Appendix 1: Location-aware Resource Discovery Process

Our resource discovery solution identifies static and/or dynamic resources, and adapts several graph-based algorithms to traverse the Web resource graph (i.e., formed by the linked resources following the HATEOAS principle), while considering resources location (whenever they are exposed by objects). The algorithm to be used is given as input to the resource discovery process based on a library of algorithms (i.e., BFS or DFS in this work). Algorithm 1 presents the pseudo code of the defined discovery process having the following entry data:

- **algoType** (string): denotes the algorithm type to be used
- **f** (string): is the user requested function
- **k** (integer): is the maximum number (defined in user request) of the discovered resources providing identical functions

The output is the **discovered** array, containing the pairs [**f**, **id**] that correspond to the discovered resources necessary to realize f. During the discovery process, several main functions are used:

- **IdS(string,integer)**, identifies the data collection functions required for f and gets the relative resources corresponding to the specified location in r, using the geographic hierarchy and the indexing schema.
- **funResMap(array of [string, array of string])**, is called to generate an array of [string, string] that maps each resource to its provided function.
- **getResDesc(string)**, is called to access the description of a resource, identified by its id (i.e., URI), and retrieve the related resources that can be traversed next.
- **discover(array of string)**, traverses the resource Web graph starting from a set of identified resource to discover the rest of the resources necessary to realize f. Its pseudo code is presented in Algorithm 2.
- **functionMatch(string, string)**, checks if two functions are identical.

The discovery process (DP) starts by identifying the resources realizing the data collection functions necessary to f, and located in the required location, using the **IdS(f,k)** function (line 8). The data collection functions with their relative identified resources are added into the **dataCollectionRes** array.

Algorithm 1: Pseudo code of the Discovery Process (DP)

```
1  algoType string
2  input: f string
3  input: k integer
4  output: discovered array of [string, string]
5  dataCollectionRes: array of [string, array of string] // contains the data collection
       functions relative to f with their resources corresponding to the required
       location
6  resToExploreNext: array of string
7  discRes: array of [string, string]
8  dataCollectionRes = IdS(f,k) // gets k resources for each data collection function
       required for f in the necessary location
9  discovered = funResMap(dataCollectionRes)
10 foreach func in dataCollectionRes do
11     foreach id in dataCollectionRes[func][1] do
12         Descriptor desc = getResDesc(id) // get the description of a resource using
               its id
13         if not id.L.empty() then
14             resToExploreNext.insert(id.L) // insert the next related resources to
                   id, as defined in its description in resToExploreNext
15     if dataCollectionRes[func][1].empty() then
16         outputMessage ("no resources are identified in the required location")
17 discRes = discover(algoType, resToExploreNext) // start resource graph traversal
       from the resources in resToExploreNext to discover other required
       resources providing the necessary functions
18 discovered.insert(discRes)
19 return discovered
```

The **funResMap(dataCollectionRes)** function is used to map each function to its identified resource, and store the results in the **discovered** array (line 9). For each data collection function, func, in **dataCollectionRes**, DP retrieves the corresponding resource description (expressed in Hydra in this work) using the **getResDesc(id)** function (line 12) to get the linked resources ids that will be traversed next. These ids, stored in the **resToExploreNext** array (line 13), are later used by the **discover(algoType, resToExploreNext)** function, presented in Algorithm 2, to identify the resources providing the other required functions. When there are no resources providing a data collection function in the required location, a message will be sent (lines 15-16). Once the necessary resources are inserted into **resToExploreNext** array, they will be used by the **discover(algoType, resToExploreNext)** function to traverse the resource graph, and identify the rest of the resources necessary to answer f (line 17). All of the discovered resources realizing the required functions are saved in the returned **discovered** array (line 19).

Based on the given **algoType**, the discover function, presented in Algorithm 2, runs the corresponding algorithm (**runAlgoType(algoType)**) that will explore the Web resource graph starting from the resources included in the **resToExploreNext** array. **currentId** refers to the resource that is being processed, which is initially the first resource of the **resToExploreNext** array. For each unvisited resource, the algorithm gets the relative Hydra description through **getResDesc()** (line 12). If the operation function matches one of the functions in **F'** using the **functionMatch()** (line 14), the algorithm checks

Algorithm 2: Pseudo code of the discover(agloType, resToExploreNext) function

```
1  algoType string
2  input: resToExploreNext array of string // set of resources from which the Web
       resource graph traversal will start
3  output: discRes array of [string, string]
4  visited: array of string
5  F': array of string // contains the non data collection functions required to
       realize f as defined in FG
6  currentId = resToExploreNext[0]
7  F'= FunG(f) // Gets the required non data collection necessary to answer f
8  runAlgoType(algoType): // the execution of the algorithm corresponding to
       the given algoType
9  while not F'.empty() do
10     if not currentId in visited then
11         visited.insert(currentId)
12         Descriptor desc = getResDesc(currentId) foreach operation in desc.Operation
             do
13             foreach f in F' do
14                 if functionMatch(operation.function, f) then
15                     if not desc.RESD.empty() then
                           // the description contains a collection of dynamic
                              resources
16                         foreach adhoc in desc.RESD do
17                             discRes.insert([f, id])
18                             if resFound(discRes, f) = K then
19                                 F'.remove(f)
20                     else
21                         discRes.insert([f, currentId])
22                         if resFound(discRes, f) = K then
23                             F'.remove(f)
24         foreach link in desc.Link do
25             if (link.relationType = isSimilar or link.relationType = isComplementary
                 or link.relationType = isRelated) then
26                 resToTraverse.insert(link.entrypoint) // stores the resources linked
                       to the current traversed resource
27         currentId = resToTraverse.selectNext() // selects the next resource to
               traverse
28     else
29         currentId = resToExploreNext.next()
30 return discRes;
```

whether the resource is virtual (which are used to hold the connected dynamic resources) or static (lines 15 to 20). When it is virtual, the ids of the dynamic resources included in the description (line 16) are inserted with their relative function in the **discRes** array. When it is static, the corresponding resource id is stored in the **discRes** array with the relative function (line 21). If the number of the discovered resources realizing the current function is equal to **k** (lines 18 and 22), the function is removed from **F'**. Such number is calculated using the **resFound()** that we implemented apart. To explore other resources, the algorithm follows the resource annotated links (lines 24 to 26).