```
A(2, 2)
ans = 5
```

Accessing multiple elements

Multiple elements can also be referred to in a MATLAB matrix. Subscript expressions involving colons refer to portions of a matrix. The expression *A(1:m, n)* refers to the elements in the rows *1* through *m* of column *n*. The expression *A(1:2:m, n)* gets nonconsecutive elements where *2* is the *step*. This expression refers to every second elements in rows *1* through *m* of column *n*. MATLAB supports an array indexing scheme that uses one array as the index into another array. Example below uses the same array *A* created previously to illustrate indexing multiple elements with another array. In this example, *b* is an array with three elements 1, 2, 3 and the expression *A(b)* refers to the 1st, 2nd and 3rd elements in the sequence format.

```
A =
```

```
     8 1 6
     3 5 7
     4 9 2

b= [1, 2, 3]
b = 1 2 3

A(b)
ans = 8 3 4
```

Accessing elements in multi-dimensional arrays

In MATLAB, a high-dimensional array is an array having more than two dimensions. They are the extension of normal two-dimensional matrices. Subscripts are used to index element in a high-dimensional array as well. A four-dimensional array, for example, will have four subscripts. To index the element in *row* 2, *column* 3 and *page* [12] 2 of a three-dimensional array below, the subscript used will be *A(2, 3, 2)*.

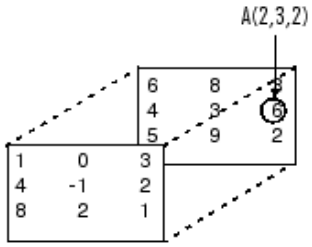

Fig.6 Using subscript to index an element in a high-dimensional array [7]

MATLAB functions

MATLAB users can call built-in functions, define their own functions and call the user-defined functions. The script below defines a MATLAB function *new3Darray* that calls a MATLAB built-in function *cat* [13] to extend a 2D array to a 3D array.

```
function newArray = new3Darray(array)
newArray = cat(3, array, array+1, array+2);
end
```

MSL provides MATLAB functions to let MATLAB applications access SSDM through SciSPARQL.

2.2 Scientific SPARQL and SSDM

SciSPARQL is a query language that extends SPARQL in functionality of array representation and operations over scientific data. It is processed by SSDM, a prototype system utilizing the extensible DBMS Amos II.

Fig.7 shows the overall architecture. Application programs and users ship SciSPARQL queries and command to the SSDM server for execution. SciSPARQL maintains its own main-memory local database for RDF triples. It also provides extensibility by allowing different kinds of back-end storage managers for RDF, e.g. a relational database (RDB). SSDM can read and write RDF triples from or to external files. SSDM has special representations methods for numerical data such as matrices. The system also is extensible by enabling calls to external computational engines, such as Python, in order to make use of the extensive libraries for scientific and engineering computing provides by these.

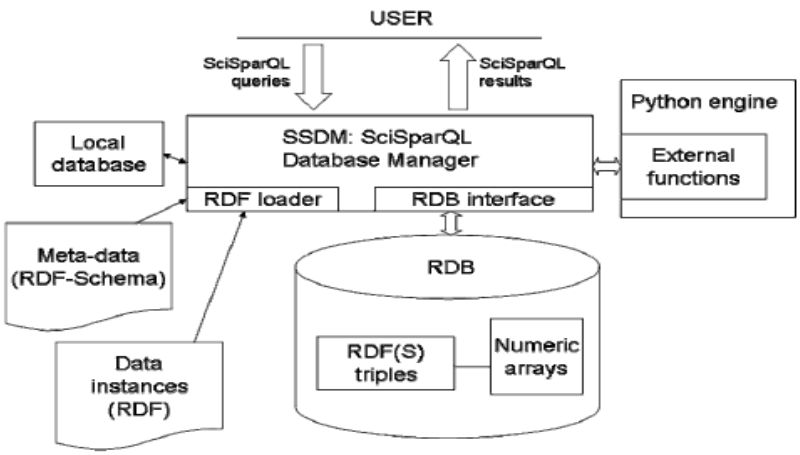

Fig.7 Overview of SSDM architecture [4]

The purpose of SciSPARQL is to enable scientific applications to manage their data, not only metadata. An important feature of SSDM is that it provides an efficient way to store scientific numerical arrays as its built-in data type, called *NMA*.

Fig. 8 below shows how an array [1, 2, 3, 4] is represented in standard RDF. As a comparison, Fig. 9 shows how the same array is represented much simpler in SSDM, where the entire array is a single object.

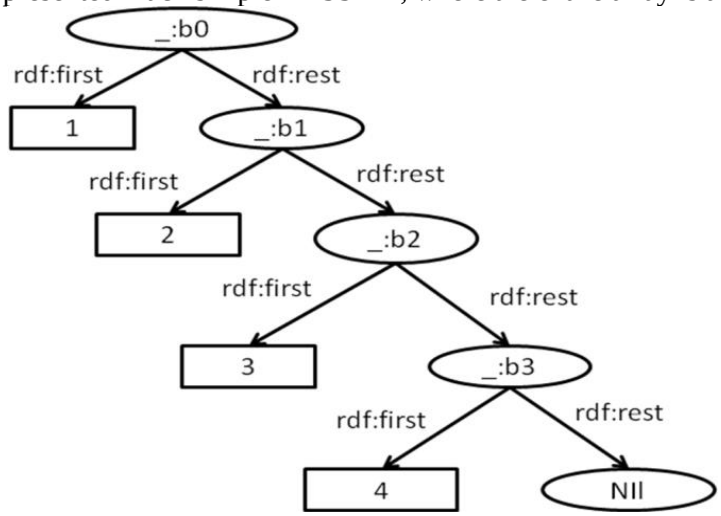

Fig.8 Naïve RDF representation of an array

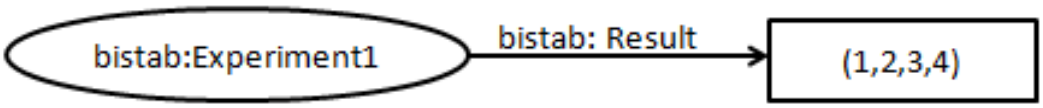

Fig.9 Array-valued triple using the *NMA* data type

The RDF statement on Fig.9 states the fact that the experiment, *bistab:Experiment1*, has a result, *bistab:result*, whose value is an array [1,2,3,4]. Once the data has been populated and stored in SSDM, queries can be made against the an RDF database containing the single triple

```
bistab:Experiment1 bistab:Result (1,2,3,4) .
```

The following query searches the database to get the first element of the result array from the experiment, *bistab:Experiment.*

```
SELECT (?a[1] AS ?res)
 WHERE { bistab:Experiment1 :bistab:Result ?a}
```

The AS clause assigns an expression (here the array reference *?a[1]*) to a variable (*?res*). SciSPARQL and SSDM provide functionality that includes multi-dimensional arrays, array references, and array slicing to allow scientific applications to store and query both scientific data and metadata in a uniform way. This project provides an interface between SSDM and MATLAB that lets MATLAB applications utilize the functionality of SciSPARQL to store, search, and update both data and metadata.

2.3 Amos II

SSDM is built on top of Amos II [19], which is an extensible DBMS allowing different kinds of back-end data sources to be queried. It is centered on a query processor for object-relational and functional queries. The data model of Amos II is based on three concepts: *objects, types,* and *functions*: a database is a set of *objects*, *types* classify the objects into different entity types, and *functions* define properties of objects and relationships between objects. Amos II provides a query language called AmosQL in terms of its data model. SSDM utilizes the query processor of Amos II to process SciSPARQL queries over data stored in the local main-memory database or in different kinds of back-end databases. SSDM translates SciSPARQL into AmosQL for query processing.

MATLAB and SSDM are interfaced by utilizing the C-based APIs provided by MATLAB [10] and Amos II[5].Amos II provides a main-memory storage manager *aStorage*[15], where an entire *local* main-memory database can be stored. In particular *aStorage* is *extensible* so that programmers can define their own data typess using a C-based API. In SSDM a special data type *NMA* [4] for efficient representation of arrays and array projections is registered with *aStorage*. MSL utilizes *aStorage* to represent in main-memory the data structures to be exchanged between MATLAB and SSDM in a local *aStorage* database linked to MATLAB. In MSL the NMA data type represents MATLAB arrays. The client-server communication primitives of Amos II are used for transferring these local data structures to the SSDM server for evaluation.

All data in *aStorage* is referenced through *handles* (C type named *oidtype*), which are indirect identifiers for physical data records in a memory area holding the database, called the *database image*. A type tag is associated with each handle. In order to make the application code both fast and independent of the internal representation of handles, the handles are always manipulated through a set of C macros and utility functions. SSDM not only supports Amos II built-in data types but also defines its own data types using *aStorage*. MSL utilizes *aStorage* to represent data in SSDM converted from/to MATLAB matrices.

A particularly important issue when interfacing MATLAB with a database manager such as SSDM is how to efficiently convert data representations in MATLAB (arrays) into corresponding data representations in the database. In MSL the module MAL (MATLAB – Amos Link) interfaces MATLAB with the Amos II kernel. The interoperability requires conversions (mappings) between Amos II data of different types with the corresponding MATLAB data representations. Both systems are linked together and share main memory. The different kinds of data mapping functions copy data between the database image of Amos II and MATLAB's workspace.

MAL provides a full AmosQL interface between MATLAB and Amos II based on the query language AmosQL. This is a prerequisite for MSL since AmosQL queries, function calls, result management (scans), error handling, and other facilities are internally used by MSL.

3. The MATLAB-SSDM Link (MSL)

MSL provides two interfaces between MATLAB applications and the SSDM system, namely the *MATLAB Client/Server Interface (MCSI)* and the *MATLAB Amos Interface (MAL)*. MSCI allows MATLAB client applications to ship SciSPARQL queries to an SDM server, while MAL allows the same thing for Amos II queries.

3.1 The MATLAB Client/Server Interface (MCSI)

The client-server interface MCSI is first illustrated through example. Then the available MATLAB functions in MSCI are then described in details.

This section explains how to:

1. Initialize the SSDM system.
2. Create a connection to an SSDM server named *udbl.it.uu.se*.
3. Populate the SSDM database.
4. Send a query to the database server for evaluation.
5. Iteratively retrieve data from the database.
6. Unload MSL from MATLAB.

First the MATLAB function *sparqlInit()* is called to load the MSL DLL into MATLAB and initialize the system. After MSL has been initialized, MATLAB users can connect to an SSDM sever on host *h* by calling *newConnection(h)*. The function *newConnection()* returns a connection identifier *Cid* which is used in MCSI interface functions when communicating with the server.

3.1.1 Queries

The function *sparqlQuery(Cid, q) -> Sid* evaluates the query string *q* in the SSDM server connected to by *Cid*. The result of a *sparqlQuery()* call is a *scan* identifier *Sid* that identifies the set of result tuples from the query. Since the result set can be very large it is not returned as a single object but as a *scan*, which is a data structure over which the application program can iterate to retrieve from the result set one tuple at the time.

For example, the following program opens a connection to the SSDM server running on *udbl.it.uu.se* and retrieves all the triples in the default graph:

```
sparqlInit();
Cid=newConnection('udbl.it.uu.se')
Sid = sparqlQuery(Cid,'SELECT ?s ?p ?o WHERE {?s ?p ?o}');
while endOfScan(Sid)~=1
    getElement(Sid,1)
    getElement(Sid,2)
    getElement(Sid,3)
    nextRow(Sid);
end
closeConnection(Cid);
```

The function *endOfScan(s)* returns 1 when there is no more tuples in the scan, and 0 otherwise. The function *getElement(Sid, p)* retrieves the object at position *p* in the current tuple of the scan. The function *nextRow(Sid)* advances to the next tuple in the scan. The function *closeConnection(Cid)* closes a connection and frees all associated scans.

The data objects created by MSL, e.g. connections, scans, and tuples, are stored in a local main-memory *aStorage* database. The function *getElement()* converts the data representation used in SSDM to

corresponding MATLAB data typess. The supported data typess are scalar values, numeric arrays, character strings, and classes defined by MSL to represent RDF-specific data types. For example, the function *makeURI(s)* creates a new URI named *s*, which is an instance of a MATLAB class *URITYPE* defined by MSL.

3.1.2 Function calls

SciSPARQL functions can called from MATLAB using the MCSI MATLAB function *sparqlCall(Cid, fn, argl) -> Sid*, where *fn* is the name of the SciSPARQL function to call and *argl* is an argument list represented as a MATLAB *cell array*[9], which can contain data of varying sizes and types. For example, the following snippet calls the SciSPARQL function *plus* to add two 2D arrays *A* and *B* together:

```
A = [1,2,3;4,5,6];
B = [7,8,9;10,11,12];
Sid = sparqlCall(Cid,'plus',{A,B});
if endOfScan(Sid)~=1
  getElement(Sid,1)
end
```

The user can define SciSPARQL functions and install them on the SSDM server by, e.g., using a MATLAB workspace as an SSDM console where commands as shipped to the SSDM server using the function *sparqlQuery()*. For example, the SciSPARQL function *square(x)* can be defined on the server by:

```
sparqlQuery(Cid,'DEFINE FUNCTION square(?x) AS SELECT (?x*?x AS ?res)');
```

Notice that you normally define the function on the server once.

3.1.3 Updates

The function *rdfInsert(Cid, s, p, o)* inserts a triple *(s, p, o)* into the SSDM database connected to by *Cid*. For example, the following snippet adds values of the property *http://example.org/ns#p* to four different subjects:

```
p=makeUri('http://example.org/ns#p'); % make a URI object
A=[1,2,3;4,5,6]; % make a 2D array
B=cat(3,A,A+1,A+2,A+3); % make a 3D array
rdfInsert(c1,makeUri('http://example.org/ns#x4'),p,'simple string');
rdfInsert(c1,makeUri('http://example.org/ns#x5'),p,3);
rdfInsert(c1,makeUri('http://example.org/ns#x6'),p,B);
rdfInsert(c1,makeUri('http://example.org/ns#x7'),p,true);
```

The variable *p* holds the URI of the added property. The variable *A* is a 2D array used to construct the 3D array *B*. *B* illustrates the functionality of SciSPARQL and SSDM to be able to represent multi-dimensional numeric arrays in queries and the database.

To delete triples there is a function *rdfDelete(Cid, s, p, p)*. Often deletion involves deleting many triples based on a query. For example,

```
sparqlQuery(Cid,'DELETE {?s, <http://example.org/ns#p>, ?o}');
```

removes all triples having the property *http://example.org/ns#p*.

Passing parameters into updates can be done either by placing them into the query string (not recommended) or by defining update in a named procedure shipped to the SSDM server. For example,

```
sparqlQuery(Cid,'DEFINE PROCEDURE delProp(?x) AS DELETE (?s ?x ?o)');
```

defines a procedure stored on the server that delete all triples having the property *?x*. Procedures are called the same way as functions. For example,

```
sparqlCall(Cid,'delProp',{makeURI('http://example.org/ns#p')});
```

, which also removes all triples in the database having the property *http://example.org/ns#p*.

3.2 MATLAB interface functions

The following MATLAB functions constitute the API to SSDM.

3.2.1 Initializing and finalizing

sparqlInit ()
This function initializes the SSDM system by loading the DLL into MATLAB.

Cid = newConnection(hostid)
The function creates a connection to a server running on a given host, *hostid.* This function returns a connection id, *Cid,* which is the unique identifier of every connection. Most functions provided in MSI require a *connection* argument. If *hostid* is an empty string, an embedded SSDM process is initialized.

setRemotePort(portNumber)
Set the port number a name server is expected to listen to. The default is 35021.

freeConnection(Cid)
This function closes the connection and frees memory allocated for the connection. After calling this function, the connection cannot be used any more. To start a new connection call *newConnection(hostid).* The purpose is to keep the memory clean when we want to re-initialize the unit while continuing running the same MATLAB application.

freeAllConnections()
This function closes all connections and frees all memory allocated for all connections. The main purpose of this function is to free memory.

finalizeSystem()
This function frees all memory associated with SSDM and unloads the interface MSL DLL from the MATLAB application.

3.2.2 Sending queries to SSDM

Sid = sparqlQuery(Cid, q)
The function takes two arguments, a connection identifier *Cid* and a query string *q.* It sends the query string to the SSDM server for execution. It returns a scan identifier, *sid,* representing the result of the query so that the MATLAB program can iterate over the result.

Sid = sparqlFunction(Cid, funName, argList)
This function calls a SPARQL function on the server. It takes three arguments, *Cid, funName, argList,* where *Cid* is the connection id, *funName* is the SPARQL function name, and *argList* is a MATLAB cell array holding the arguments that are passed to the SPARQL function. The result *Sid* is a scan of the result.

rdfInsert(Cid, S, P, O)

The function loads an RDF triple *(S, P, O)* into the database by providing *S* as *subject*, *P* as *predicate*, and *O* as *object*. The subject and the predicate must be of type of *URITYPE*, which is defined as a MATLAB data type and can be constructed by calling the *makeUri(uri)* function where the object can be of any type. The data type definitions in MATLAB's workspace and mappings between MATLAB and SSDM data types will be discussed in chapter 4.

end = endOfScan(Sid)
This function checks if the end of a scan *Sid* has been reached, i.e. if the last tuple has been retrieved. The result is a *boolean* value, where 1 (true) indicates that the end of the scan is reached and 0 otherwise.

width = scanWidth(Sid)
This function returns the number of elements in the current tuple of a scan. When applied to an empty scan, 0 is returned.

element = getElement(Sid, pos)
This function takes two inputs, *Sid* and *pos. Sid* specifies scan and *pos* specifies what element in the current *tuple* of scan *Sid* to access The SSDM data type in that position is mapped to the corresponding MATLAB data type.

nextRow(Sid)
This function advances to the next tuple in a scan *Sid*. Before calling *getElement(Sid, pos)* function, *endOfScan(Sid)* should be called to check if the end of a scan is reached or not.

printRow(Sid)
The function prints all elements in the current tuple of scan *Sid*.

freeScan(Sid)
If a scan is no longer used, this function is called to free the memory allocated for the scan. The scan is also automatically freed when the end-of-scan is reached, so *freeScan()* is used only when prematurely ending a scan.

freeAllScans()
This function frees the memory allocated for all scans.

3.2.3 Constructors

makeTimeVal(timeVector)
This data constructor creates a TIMEVALTYPE object representing time stamps. It takes one argument, *timeVector,* which is a vector representing time, e.g., [2013 12 25 12 55 33], and constructs a *TIMEVALTYPE* object which holds the value of *timeVector* as a property.

makeTimeVal(timeVector, timeZone)
This function creates TIMEVALUETYPE object representing containing time zone information passed in *timeZone* parameter. That parameter is an integer number denoting "seconds west of greenwich", e.g. -3600 for Uppsala.

makeUri(uriString)
This function creates a URITYPE object with URI string *uriString*.

makeUniStr(string, langTag)
This function creates a UNISTRINGTYPE object. It takes two arguments, and stores them in *String* and *LangTag properties*, respectively.

makeTypedRDF(literal, datatype)
This function creates a TYPEDRDFTYPE object representing typed RDF literals of the specified *datatype*, given as a string. A typed literal is string combined with a datatype URI. Typed strings enables the RDF type system to be extensible with literals in new datatypes.

3.3 MAL (MATLAB-Amos Link)

The MATLAB-Amos interface (MAL) makes is possible to connect to Amos II from MATLAB. MAL is similar to MSL: first Amos II is initialized calling *amosInit()*. Then MATLAB users can connect to any Amos II database by calling *newConnection(hostid, peer).* After a connection AmosQL queries can be executed for the particular connection by calling *amosQuery(Cid, query)* function. This function returns a scan id, *Sid* to MATLAB's workspace. As for MSL, the elements can be retrieved by calling *getElement(Sid, pos)*, end of scan can be checked by calling *endOfScan(Sid)*, and the next row in a scan can be advanced to by calling *nextRow(Sid).*

3.3.1 MAL MATLAB API examples

The snippet below shows how to use the API functions provided by MAL. Similar as MCSI, it first initializes MAL and then it connects to the server running on a certain host specified by the *hostid.* Through this connection, an AmosQL query is executed, and each row in the scan is iteratively printed out.

```
amosInit();

c1=newConnection('myserver.it.uu.se') % connect to server on a given host
s1=amosQuery(c1,'select i, "a"+i, i/3, {i,i+2}, {"b",i} from Integer i where i in iota(1,5);');
while ~endOfScan(s1)
    printRow(s1);
    nextRow(s1);
end
finalizeSystem();
```

The result of this query contains elements of type, *integer*, *string*, *real number*, *vector* of numbers, *vector* of mixed *numbers* and *strings.* This interface maps different data types between MATLAB and Amos. All elements of the query result are shown below, those shown as `[1x2 double]` represent MATLAB *vectors* mapped from values of Amos' *vector type* holding only numbers; and elements shown as `{1x2 cell}` represent MATLAB *cell array* mapped from values Amos *vector type, containing elements of different types.*

```
[1]    'a1'    [0.3333]    [1x2 double]    {1x2 cell}
[2]    'a2'    [0.6667]    [1x2 double]    {1x2 cell}
[3]    'a3'    [1]         [1x2 double]    {1x2 cell}
[4]    'a4'    [1.3333]    [1x2 double]    {1x2 cell}
[5]    'a5'    [1.6667]    [1x2 double]    {1x2 cell}
```

As soon as all the elements in a scan are retrieved, the scan is freed automatically inside *endOfScan(Sid)* function.

3.3.2 MAL MATLAB API functions

As SSDM is built on the top of Amos, basically most the functions provided by MAL are the same as the functions provided by the MSL except the functions listed below.

amosInit()

This function simply does the same things as *sparqlInit()*, but with different function name to indicate MATLAB-Amos Interface.

Sid = amosQuery(Cid, query)
This function allows MATLAB users to execute AmosQL queries. The function takes two arguments, *Cid* and *query* string, in which *Cid* is the connection id and *query* is the AmosQL query string. *Sid* is the scan identifier returned to MATLAB.

Other functions such as *getElement(), newConnection(), closeAllConnections(), closeConnection(), printRow(), endOfScan(), scanWidth(), getElement(), nextRow(), freeScan(), freeAllScans(), finalizeSystem()* are common with MSCI.

4. Data Type Mappings

Both SSDM and MATLAB have C interfaces, and these are used for interfacing MATLAB and SSDM. This requires mapping the corresponding data types between MATLAB and SSDM, not least arrays represented by the *NMA* data type in SSDM since matrix is a central data type in MATLAB. This chapter mainly explains the data type mapping between MATLAB and SSDM, and the implementation of using C interfaces will be explained in Chapter 5.

MATLAB internally supports various kinds of arrays. All MATLAB variables are arrays, no matter what type of data. Numbers are represented as single element arrays. Even strings are arrays of characters. A matrix is a two dimensional numerical array. A *cell array* is an array containing any kinds of elements called *cells*. To enable data type interoperability between MATLAB and SSDM through MSL, most regular Amos II data types are mapped to corresponding MATLAB data types. The mapped regular data types defined in Amos II are respectively *real, integer, string, Boolean, time value, vector,* and *nil*. MSL provides MATLAB with ability to work with SSDM data types, some of which are not supported by MATLAB. For example, MSL defines RDF-specific types such as *URITYPE, UNISTRINGTYPE* and *TYPEDRDFTYPE*. These types are implemented by user defined MATLAB classes defined by MSL.

Inside Amos II, every data instance is assigned a handle (C type *oidtype*), regardless what type it is. A type tag associated with an Amos II handle identifies the data type of the data object in the database image that the handle represents. To enable data type interoperability between MATLAB and SSDM through MSL, most regular Amos II data types are mapped to corresponding MATLAB data types. *mxArray* is the data type representing all kinds of MATLAB data in C/C++ programs. The programs use MATLAB's *Matrix Library* [11] to work with the *mxArray* data structure.

Vector

A data type in SSDM (and Amos II), called *Vector*, has the ability to hold sequences of elements of different types. It is mapped to MATLAB's data type *cell array* represented as special kind of *mxArray* in C. When all elements in are numbers only, the vector is instead mapped to MATLAB's data type *matrix.*

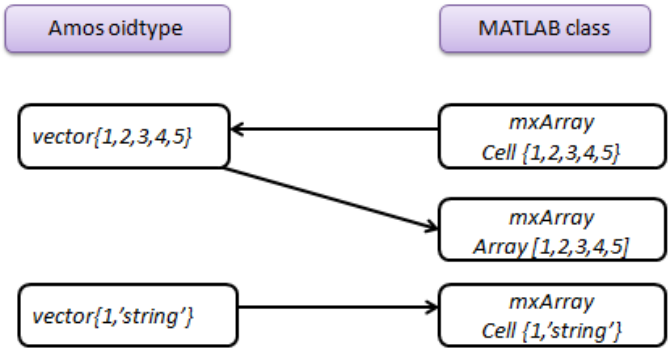

Fig.10 *Vector* data type mapping between SSDM and MATLAB

Time values

The *TIMEVALTYPE* in SSDM (and Amos II) is constructed as a structure with seven integer elements representing *year, month, day, hour, minute, second,* and *millisecond.* MATLAB represents time as a vector holding six numeric elements. For instance, an SSDM *time vector* is represented as [2012 12 12 0 0 0 99], while the corresponding MATLAB *time vector* is represented as [2012 12 12 0 00.099]. As we can see, milliseconds and seconds are stored as two integers in the SSDM *time vector*, while MATLAB represents

milliseconds and seconds in a single floating point number. To make a *time vector* distinct from the regular vectors in MATLAB, a MATLAB class called *TIMEVALTYPE* is created in MATLAB. As the graph shows below, the TIMEVALTYPE class contains one property called *TimeVector* which holds a *time vector* as value.

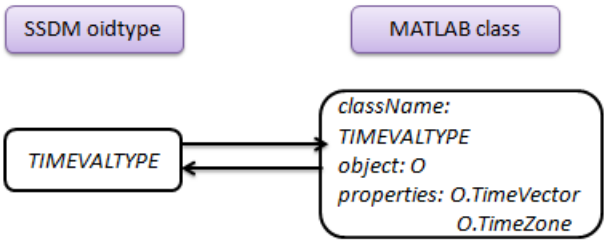

Fig.11 *TIMEVALTYPE* data type mapping between SSDM and MATLAB.

In MATLAB, this class is defined as:

```
classdef TIMEVALTYPE
    properties
        TimeVector = '';
       TimeZone = '';
    end
end
```

A MATLAB function, *makeTimeVal(timeVector)* is provided to construct *TIMEVALTYPE* objects. It takes one argument which is the *time vector*; a new *TIMEVALTYPE* object is created.

Numerical arrays

SSDM's data type *NMA* (numerical arrays) is mapped to MATLAB's native *array* data type represented by the *mxArray* in MSL. Data of Amos II type *vector* is mapped to an *mxArray matrix* if all elements in the list are single numbers; otherwise, an *mxArray* cell is created to hold instances of arbitrary data types. To map an *mxArray* to an Amos *ARRAYTYPE*, arguments from MATLAB workspace are required to be cell arrays. When using MCSI, MATLAB regular double matrices are mapped to SSDM *NMD* data type. The graph below shows the mapping between *NMA* data and different kinds of MATLAB *arrays.*

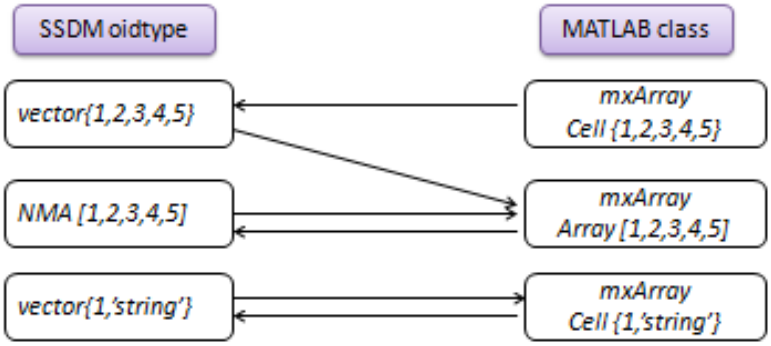

Fig12.Array datatype mapping between SSDM and MATLAB

Strings

RDF strings may have language and locale information. String values that represent literals in RDF are therefore represented in SSDM as the *Unistring* type, which has all necessary string properties.

Unistring objects containing language and/or locale information are represented by a new MATLAB class *UNISTRINGTYPE*, where an additional attribute called *LangTag*, is defined, as shown on Fig.13.

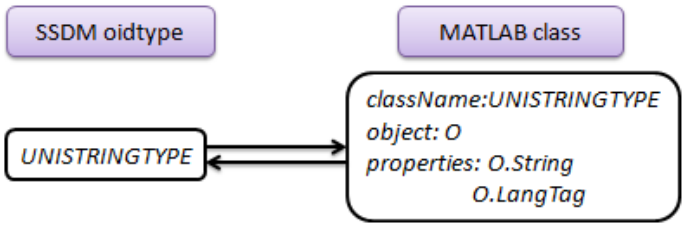

Fig.13 *UNISTRINGTYPE* data type mapping

Definition of class *UNISTRINGTYPE* in MATLAB:

```
classdef UNISTRINGTYPE
    properties
        String = '';
        LangTag = '';
    end
end
```

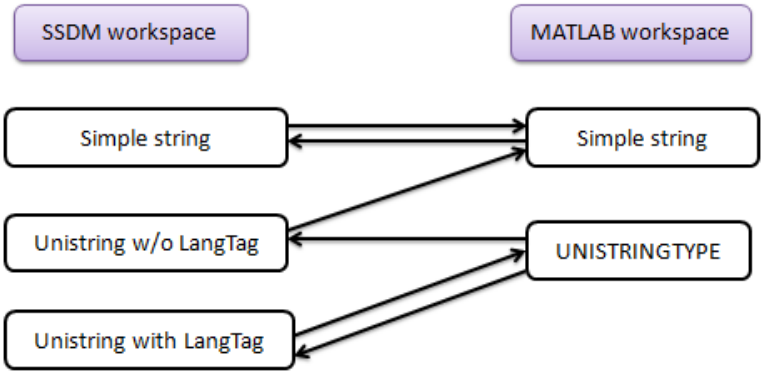

Fig.14 Mapping different string types

As shown on Fig. 14, instances of MATLAB *UNISTRINGTYPE* class are always mapped to SSDM *Unistrings objects*, either with or without the language tag. Simple SSDM string objects and *UNISTRINGTYPE* objects without *LangTag* are mapped to MATLAB's simple string type when no information will be lost; MATLAB *UNISTRINGTYPE* objects are anyway mapped to SSDM *Unistring* no matter whether the *LangTag* field holds information or not. For creating *UNISTRINGTYPE* values in MATLAB, the constructor function *makeUniStr(str, lang)* is defined. This function takes two arguments, *str* and *lang*, which are stored in *String* and *LangTag* properties respectively:

```
function [unistr] = makeUniStr(str,lang)
unistr = UNISTRINGTYPE;
unistr.String = str;
unistr.LangTag = lang;
end
```

URIs

A *URI* data instance is nothing else than a string with a special semantics and format, for example "http://www.uu.se". In order to distinguish URIs from strings, a class called *URITYPE* is defined in MATLAB. An object of this class is created to map *URITYPE* data from SSDM.

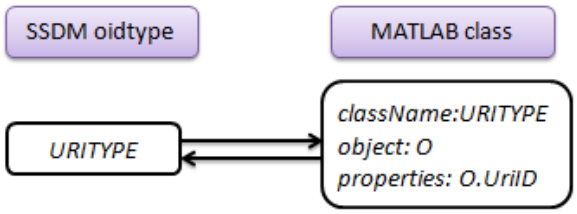

Fig.15 *URI* mapping

Definition of *URITYPE* class in MATLAB:

```
classdef URITYPE
    properties
        UriID = '';
    end
end
```

The URITYPE constructor is *makeUri(uriStr)*. It stores its string argument in the *UriID* property.

Typed RDF literals

A typed RDF literal is represented with two properties, *literal*, e.g. 'cat' and *type URI*, e.g. <http://example.org/ns#x00>. The corresponding data type in MATLAB is a new class *TYPEDRDFTYPE* with two properties *Literal* and *DataType*.

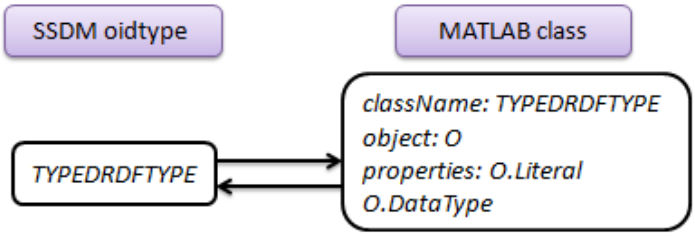

Fig.16 *Typed RDF literal* mapping

Definition of TYPEDRDFTYPE:

```
classdef TYPEDRDFTYPE
     properties
         Literal = '';
         DataType = '';
     end
 end
```

makeTypedRDF(literal, type) is the TYPEDRDFTYPE data constructor which takes two arguments that are stored in *Literal* and *DataType* properties.

5. Implementation

This chapter first introduces techniques used to implement MSL and the terminology used. The details of the implementation follow.

5.1 MATLAB – C interface

The MATLAB – C interface [10] and the Amos – C interface [5] provide us with the facility to implement interfaces like MSL and MAL, which allow MATLAB and Amos II engines to interact. This section explains the underlying MATLAB facilities used to implement MSL.

5.1.1 MATLAB shared library calls

The MATLAB external programming interface can call functions declared in a *shared library*, referred to as a *dynamic link library* (DLL) in Windows and *shared objects* in Unix, by calling *loadlibrary()* and *calllib()* MATLAB functions. Examples of how to call functions in a shared library are shown below. For this example let's assume there is a C interface file *mylib.h* file defining a C function, *int add(int a, int b)*, which is implemented by the DLL *mylib.dll*.

The DLL file is loaded by this MATLAB function call:
loadlibrary('mylib', 'mylib.h')

Once loaded the C function *add()* in the loaded library can be called by the MATLAB function *callib()*:
calllib ('mylib', 'add', 1, 2)

5.1.2 MEX API

MATLAB functions can be invoked from C using MATLAB's MEX API[18], e.g., *mexCallMATLAB()* that executes a MATLAB command and *mexPrintf()* that prints to the MATLAB console.

5.1.3 C/C++ Matrix Library

The interface between SSDM and MATLAB needs to passing data from one workspace to the other. In C/C++, a MATLAB array is declared to be of C structure *mxArray*, which in aStorage the corresponding data structures are represented by handles. The Matrix Library is used to access the *mxArray* structure. This API allows developers to create, read, and query information about the MATLAB data.

Arrays of specific types are defined inside the *mxArray* structure, which include *numeric arrays, char arrays, logical arrays, sparse arrays, structure arrays,* and *cell arrays.* Single numbers are stored as arrays of one element.

The following examples illustrate how to create *mxArray* data in a C program and free the memory after using them. The statements below declare two *mxArray* pointers and a double pointer which are assigned to variables *C, D,* and *D_elem*, respectively. Then the program allocates the memory for these two *mxArray* data

elements instances by allocating an *mxArray* data type. Finally the pointer to a floating point number *D_elem* is set to point to the first element of *mxArray D.*

```
mxArray *C,*D;
double *D_elem;
C = mxCreateCellMatrix(1, 1);
D = mxCreateDoubleMatrix(1, 1, mxREAL);
D_elem = mxGetPr(D);
```

The following statements set the value of the first field of the *mxArray* cell array *C* to the matrix *D* by calling *mxSetCell()* with index, 0, and the value *D*. After that, all *mxArray* data instances are destroyed by calling *mxDestroyArray()* function.

```
mxSetCell(C, 0, D);
mxDestoryArray(D);
mxDestroyArray(C);
```

mxArray data instances can be created by calling functions with prefix, *mxCreate* followed by a MATLAB data type , e.g., *mxCreateString.* These functions dynamically allocate memory for the created *mxArray* data instances. As we can see from the code above, two *mxArray* data instances were represented by the types *cell array* and *double array*. *mxCreateCell* allocates memory for the *cell array C,* which holds one field, in this case. *mxCreateDoubleMatrix* allocates memory for *double matrix, D,* which contains only one *double* element. To free memory after using these two *mxArray* data instances, *mxDestoryArray* is called to deallocate the memory occupied by the specified *mxArray* data instances.

5.2 Data conversions

In MAL, two functions, *amos_make_mx(oidtype)* and *amos_make_oid(mxArray*),* enable data exchange between Amos II database images and the MATLAB workspace; in MCSI, two functions, *make_mx(oidtype)* and *make_oid(mxArray*),* handle the data conversion between MATLAB and SSDM. The graph below shows the data mapping done by MCSI and the responsible function along the arcs:

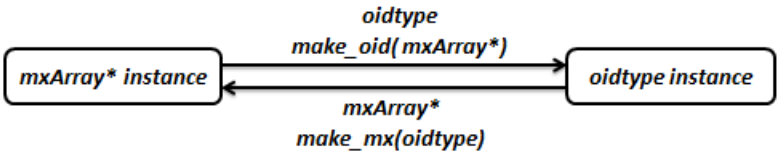

Fig.17 Mappings between *mxArray* and *oidtype* data types

Converting from Amos II to MATLAB

The graph below shows functions for data conversion from C data structure *oidtype* to *mxArray*. The function *amos_make_mx(oidtype)* is responsible for converting the Amos II database object identified by a handle into the corresponding *mxArray* instance. *make_mx(oidtype)* is the function handling data conversion in MSL and it calls *amos_make_mx* when the data need to be mapped are of Amos II data type. Based on the handle type tag more specific functions convert different kinds of Amos II objects or SSDM objects to *mxArray*.

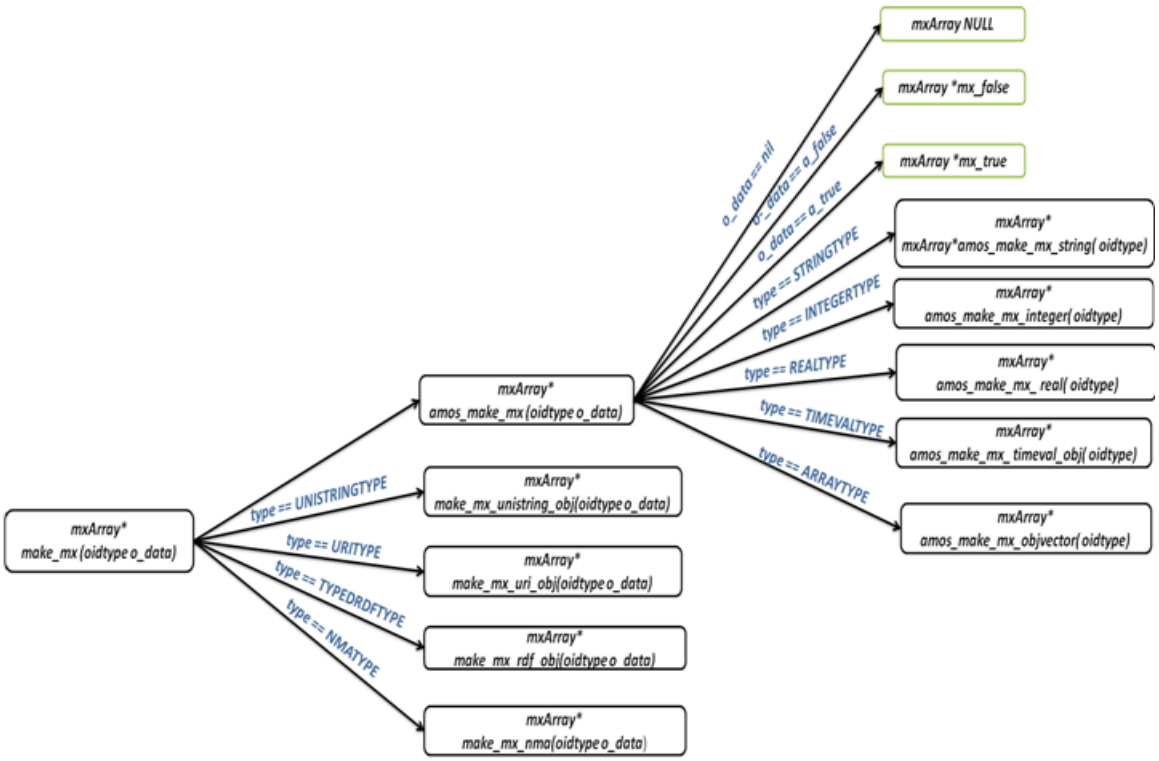

Fig.18 The call graph of *make_mx C* function

Converting from MATLAB to Amos II

amos_make_oid(mxArray)* function handles data conversion from MATLAB to Amos II and *make_oid(mxArray*)* function handles data conversion from MATAB to SSDM. They call specific functions for data mapping according to the MATLAB data type represented by the *mxArray* structure.

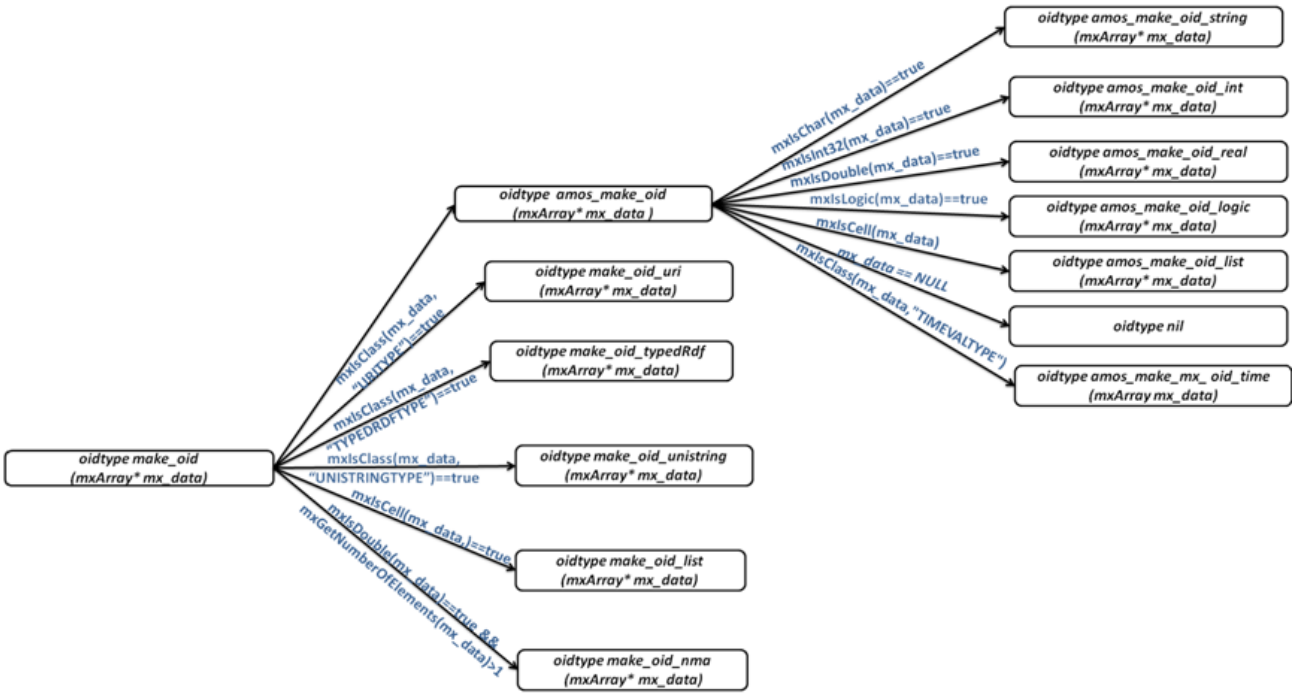

Fig.19 The call graph of *make_oid* C function

5.3 MATLAB - Amos Link (MAL)

MSL is built on top of MAL. There are six MAL C interface functions illustrated by Fig.17:

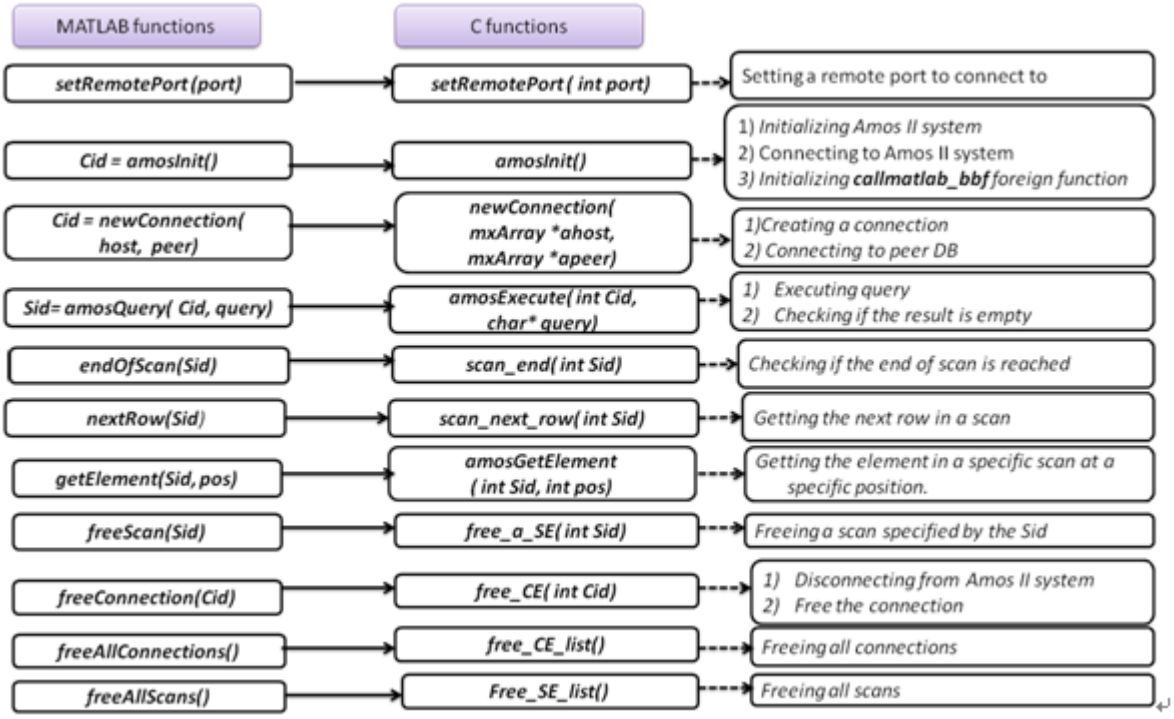

Fig.20 MAL C function and MATLAB function mappings

5.3.1 Global variables and structures

Two C structures, *ConnectEntry(CE)* and *ScanEntry(SE)* are defined to hold connections and scans, respectively.

ConnectionEntry (CE)

A *CE* is a structure extending Amos structure *a_connection* by adding an ID for each connection. The field *Cid* is holds the *CE* identifier. A *CE* includes *a_connection_rec* structure, which is a pointer to the next *CE*, a *Cid* identifying a connection, and a *host* name identifying which SSDM database server the *connection* connecting to. The attribute *sparqlFn* holds a handle to an Amos II function on the server that executes each SciSPARQL query received through the connection. Analogously, the attribute *rdfInsertFn* references an Amos II function to insert new RDF data triples in the SSDM server.

```
typedef struct ConnectEntry{
    a_connection Connection;
    int cid;
    char *host;
    struct ConnectEntry *next;
    oidtype sparqlFn, rdfInsertFn;
} CE, *cep;
```

ScanEntry (SE)

Amos II query results are always returned as scans. In order to interact with Amos II and access elements of tuples in a scan, MSL defines a structure *ScanEntry (SE)*, where an integer *Sid* identifies of the scan. This enables MATLAB to access scans by supplying a specific *Sid.* As the definition of *SE* shows below, every *SE*

has a field called *tpl* pointing to the current tuple in a scan. The first row of a scan is assigned to *tpl* when an *SE* is allocated after a query has been executed. The field *vectorized* is a tag to specify how the result was generated: by an AmosQL query, by a SciPARQL query, or by a SciSPARQL function call. A *vectorized* value 0 indicates that the scan holds result from either an AmosQL query or a SciSPARQL function, so that scan width is equal to the result width of query or a function. When this flag set to 1, it indicates that the scan holds the result from a SciSPARQL query, that is, internally, obtained by the call to *sparql()* function in Amos, and each result comes encapsulation in a vector.

```
typedef struct ScanEntry
{
    a_scan scan;
    int sid;
    int vectorized;
    a_tuple tpl;
    struct ScanEntry *next;
} SE, *sep;
```

Global Variables

In the client-server interface multiple connections to different SSDM databases are possible. The connections are maintained through some global C variables. The variables *CE_COUNTER* and *SE_COUNTER* hold the identity integer of the latest connection and scan identifier, respectively. They are initialized to 1.

```
int CE_COUNTER = 1;
int SE_COUNTER = 1;
```

*CE*s are stored as a list and there are three pointers that refer to the *CE* list. *CE_LIST_HEAD* points to the head of this list; *LAST_FOUND_CE* points to the latest *CE*, and *PRE_LAST_FOUND_CE* points to the previous *CE*, to make deleting a *CE* more efficient.

```
cep CE_LIST_HEAD=NULL;
cep LAST_FOUND_CE =NULL;
cep PRE_LAST_FOUND_CE=NULL;
```

Similar to the *CE* list, *SEs* are also stored in a list. Three pointers are reference the head of the *SE* list, the latest *SE*, and the previous *SE: SE_LIST_HEAD*, *LAST_FOUND_SE*, and *PRE_LAST_FOUND_SE*.

```
sep SE_LIST_HEAD=NULL;
sep LAST_FOUND_SE=NULL;
sep PRE_LAST_FOUND_SE=NULL;
```

5.4 The MATLAB Client/Server Interface (MCSI)

MCSI (MATLAB Client/Server Interface) is implemented on top of MAL, which uses the client-server interface of Amos II [c-interface] to communicate with the server. MATLAB data to be transmitted to the server is first converted into *aStorage* objects in a local main-memory area using *make_oid(mxArray*)*. The client/server communication facilities of Amos II are then used to send the local data objects to the server. When data arrives from the server they are first materialized in the local memory area before they are converted to MATLAB object by *make_mx(oidtype)*.

Fig.21 summarizes the C functions implementing MCSI calls from MATLAB. They are implemented in terms of *make_oid()* and *make_mx()*.

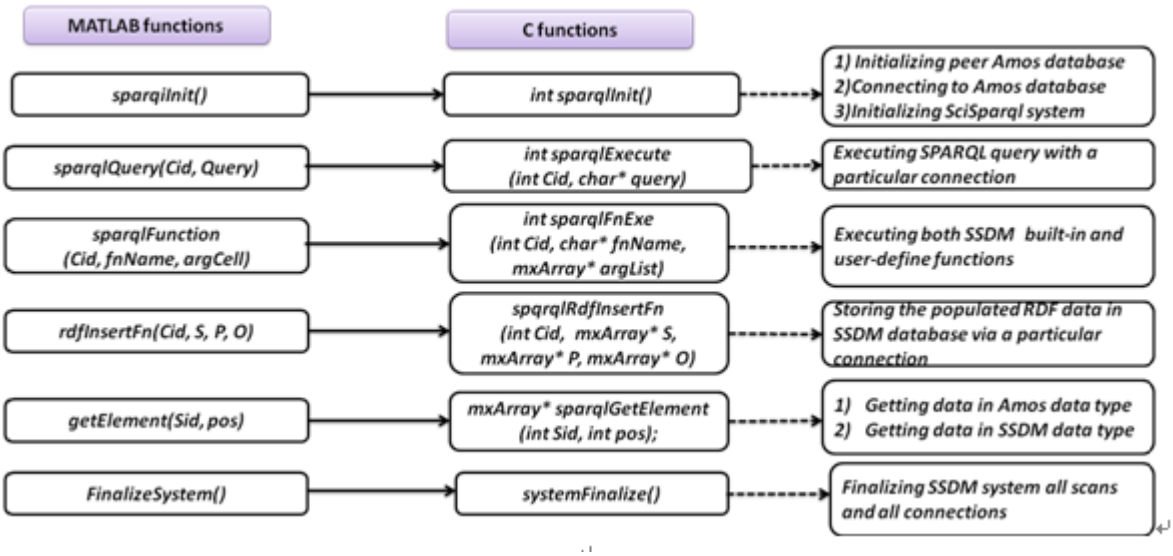

Fig.21 C functions called from MATLAB workspace

6. Conclusion and Future Work

MSL provides MATLAB users ability to access and query SSDM servers to manage RDF data. It supports a client/server architecture where connections can be established to SSDM database servers and data and directives exchanged between MATLAB and SSDM using the query language SciSPARQL. In order to pass data between MATLAB's workspace and the SSDM database server, the data has to be converted, due to the differences in storage representation. This requires data mappings between the main-memory representation of SSDM objects and corresponding MATLAB objects. In particular the SSDM data type *NMA* is converted to numerical MATLAB arrays.

The conversion of different kinds of data between the storage representations is currently copying all data objects. This can be slow for large objects. Future work includes avoiding such copying when possible. A complication is that the internal representation of arrays may differ between the different systems and programming languages, such as Python and MATLAB, which makes copying necessary when interfacing a programming language. How to avoid such copying is an area for future work.

References

[1] K.G.Clark and L.Feigenbaum, E.Torres: SPARQL Protocol for RDF, W3C Recommendation 15 January 2008. [Online], available: http://www.w3.org/TR/rdf-sparql-protocol/ , [accessed 10 March 2014]
[2] P.Hayes and B.McBride: RDF Semantics, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/, [accessed 10 March 2014]
[3] F Manola and E Miller: RDF Primer, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-primer/, [accessed 10 March 2014]
[4] A.Andrejev and T.Risch: Scientific SPARQL: Semantic Web Queries over Scientific Data, Data Engineering Workshops, 28th IEEE International Conference, Arlington, VA, 1-5 April 2012. [Online], available: http://www.it.uu.se/research/group/udbl/publ/desweb2012.pdf, [accessed 12 December 2013]
[5]T.Risch:AmosII External Interfaces , Uppsala, Sweden, 2001-01-24. [Online], available: http://user.it.uu.se/~torer/publ/external.pdf, [accessed 12 January 2014]
[6] T.Heath and C.Bizer(2011): Linked Data: Evolving the Web into a Global Data Space(1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool. [Online], available: http://linkeddatabook.com/editions/1.0/, [accessed 10 May 2014]
[7] [Online], available: http://www.mathworks.se/help/matlab/math/multidimensional-arrays.html, [accessed 10 March 2014].
[8] G.Klyne and J.Carroll: Resource Description Framework (RDF): Concepts and Abstract Data Model, W3C Working Draft, 29 August 2002. [Online], available: http://www.w3.org/TR/2002/WD-rdf-concepts-20020829/, [accessed 10 March 2014]
[9] [Online], available: http://www.mathworks.se/help/matlab/cell-arrays.htm, [accessed 10 March 2014]
[10] [Online], available: http://www.mathworks.se/help/matlab/external-interfaces.html, [accessed 10 March 2014].
[11] [Online], available: http://www.mathworks.se/help/matlab/cc-mx-matrix-library.html, [accessed 10 March 2014].
[12] [Online], available: http://www.mathworks.se/help/matlab/math/multidimensional-arrays.html#f1-86795, [accessed 10 March 2014].
[13] [Online], available: http://www.mathworks.se/help/matlab/ref/cat.html, [accessed 10 March 2014].
[14] Jeremy J. Carroll and Jeff Z. Pan: XML Schema Datatypes in RDF and OWL, W3C Working Group Note , 14 March 2006. [Online], available: http://www.w3.org/TR/swbp-xsch-datatypes/ , [accessed 10 December 2013]
[15] T.Risch: aStorage – a main memory storage manager, UDBL, Dept. of Information Technology, Uppsala University, Sweden, September 2009. [Online], available: http://user.it.uu.se/~torer/publ/aStorage.pdf, [accessed 12 Dec 2013]
[16] G.Fahl and T.Risch: Amos II Tutorial , Uppsala Database Laboratory, Department of Information Technology, Sweden, August 2008. [Online], available: http://www.it.uu.se/research/group/udbl/amos/doc/tut.pdf, [accessed 10 February 2014]
[17] [Online], available: http://www.mathworks.se/help/matlab/learn_matlab/matrices-and-arrays.html, [accessed 10 March 2014]
[18] [Online], available: http://www.mathworks.se/help/matlab/create-mex-files.html, [accessed, 10 March 2014]
[19] S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and E.Zeitler :Amos II Release 16 User's Manual, Uppsala DataBase Laboratory, February, 2014. [Online], available: http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html, [accessed 10 January 2014].