Modelling, Interlinking and Discovering Capabilities

Wassim Derguech
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway
Email:wassim.derguech@deri.org
www.deri.org

Sami Bhiri
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway
Email:sami.bhiri@deri.org
www.deri.org

Abstract—Even though the concept of capability is an important element in service oriented architectures and enterprise information systems, little effort has been put towards modelling it as a first class citizen. Major related contributions were part of other efforts such as modelling business processes, service description and search requests. In case of service descriptions, current approaches confuse capabilities with invocation interfaces or do not go beyond the classical IOPEs paradigm. Both approaches do not allow to determine intuitively what the service does. In our work, we are interested in modelling and managing capabilities as stand alone entities, presented via an action verb and a set of domain related attributes. Presenting capabilities as such allows us to represent them at different levels of abstraction and make explicit links between them. These links permit to create a direct acyclic graph. Given this graph, we provide in this paper a heuristic approach for capability discovery. We also present the needed algorithms for building and maintaining such structure.

Keywords—Reuse; Capability Description; Service;

I. INTRODUCTION

The concept of capability defines what an action can do from a functional perspective. An action can range from a simple task to a service or an entire business process. One of the objects of a capability description is to allow customers to discover services or business processes that perform particular functionalities that satisfy their needs. In our work, we address the problems related to capability descriptions and their discovery. In such context, a “good capability description” is a must either for allowing machine processing or human centricity.

The concept of capability is an important asset either in service computing or process management. Actually, it is even part of their definitions. Indeed, a service is an access mechanism for a certain capability and a business process is a set of ordered activities aiming to achieve a particular business goal. And here a business goal is a capability as perceived from a provider perspective. Although, this concept has not attracted as much attention as it deserves and it has been marginally modeled as part of other concepts such as service description, process modelling or search requests.

Currently, with the advent of the Semantic Web, ontologies and languages allowed for several ways to describe services [1, 2, 3, 4, 5, 6]. However, current approaches suffer from three main shortcomings. First, they consider capabilities as annotated invocation interfaces and not as functionalities. Second, capabilities are described at several levels of abstraction in terms of Inputs, Outputs, Preconditions and Effects; in such settings, to know what the service does users need to read its documentation. Third, there is no explicit link between these levels and a manual intervention is needed to determine concrete capabilities (i.e., capabilities that satisfy particular customer’s needs).

In our work, we plan to provide a meta model for describing capabilities as first class citizens and especially, our meta model should respect three main principles. First, a capability is described via domain specific features. Actually, we represent a capability as an action verb and a set of domain specific properties (i.e., attributes). The action verb as well as these attributes are defined in a domain related ontology that, to some extent, provides the possibles values each attribute can have. Second, capability offers can be generated automatically from the abstract descriptions. Indeed, in a recent work [7], we have defined a set of attribute types that allow for such feature. We have illustrated this via matchmaking scenario [7]. Third, capabilities are described at several levels of abstraction and explicit links between these levels are captured. In fact, we define links based on the attributes and attribute values of capabilities in order to establish a Capability Description Graph that draws essential links between capabilities.

We define, in this paper, the concept of Structured Entity in Section II. It allows for modelling any resources that can be defined as a set of attributes and we use it for describing capabilities. Actually, a capability is also an attribute-featured entity that has a particular mandatory attribute which is the action verb and a set of other attributes that define the various domain related properties. In this section, we define also two relations that may exist between structured entities needed to construct a structured entities graph. Describing capabilities as structured entities allows also for creating a Capability Description Graph that we use in the discovery process. Indeed, we provide a heuristic approach for capability discovery based on this graph. This technique takes as input a search request and provides
the exact capability if it exists in the graph otherwise it provides the $n$ most similar capabilities. Section III gives a detailed description of our discovery method. In addition, we provide the necessary algorithms for maintaining this Capability Description Graph. Section IV lists the required algorithms for creating, reducing and updating the graph. Finally, Section VI draws conclusion and future work after discussing and relating our work with existing approaches in Section V.

II. STRUCTURED ENTITIES GRAPH

In this section, we introduce the concept of structured entity and define a graph of structured entities. Actually, we use this concept of structured entity in order to model capabilities. In our work, we consider a capability as an attribute feature entity. This entity is defined via an action verb and a set of “attribute” and “value” pairs. The action verb is not a simple lexical term that gives a natural language indication about what a capability does. It is a concept from a domain related ontology that defines the semantics of the action of the capability and to some extent, it defines the required attributes and their possible values.

In this section, we present a structured entity as a general concept that can serve any conceptualization for modelling any attribute featured entities. Particularly, in our case, we apply it for modelling capabilities.

A. Structured entity

A structured entity is any object/thing which is described via a set of attributes. An attribute is a property (in RDF terms). But it is a special kind of property; it is an intrinsic property of the entity. For instance, name, date of birth, address, friend-of, work-in are all properties of an entity Person. Actually, the first three properties are intrinsic properties of a person. However, the last two express rather relationships with other entities (another person and an organisation).

We define the concept of attribute as a sub class of the class rdfs:Property. More formally:

\[
\text{se:attribute} \text{ rdfs:subClassOf } \text{rdfs:Property}\text{\footnote{rdfs: is the namespace prefix that refers to http://www.w3.org/2000/01/rdf-schema#}.}
\]

It is important to note that it is by misnomer and for presentation purposes we say that an attribute of an entity has a value. As explained above an attribute is a property and what we refer here as a value is the range of that property.

In our work, we consider this vision of modelling any structured entity by a set of attributes and apply it in modelling capabilities. In a previous work [7], we have presented in details the required attribute types for modelling capabilities. Definition 1 introduces the concept of capability in our meta-model that is considered as an attribute-featured entity.

**Definition 1.** (Capability) A tuple $\text{Cap} = (\text{ActionVerb}, \text{Attributes})$ is a capability, where:

- **ActionVerb:** This concept has been previously introduced by [3] in order to define, in a natural language, what is the action being described. Different to [3], we consider the action verb as a concept from a domain related ontology that comes form a shared agreement on its semantics and it also comes with the required attributes for a particular capability and, to some extent, it defines their possible values.
- **Attributes:** Represents a set of pairs (Attribute, AttributeValue) that correspond to the set of characteristics of the capability. An Attribute corresponds to a particular property of the capability and AttributeValue corresponds either to the value or the possible values that this Attribute can have.

We refer to [7] for a more detailed definition of the capability and the possible attribute value types that we have defined.

B. Relations between entities

We define different types of relations between entities. These relations exist based on the entity attributes or/and their values. There are coarse-grained and fine-grained relations. A fine-grained relation indicates which attribute the coarse-grained relation, it derives from, applies on.

In this paper, we will detail only the relation specify and extend (coarse and fine-grained versions). Before, we detail these relations, we need to introduce the relation variantOf that may exist between two entities. The relation variantOf has as domain and range rdf:Ressource\footnote{rdf: is the namespace prefix that refers to http://www.w3.org/1999/02/22-rdf-syntax-ns#}. More formally:

\[
\text{se:variantOf } \text{rdfs:range } \text{rdfs:Ressource} \\
\text{se:variantOf } \text{rdfs:domain } \text{rdfs:Ressource} \\
\text{se:variantOf } \text{rdfs:subClassOf } \text{rdfs:Property}
\]

The relation variantOf unifies, in fact, the two relations rdfs:subClassof and rdfs:type. More formally, for all resources $r_1$ and $r_2$:

\[
\text{r}_1 \text{ rdfs:subClassOf } \text{r}_2 \text{ OR } \text{r}_1 \text{ rdfs:type } \text{r}_2 \\
\Rightarrow \text{ r}_1 \text{ variantOf } \text{r}_2
\]

We do not distinguish between a sub class and an instance. In fact, as we consider these resources as classes, they might correspond to sets in set-theory. These relations (i.e., subclass and instance) are equivalent. Actually, a subclass relation corresponds to the subset relation and similarly for the instantiation. An instance is in fact a set of one element.

Before defining the relations specify and extend, we need to introduce a running example that will be used
for illustrating these relations. As it has been mentioned previously, we apply the notion of structured entity for representing capabilities and these relations between structured entities hold as well between capabilities. In such context, we will introduce a running example for representing delivery capabilities.

![Figure 1. Three delivery capabilities.](image)

In Figure 1, we present three delivery capabilities that we are going to use for illustrating specification and extension relations between them. This example presents three capabilities presented as structured entities. All these capabilities share the same attribute ActionVerb and its value “deliver”. ShippingCap1 consists of delivering a physical object within Europe whereas ShippingCap2 operates within Ireland and ShippingCap3 indicates a specific means of transport used for the delivery.

**The relation specify:** Let $e_1$ and $e_2$ be two entities, $e_2$ specify $e_1$ iff:

1. $e_1$ has exactly the same attributes of $e_2$ (in other terms $e_2$ inherits all the attributes defined in $e_1$),
2. for every (shared) attribute $at$, the value of $e_2.at$ is either equal to or variantOf the value of $e_1.at$, and
3. there exists at least one attribute $at'$ such that the value of $e_2.at'$ is variantOf the value of $e_1.at'$.

For instance, with reference to Figure 1, ShippingCap2 specify ShippingCap1. Actually, for the attributes From and To, ShippingCap2 has more specific values. Indeed $d:$Ireland variantOf $d:$Europe which leads to the relation: ShippingCap2 specify ShippingCap1.

Fine-grained relations can be defined based on the relation specify. Let $e_1$ and $e_2$ be two entities such that $e_2$ specify $e_1$, and let $at$ be a shared attribute, we say that $e_2$ specify $e_1$ on $at$, denoted $e_2$ specify$_{at}$ $e_1$ iff the value of $e_2.at$ is variantOf the value of $e_1.at$.

For instance, if we consider ShippingCap1 and ShippingCap2 of the Figure 1, we can derive the relations:

**ShippingCap2 specify$_{From}$ ShippingCap1** and **ShippingCap2 specify$_{To}$ ShippingCap1**.

**The relation extend:** Let $e_1$ and $e_2$ be two entities, $e_1$ extend $e_2$ iff:

1. $e_1$ has all the attributes of $e_2$ and has additional attributes, and
2. for every shared attribute, $at$, the value of $e_1.at$ is equal to the value of $e_2.at$.

For instance, with reference to Figure 1, ShippingCap3 extend ShippingCap2. Indeed, ShippingCap3 has an additional attribute (i.e., Vehicle). For the shared attributes, both entities have the same values. Consequently, we can establish the relation: ShippingCap3 extend ShippingCap1.

Fine-grained relations can be defined based on the relation extend. Let $e_1$ and $e_2$ be two entities such that $e_2$ extend $e_1$, we say that $e_2$ extend $e_1$ on the attribute $at$, denoted $e_2$ extend$_{at}$ $e_1$ iff $at$ is an attribute of $e_2$ and not of $e_1$.

For instance, if we consider ShippingCap2 and ShippingCap3 of the Figure 1, we can derive the relation:

**ShippingCap3 extend$_{Vehicle}$ ShippingCap2**

In order to automatically derive these relations given two structure entities, we define Algorithm 1. This algorithm determines the specify and extend operations that need to be applied for transforming a entity into another. Please note that we consider two additional relations: generalize and remove which are the opposite relations of specify and extend (i.e., if $e_1$ extend $e_2$ then $e_2$ remove $e_1$, similarly for specify and generalize).

### Algorithm 1: Computing the set of operations needed to transform a structured entity $e_1$ into $e_2$.

**Input:** StructuredEntity $e_1$, $e_2$: Two structured entities.

**Output:** Operations $op$: Set of operations needed to transform $e_1$ into $e_2$.

1. begin
2. **foreach** $Attribute \in (e_1.attributes \cap e_2.attributes)$ do
3. 1. if $SubType(e_1.Attribute,e_2.Attribute)$ then
4. 1.1. $op.Insert("generalize","Attribute.name");$
5. 1. end
6. if $SuperType(e_1.Attribute,e_2.Attribute)$ then
7. 1. $op.Insert("specify","Attribute.name");$
8. 1. end
9. end
10. **foreach** $(Attribute \in e_1.attributes)$ and $(Attribute \notin e_2.attributes)$ do
11. 1. $op.Insert("remove","Attribute.name");$
12. end
13. **foreach** $(Attribute \notin e_1.attributes)$ and $(Attribute \in e_2.attributes)$ do
14. 1. $op.Insert("extend","Attribute.name");$
15. end
16. return $op$;
17. end

Algorithm 1 takes as input two structured entities: $e_1$ and $e_2$, and returns the set of transformation operations $op$ needed to transform $e_1$ into $e_2$. This algorithm starts by checking the common attributes of $e_1$ and $e_2$ to verify if there are any specification (line 3) or generalization (line 6) operation. Then, for each attribute of $e_1$ that does not exist in $e_2$, it adds the corresponding remove operation (line 10). Finally, for each attribute of $e_2$ that does not exist in $e_1$, it adds the corresponding extension operation (line 13).
Using these relations between entities we can create their corresponding acyclic and directed graph. Nodes of this graph are the structured entities and their edges denote the specification and extension relations between them; these edges are annotated by the type of the relation. As an example of a graph, we refer to Figure 2 that illustrates a graph with the three capabilities introduced in Figure 1. It is important to notice that the relations specify and extend are exclusive. Actually, as a design choice, we can only have edges annotated with either specify or extend relations. We will see later in the paper how to deal with relations that contain both specifications and extensions.

In general, the concepts of structured entities and their graph are applicable for modelling any objects/things that can be described by a set of attributes and establishing explicit links between them. It is for example possible to use this conceptual model for describing a catalogue of products and create explicit similarity links between these products based on specification and extension relations as introduced previously or create other relations.

In our work, we apply such concepts for modelling capabilities. In a previous work [7], we have presented in details our conceptual model for capability description. In this paper, we build on top of such model some useful and interesting algorithms that allow us for the discovery of capabilities and for maintaining their corresponding structured entities graph. In the rest of the paper, what we call a capability, actually, refers to a structured entity and what we call capabilities description graph refers to a structured entities graph.

III. DISCOVERY OF A CAPABILITY FROM THE CAPABILITY DESCRIPTION GRAPH

As previously mentioned, a capability can be described as a set of attributes and actually it can be seen as a structured entity. Consequently, we can create its corresponding structured entities graph that we refer to it as Capabilities Description Graph (CDG for short). Having such structure in hand is very useful to perform operations such as discovery of a particular capability. In this section, we present a heuristic algorithm that allows to discover the n most similar capabilities to the one given as a request. For such purpose, we start by providing our similarity metric that computes the similarity degree between two capabilities.

**Similarity degree between two capabilities:** Given two capabilities Cap1 and Cap2, a similarity degree between them gives an idea about the likeness of their attributes. To this end we rely exclusively on the attributes and their values that define each capability. Equation 1 defines our similarity measure given two capabilities:

\[
\text{Similarity}(\text{Cap}_1, \text{Cap}_2) = \frac{||\text{Cap}_1.\text{Att} \cap \text{Cap}_2.\text{Att}||}{||\text{Cap}_1.\text{Att} \cup \text{Cap}_2.\text{Att}||} + \beta \times ||\text{SpecRelations}|| \tag{1}
\]

This measure (i.e., Equation 1), allows for computing the similarity degree between two capabilities; knowing that they are both described as structured entities. The numerator corresponds to the number of common attributes between capabilities by considering even those that have different values. The denominator, contains two parameters: the first one relates to the total number of attributes union of both capabilities attributes; and the second one serves to penalize this measure if there are any specification relations between the capabilities. \( \beta \) is penalty weighting parameter. \( \beta = 1 \) if all the specification relations are from one capability to the other. \( \beta = 2 \) if there are specification relations in both directions between both capabilities. With such equation, if two capabilities do not share any attribute, the numerator is going to be 0 and consequently the similarity measure would be 0%. If both capabilities are exactly the same, then \( \text{SpecificationRelations} = 0 \) which makes both numerator and denominator equal and consequently the similarity measure would be 100%. Referring back to Figure 1, the similarity measure between ShippingCap1 and ShippingCap3 is:

\[
\text{Similarity}(\text{ShippingCap1}, \text{ShippingCap3}) = \frac{3}{(4 + 1 + 2)} = 50\%
\]

We use this similarity measure in order to discover a particular capability. A first intuitive algorithm would consist of checking the similarity degree between a request expressed as a structured entity and all the capabilities in a repository. If there is no exact matching, the n most similar capabilities would be presented to the requester. Such method is quit simple and easy to implement if the number of capabilities in a repository is reasonable. Otherwise, we propose in the rest of this section a heuristic algorithm that allows, given a request, to discover its n most similar capabilities given the capabilities description graph.

**Discovery of the n most similar nodes in the CDG to a request:** Here we introduce Algorithm 2 that searches for n most similar nodes, according to a heuristic approach, in

---

Please note that for presentation purposes we use Att to refer to Attributes and SpecRelations to refer to the specification relations.

Please note that we do not consider the action verb as an attribute in this equation. If two capabilities have two different action verbs their similarity score is 0.
the CDG to a request. It is important to note that the request is expressed as a structured entity. This algorithm takes as input a CDG, a Request, the number of needed nodes for a result and the number of iterations after which the algorithm stops in case he cannot find an exact node that matches the request. The algorithm returns a list of nodes ranked based on their similarity measure compared to the Request.

Algorithm 2: Discovery of a Capability given a Capability Description Graph.

Input: Graph CDG: the Capability Description Graph.
Request: request expressed as structured entity.
n: the number of returned (required) nodes (i.e., capabilities).
i: the number of iterations after which the algorithms stops.

Output: Capabilities OrderedList: the set of discovered capabilities.

begin
Capabilities Candidates, Neighbors;
Capability Neighbor, Node, CurrentNode;
Integer counter;
Candidates = RandomlySelect(n, CDG.E);
OrderedList = Order(Candidates);
while (counter ≤ i and Similarity(OrderedList(0), Request) ≤ 100) do
counter = counter + 1;
foreach (CurrentNode ∈ Candidates and ¬ CurrentNode.Visited) do
Neighbors = getNeighbors(CurrentNode, GDG);
foreach (Neighbor ∈ Neighbors) do
if Similarity(CurrentNode, Request) < Similarity(Neighbor, Request) then
Candidates.Add(Neighbor);
OrderedList = Order(Candidates);
end
OrderedList = Order(Candidates);
end
return OrderedList;

end

Algorithm 2 operates as follows: It starts by randomly selecting n nodes of the CDG and adding them to a list of Candidates (see line 5). Then it orders these candidates in OrderedList based on their similarity measure to the Request. This OrderedList contains only the top n nodes. If the first element of OrderedList has a 100% similarity to the request, then the algorithm terminates and returns the current OrderedList. Otherwise, the algorithm compares the similarity measure of a node from Candidates and its neighbors. If any of the Neighbors ameliorates this measure, it is then added as a Candidate and the OrderedList is consequently updated. The algorithm terminates if all the nodes of Candidates have been examined or if a 100% match exists.

To be sure that the algorithm operates efficiently, it is required that the Capability Descriptions Graph is well formed. By well formed we mean that neighbor nodes are quite similar and that is why we have decided to consider only links with homogeneous operations (i.e., exclusively either specify or extend relations). As we are using a heuristic approach here, it is quite common that the n provided nodes are not the most similar ones that might exist in the graph. And one possible way to improve the algorithm is to consider a certain threshold that allows to consider neighbors that even do not improve the similarity measure but not exceeding this threshold.

IV. MAINTENANCE OF THE CAPABILITY DESCRIPTION GRAPH

In this section, we provide: first, an algorithm that allows for creating such structure having as input a set of capabilities (see Section IV-A); second, an algorithm that allows to add a node to an existing CDG (see Section IV-B).

A. Building The Capability Descriptions Graph

In order to create the Capability Description Graph (CDG for short), we need to check all capability descriptions and represent in a graph their relations in terms of transformation operations. This is done in a two step operation: starting by creating an initial CDG then reducing the obtained graph.

Building the Initial Graph: To create the CDG, we need to compute its set of nodes N and edges E. For the nodes N, it is obvious that it contains all capability descriptions. For the edges E, we can take each pair of capabilities and computes the transformation operations by using Algorithm 1.

Remember that Algorithm 1 provides even remove and generalize operations, however, we are interested only in specify and extend relations. Then, we simply create an edge for each pair of capabilities if only specification or extension relations exist. These steps are described in Algorithm 3.

Algorithm 3: Constructing an initial Capability Description Graph.

Input: Capabilities capSet: A set of capabilities described as attribute-featured entities.

Output: Graph CDG: the Capability Description Graph.

begin
Capability cap1, cap2;
Operations op;
CDG.N = capSet;
foreach cap1 ∈ (capSet) do
foreach cap2 ∈ (capSet) do
op = AlgorithmTransformation(cap1, cap2);
if ~(op.contains("Remove") or op.contains("Generalize")) then
CDG.E = CDG.E ∪ (cap2, op, cap1);
end
end
return CDG;

end

The CDG delivered by Algorithm 3 contains specification and extension relations that may exist between each pair of capabilities which can result in having duplicate links.
that can be obtained via transitive relations. However, we are more interested in having a compact representation of this CDG and we propose to reduce the possible transitive relations. A transitive relation is a relation that might be obtained by intermediate relations between other capabilities. We use Algorithm 4 in order to keep only direct relations and remove those that can be obtained through transitivity.

**Reducing the Capability Description Graph:** Algorithm 4 allows for reducing the CDG with respect to two main objectives: (i) reducing the graph by removing edges that can be obtained by transitivity (see lines 5-14) and (ii) keeping only edges with homogeneous operations in the Relations field (see lines 15-24). The last step of this algorithm is the removal of any duplicate edges (see line 25) that consists of checking any pair of edges; if both source and destination are the same, then one of the edges is deleted from the CDG.

**Algorithm 4:** Reducing the Capability Description Graph.

<table>
<thead>
<tr>
<th>Input: Graph CDG: Capability Description Graph.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. begin</td>
</tr>
<tr>
<td>2. Capability cap1, cap2;</td>
</tr>
<tr>
<td>3. Operations op;</td>
</tr>
<tr>
<td>4. Integer NumOP = 1;</td>
</tr>
<tr>
<td>5. foreach Edge1 ∈ CDG do</td>
</tr>
<tr>
<td>6.     foreach Edge2 ∈ CDG do</td>
</tr>
<tr>
<td>7.         if Edge1.Destination = Edge2.Destination and</td>
</tr>
<tr>
<td>8.             Edge1.Operations ⊆ Edge2.Operations then</td>
</tr>
<tr>
<td>10.                Edge2.Destination = Edge1.Source;</td>
</tr>
<tr>
<td>11.                RecursivelyCallTheSameAlgorithm(CDG);</td>
</tr>
<tr>
<td>12.                Exit();</td>
</tr>
<tr>
<td>13. end</td>
</tr>
<tr>
<td>14. end</td>
</tr>
<tr>
<td>15. foreach Edge ∈ CDG do</td>
</tr>
<tr>
<td>16.         if &quot;Specify_&quot; ∈ Edge.Operations and &quot;Extend_&quot; ∈</td>
</tr>
<tr>
<td>17.             Edge.Operations then</td>
</tr>
<tr>
<td>18.                 CreateNewCapability(Cap1);</td>
</tr>
<tr>
<td>19.                 CreateNewCapability(Cap2);</td>
</tr>
<tr>
<td>20.                 AddNewEdge((Edge.Source,Edge.getSpecifyOperations(),Cap1) );</td>
</tr>
<tr>
<td>21.                 AddNewEdge((Edge.Source,Edge.getExtendOperations(),Cap2) );</td>
</tr>
<tr>
<td>22.                 AddNewEdge((Cap1,Edge.getExtendOperations(),Edge.Destination));</td>
</tr>
<tr>
<td>23.                 AddNewEdge((Cap2,Edge.getSpecifyOperations(),Edge.Destination));</td>
</tr>
<tr>
<td>24. end</td>
</tr>
<tr>
<td>25. RemoveDuplicateEdges(CDG);</td>
</tr>
<tr>
<td>26. end</td>
</tr>
</tbody>
</table>

B. Updating the Capability Descriptions Graph When Adding a New Capability

We provide in this section a method that consists using a heuristic approach using the similarity measure introduced previously combined with the building algorithm. Actually, at a first step, we can discover the $n$ most similar nodes, according to our heuristic method introduced in Section III, in the CDG to the new node. At a second step, we can apply the previously introduced algorithm (see Section IV-A) between the new node and its $n$ nodes given by the first step.

In the first step, we use the discovery algorithm presented in Section III. If it provides a node with an exact matching (i.e., 100% similarity), we simply check if the corresponding node is concrete and we stop this operation by indicating that this capability exists in the CDG. If the corresponding node is not concrete, we change its status into concrete and stop this operation by indicating that the node has been inserted successfully. Otherwise we move to the second step.

The second step of our algorithm is applied only if the similarity algorithm did not find any node with 100% similarity degree. As mentioned previously, this second step consists of operating in a similar way as in Section IV-A:

1) Using Algorithm 1 in order to determine all the transformation operations between NewNode and all the nodes returned from the similarity algorithm (i.e., OrderedList).

2) Update CDG.E (i.e., the set of edges of CDG).

3) Apply Algorithm 4 for reducing CDG by removing any edges that can be found by transitivity.

There is an alternative method for adding a new node to the Capability Descriptions Graph (CDG for short), by using the algorithm defined in Section IV-A: one can consider all the possible relations with the existing nodes of the CDG with the new node. Such solution can be a very costly operation in terms of complexity if there is huge number of capabilities given as input. Indeed, computing all the possible transformation operations from one capability to another for building the initial graph has a complexity of $(N \times M)^2$ where $N$ is the number of capabilities and $M$ is the maximum number of attributes that a capability has.

V. RELATED WORK

The Semantic Web uses ontologies and languages allowing several ways to describe web services. For example WSDL-S\(^6\) and its successor SA-WSDL\(^1\) use ontologies to enrich WSDL description and XML Schema of Web services with semantic annotations. Such techniques consider a capability as an invocation interface. However, as explained in this paper a capability is not an interface. It is an entity featured via a set of attributes. We believe that our vision and understanding of capability is more accurate. The background of the genesis of SOA, which is distributed method invocation, has influenced these techniques.

In a more refined fashion, languages such as OWL-S\(^8\), WSMO\(^9\), SWSO\(^7\), provide a semantic description of Web services. However, these approaches do not go beyond the

---

\(^6\)http://www.w3.org/Submission/WSDL-S.
\(^7\)http://www.w3.org/Submission/SWSF-SWSO.
classical Input, Output, Precondition and Effect paradigm to define services capabilities. They do not feature the business aspects of a capability. In addition, they describe capabilities at an abstract level. They are not able to model concrete capabilities that correspond to specific needs of consumers. However, what clients are interested in are concrete capabilities. The matching of consumer requests has to be against concrete capabilities.

Oaks et al., [3] propose a model for describing service capabilities going one step beyond the IOPE paradigm by distinguishing in particular the corresponding action verb and informational attributes (called roles in the paper [3]). However, the semantics of capabilities remain defined via the IOPE paradigm and therefore has the same problems of the previously described approaches.

The notion of (service) offer is currently not addressed in service description languages such as Universal Service Description Language (USDL\(^8\)). Service descriptions have a varying level of concreteness depending on the stage of a matchmaking and configuration. This notion is currently missing from the USDL. Introducing different levels of concreteness in USDL Functional Module should make USDL applicable for describing highly configurable services.

Other related works worth mentioning are [4, 5, 6]. These works have identified the gap between current modelling techniques and real world requirements and initiated the discussions about abstract services and concrete offers descriptions. Similar to all related work, the concept of capability was not tackled as first-class entity. The focus was rather on the service model. In addition, [5] and [6] rely on and extend the Input, Output, Precondition and Effect paradigm without making explicit and clear the business features of services functionalities. They also do not explicitly distinguish between abstract services and service offer, nor do they define the links between them.

VI. CONCLUSIONS

We defined, in this paper, the concept of Structured Entity which is generic enough to be used for modelling any attribute featured resource. Particularly, we use such concept for modeling capabilities. Actually, a capability is an attribute-featured entity defined via an action verb and a set of domain related attributes. The action verb, the attributes and their values are defined in a domain related ontology such that the modelling of capabilities can be controlled. Describing capabilities as such has the advantage of being user-centric. Indeed, the action verb helps to determine the semantic of the action performed by the capability and having domain related attributes also facilitates to determine the exact functionality of the concerned capability.

Additionally, we defined, in this paper, two relations that may exist between capabilities needed to construct a capability descriptions graph. Given this graph, we provide a heuristic approach for capability discovery. This technique provides the exact capability if it exists in the graph otherwise it provides the \( n \) most similar capabilities, similar to query relation techniques. Furthermore, we provided the necessary algorithms for maintaining this Capability Description Graph: creation, reduction and update.

Our approach is useful for reuse oriented capability modelling. In fact, modelling a new capability can be done through navigating the graph and extending one of its nodes in order to create a new capability.

As part of our future work, we plan to investigate further relations that might be useful for the capability descriptions graph. Additionally, we plan to provide the required environment for testing and evaluating our approach. And also we plan to define the required transformation algorithms that allow for generating (semi-)automatically service capabilities given their corresponding WSDL files.

ACKNOWLEDGMENTS

This work is funded by the Lion II project supported by Science Foundation Ireland under grant number 08/CE/I1380.

REFERENCES