SOCA
DOI 10.1007/s11761-011-0090-y

SPECIAL ISSUE PAPER

Weighted fuzzy clustering for capability-driven service aggregation

Christoph Dorn · Schahram Dustdar

Received: 1 April 2011 / Revised: 7 July 2011 / Accepted: 17 September 2011

Abstract Workflow design, mashup configuration, and composite service formation are examples where the capabilities of multiple simple services combined achieve a complex functionality. In this paper, we address the problem of limiting the number of required services that fulfill the required capabilities while exploiting the functional specialization of individual services. Our approach strikes a balance between finding one service that matches all required capabilities and having one service for each required capability. Specifically, we introduce a weighted fuzzy clustering algorithm that detects implicit service capability groups. The clustering algorithm considers capability importance and service fitness to support those capabilities. Evaluation based on a real-world data set successfully demonstrates the effectiveness of and applicability for service aggregation.

Keywords Fuzzy clustering · Service capabilities · Service aggregation

1 Introduction

Service aggregation describes the process of selecting a set of services to exploit their combined functionality. Applications such as designing a service mashup, developing a composite service, or determining services for a workflow require the selection and aggregation of suitable services. Existing work applies advanced mechanisms such as ontologies [23], goal models [9], context-based mediation [17], or QoS metrics [31] to determine the optimal set of services that provide the required functionality. These approaches can be roughly classified according to two extreme categories. Either each service is expected to provide exactly one of the operations or a single service provides all operations.

In this paper, we present an approach to service aggregation that resides between those extremes. The problem is finding implicit functional groups that are prevalent among the available services. An implicit functional group describes a set of operations that tend to be provided together on a service instance. One major challenge is automatically extracting those groups. Simultaneously, we also need to address the importance of each function in the overall aggregation as well as the actual support of each operation by the set of available service instances.

We apply the concept of capabilities to abstract from the technical service specification. Capabilities are metadata about a service that go beyond the pure interface description (WSDL) but remain more specific than QoS attributes. A storage service, for example, provides an *upload* operation and comes with corresponding capabilities that specify the maximum allowed file size and file count.

We compare required service capabilities and provided capabilities to obtain matching scores. QoS metrics filter out services that fail to offer the required performance requirements. Further pre-clustering analysis identifies generic capabilities that subsequently receive very low importance to ensure they do not distort the clustering result. The scores are then clustered to extract the implicit capability groups. After ranking the services in each cluster, we can select the top

This work has been partially supported by the EU STREP project Commius (FP7-213876) and Austrian Science Fund (FWF) J3068-N23.

C. Dorn (✉)
Institute for Software Research, University of California, Irvine, CA 92697-3455, USA
e-mail: cdorn@uci.edu

S. Dustdar
Distributed Systems Group, Vienna University of Technology, 1040 Vienna, Austria
e-mail: dustdar@infosys.tuwien.ac.at

scoring services for the aggregation. Additional aggregation constraints allow for specifying dependencies between service instances and service providers. The main focus, however, lies on the customization of the clustering algorithm. Subsequent challenges in the composition process such as interface mediation remain outside the scope of this article.

Our salient contributions in this article are:

- A modified fuzzy c-means (WFCM) clustering algorithm that is able to intelligently consider the importance and support of individual capability constraints when generating clusters.
- A mechanism for detecting and suppressing generic capabilities that would otherwise distort the clustering result.
- Integration of QoS metrics to define performance constraints.
- An algorithm determining the optimal service composition subject to aggregation constraints.

Evaluation based on a real-world data set successfully demonstrates the applicability and feasibility of our clustering algorithm. Our weighted fuzzy clustering technique provides consistently more sensible capability groups than the regular, non-weighted fuzzy c-means (FCM) clustering technique [1,2].

The remainder of this paper is structured as follows. Section 2 presents a motivating scenario to outline the problem and challenges in more detail followed by the approach in Sect. 3. Section 4 outlines the mechanisms for data transformation in preparation for clustering. Section 5 details the weighted fuzzy clustering algorithm. Section 6 presents the results of the clustering algorithm when applied to a real-world data set. Section 7 discusses related work before Sect. 8 concludes with an outlook on future work.

2 Motivating scenario

The advantage of extracting implicit capability groups instead of searching for a single, optimal mapping of capabilities to services is *replacability*. A capability group represents a considerable number of similarly structured services as otherwise the clustering algorithm would not determine such a group. This allows for run-time replacement of services within a group, while leaving the structure of the remaining service aggregation untouched.

We expect that capabilities are not uniformly distributed across services. A few capabilities will be available on most services, while other capabilities are very specific and provided only by a subset. Fuzzy clustering allows us to assign those generic capabilities to every cluster with some extent, while hard clustering (e.g., K-means clustering) would require to assign such capabilities to exactly one cluster. Unfortunately, fuzzy clustering alone does not guarantee that the generic capabilities always end up evenly spread cross all detected clusters. They might be distinct enough to create a cluster of their own.

We argue that the clustering algorithm needs to become aware of capability importance (i.e., constraint weights) and capability support (i.e., how well are capabilities provided by the service instances in terms of measured utility). Capability importance enables defining vital and optional capabilities. The clustering technique should produce clusters of specific functionality, rather than creating two types of clusters: the one type describes specific capabilities and the other type describes generic capabilities. Also, when encountering situations where some capabilities are hardly supported by available services, we want to avoid having two clusters emerge: one around the well-supported capabilities, and one containing all unsupported capabilities. The following scenario outlines these effects based on example data.

Let us suppose we have a set of 10 services ($S_1 \rightarrow S_{10}$) that match at least some of the desired 10 capabilities ($C_1 \rightarrow C_{10}$). Table 1 provides example utility values for each capability and service. Utility values close to 100 denote a (almost) perfect match of required capabilities and provided service capability. Values close to zero indicate that the service hardly fulfills the required capability.

Ideally, the clustering algorithm generates two clusters: one for capabilities $C_1 \rightarrow C_3$ and one for $C_4 \rightarrow C6$ (see Table 2 right side). As services hardly support capabilities C_6 and C_7, we want to avoid having these capabilities form a cluster and then select a service which barely fits. In contrast, basically all services provide C_9 and C_{10}. There is no need to have these form a distinct cluster as any services from another cluster is able to provide also these generic capabilities. Regular FCM clustering, however, produces the cluster configurations as provided in Table 2 left side, all of them unsuitable for efficient service aggregation. Two clusters split the capabilities into the set of best supported and the rest. Three clusters have the same overall effect, but distinguish between storage and communication capabilities. The right side of Table 1 lists the service ranking result for each of those three clusters. An aggregation of the top three services (S_7, S_8, S_2) would result in duplicating the efforts for providing capabilities $C_4 \rightarrow C_6$. As the ranking scores R_2 arising from cluster $K_{3,2}$ are comparatively high, a user might select also some lower ranking services (as they still yield high scores) and include one of S_4; S_5; S_6 to the aggregation. When increasing the cluster number to four, adaptation and general capabilities become mixed up. Results of five clusters and more become even more unusable and thus are not shown.

Table 1 Columns 3–12: utility matrix matching required capabilities and provided service capabilities from the scenario. Columns 13–15: ranking services when regular FCM produces three clusters

Capability description	Id	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	R_1	R_2	R_3
Task notification	C_1	80	90	50	0	0	0	5	5	0	0	S_7 (61.33)	S_8 (94.04)	S_2 (41.81)
Task delegation	C_2	50	70	90	0	0	0	0	0	0	0	S_8 (48.89)	S_{10} (89.94)	S_1 (40.87)
Blackboard	C_3	60	40	30	0	0	0	0	10	0	5	S_9 (43.86)	S_3 (88.52)	S_3 (38.65)
File versioning	C_4	0	0	0	0	5	0	80	90	70	60	S_{10} (33.56)	S_7 (87.07)	S_8 (6.55)
Storage flexibility	C_5	0	10	0	0	0	0	70	30	0	40	S_3 (6.27)	S_1 (86.00)	S_7 (3.95)
File size	C_6	0	0	10	0	8	0	70	50	80	20	S_2 (4.86)	S_4 (85.86)	S_{10} (3.43)
Location-awareness	C_7	7	0	10	0	5	3	0	10	0	4	S_5 (4.12)	S_9 (82.74)	S_9 (1.62)
Device-awareness	C_8	0	0	10	0	0	0	0	0	5	0	S_1 (2.41)	S_2 (77.17)	S_5 (0.91)
User accounts	C_9	100	80	87	90	80	80	90	95	87	99	S_6 (0.21)	S_5 (76.41)	S_6 (0.51)
Secure access	C_{10}	78	79	96	90	80	80	90	100	85	88	S_4 (0.02)	S_6 (76.34)	S_4 (0.01)

Table 2 Columns 2–11: scenario results for 2, 3, and 4 clusters with regular FCM. Columns 12–15: best cluster result with weighted, variable importance WFCM. Crisp cluster membership values in bold font

	τ	$K_{2,1}$	$K_{2,2}$	$K_{3,1}$	$K_{3,2}$	$K_{3,3}$	$K_{4,1}$	$K_{4,2}$	$K_{4,3}$	$K_{4,4}$	τ_v	ω	$K_{2,1}$	$K_{2,2}$
C_1	0.1	0.08	**0.92**	0.05	0.02	**0.93**	0.01	**0.98**	0.01	0.00	0.125	0.173	0.003	**0.997**
C_2	0.1	0.07	**0.93**	0.04	0.02	**0.94**	0.03	**0.96**	0.01	0.00	0.125	0.158	0.006	**0.994**
C_3	0.1	0.01	**0.99**	0.01	0.00	**0.99**	0.17	**0.80**	0.02	0.01	0.125	0.109	0.026	**0.974**
C_4	0.1	0.18	**0.82**	**0.96**	0.01	0.03	0.02	0.01	**0.97**	0.01	0.125	0.229	**0.997**	0.003
C_5	0.1	0.03	**0.97**	**0.85**	0.01	0.14	0.46	0.09	0.44	0.02	0.125	0.113	**0.893**	0.107
C_6	0.1	0.08	**0.92**	**0.97**	0.01	0.02	0.02	0.01	**0.97**	0.00	0.125	0.179	**0.993**	0.007
C_7	0.1	0.01	**0.99**	0.22	0.02	**0.76**	**1.00**	0.00	0.00	0.00	0.125	0.029	0.391	0.609
C_8	0.1	0.01	**0.99**	0.23	0.02	**0.75**	0.00	0.00	0.00	**1.00**	0.125	0.011	0.392	0.608
C_9	0.1	**1.00**	0.00	0.00	**1.00**	0.00	**0.99**	0.00	0.00	0.00	0.0001	0.001	0.565	0.435
C_{10}	0.1	**1.00**	0.00	0.00	**1.00**	0.00	0.00	0.00	0.00	**1.00**	0.0001	0.001	0.590	0.410

3 Approach

Capability-driven service aggregation starts with defining the required capabilities (Fig. 1). This includesparametrization of the utility functions and specification of QoS constraints. Pre-clustering QoS evaluation removes those services that fail to achieve the required QoS levels during an initial filtering of the available service capabilities. QoS requirements themselves are not included in the actual clustering process. Subsequently, utility calculation determines for each service and capability combination the corresponding utility value. The resulting utility matrix is input for adjustment according to user constraint weights and capability support by services. The weighted FCM clustering algorithm produces overlapping capability clusters. Each service is ranked within each cluster. The final aggregation of the top scoring service from each cluster is subject to post-clustering SLA constraints. An example aggregation condition specifies that service S_A and S_B must not be used in the same composition.

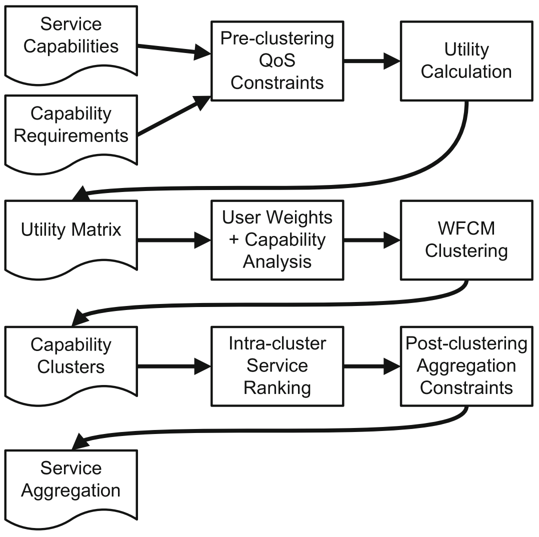
Service Capabilities
Capability Requirements
Pre-clustering QoS Constraints
Utility Calculation
Utility Matrix
User Weights + Capability Analysis
WFCM Clustering
Capability Clusters
Intra-cluster Service Ranking
Post-clustering Aggregation Constraints
Service Aggregation

Fig. 1 Capability-driven service clustering

4 Clustering preparations

4.1 Capability profiles and requirements

As briefly mentioned before, capabilities are metadata about a service that go beyond the pure interface description

```
1  <cap:Profile ProfileId="http://.../profiles?id=56879413">
2    <cap:WSDLlocation>http://example.org:8080/Storage_Service/service/ </cap:WSDLlocation>
3    <cap:Component ComponentId="http://example.org/Upload">
4      <cap:Capability>
5          <cap:CapabilityId>http://.../StorageTransfer</cap:CapabilityId>
6          <cap:Property PropertyId="http://.../StorageTransfer/MaxFileSize">
7                  <cap:value><cap:intValue>100</cap:intValue></cap:value>
8              </cap:Property>
9      </cap:Capability>
10     <cap:SelectableCapability
11         DefaultSelection="http://.../StorageCapability/Fixed" RequiredSelection="true">
12         <cap:CapabilityId>http://.../StorageDynamics</cap:CapabilityId>
13         <cap:Alternative>
14             <cap:CapabilityId>http://.../StorageCapability/Fixed</cap:CapabilityId>
15             <cap:Property PropertyId="http://.../StorageCapability/MinStorageSize">
16                 <cap:value><cap:intValue>1000</cap:intValue></cap:value>
17             </cap:Property>
18             <cap:Property PropertyId="http://.../StorageCapability/MaxStorageSize">
19                 <cap:value><cap:intValue>5000</cap:intValue></cap:value>
20             </cap:Property>
21         </cap:Alternative>
22         <cap:Alternative>
23             <cap:CapabilityId>http://.../StorageCapability/Dynamic</cap:CapabilityId>
24             [...]
25         </cap:Alternative>
26         <cap:Alternative>
27             <cap:CapabilityId>http://.../StorageCapability/Growing</cap:CapabilityId>
28             [...]
29         </cap:Alternative>
30         <cap:Alternative>
31             <cap:CapabilityId>http://.../StorageCapability/Shrinking</cap:CapabilityId>
32             [...]
33         </cap:Alternative>
34     </cap:SelectableCapability>
35     <cap:WSDLoperationScope>http://example.org/Storage_Service/service/storeFile</cap:WSDLoperationScope>
36   </cap:Component>
37   <cap:Component ComponentId="http://example.org/ContentManagementComponent">
38     <cap:QoSCapability>
39         <cap:CapabilityId>http://.../StorageContent</cap:CapabilityId>
40         <cap:Property PropertyId="http://.../StorageContent/MaxItemListSize">
41                 <cap:value><cap:intValue>200</cap:intValue></cap:value>
42             </cap:Property>
43         <cap:QoSProperty PropertyId="http://.../QoSMetric/ResponseTime">
44             <cap:value><cap:intValue>100</cap:intValue></cap:value>
45         </cap:QoSProperty>
46     </cap:QoSCapability>
47     <cap:WSDLoperationScope>http://example.org/Storage_Service/service/queryContent</cap:WSDLoperationScope>
48   </cap:Component>
49 </cap:Profile>
```

Listing 1 Capability profile excerpt containing two components which include one regular capability, one selectable capability, and one QoS capability

(WSDL) but remain more specific than QoS attributes. An example profile excerpt based on the scenario is provided in Listing 1. The profile describes storage services and defines exemplar capabilities on the maximum file size (lines 4–9), and the various strategies to dynamically adjust the available storage capacity (lines 10–34). Each selectable alternative capability provides the same properties on min and max storage limits (lines 15–20), and the values, however, may differ. We assume that service providers create a capability profile for each service and make this profile publicly available—similar to publishing the obligatory WSDL description. Currently such information is only available in human readable form (e.g., server hosting providers describing various packages of cpu power, available memory, maximum hard disk space). The interested reader is referred to [7] for a detailed discussion of the capability model, the specification of capability requirements, and the subsequent matching against capability profiles.

4.2 Pre-clustering QoS-based service filtering

QoS constraints are orthogonal to regular capability requirements as they describe the performance of a service rather than the limits of utilization. QoS criteria such as throughput, response time, or execution time are independent of the services purpose and equally apply to storage services and communication services. We thus distinguish how well a service does its job (QoS) and how to best use it, respectively well it fits the particular underlying purpose (Capability). The process of deriving the actual QoS metrics is out of scope of this paper. Existing research efforts such as by Rosenberg et al. [24,25] provide various techniques to measure and manage QoS metrics.

We extend our capability model to incorporate QoS metrics and introduce a new capability subclass: $QoSCapability$. Each $QoSCapability$ defines one or more $QoSProperties$. Listing 1 demonstrates how to define that a ContentManagementComponent delivers up to 200 item entries within 100 ms (lines 37–48). Defining the corresponding QoS constraint is then simply a matter of referencing the respective capability, relevant QoSMetric, and QoS utility evaluation parameters (see Listing 2 lines 28–40). The example constraint here considers every service below 200 ms of equally high performance and accepts degrading performance of up to 1,000 ms. Services beyond that threshold fail the QoS requirement. As QoS metrics are regarded as

```
 1 List reqList = new ArrayList<Requirement>();
 2
 3 TCapabilitySelectionRequirement req1 =
 4 RequirementsFactory.getSelectConstraint(
 5     "Set of acceptable storage adaptation strategies",
 6     "http://.../StorageDynamics",
 7     new String[]{http://.../StorageCapability/Growing,
 8                 http://.../StorageCapability/Fixed},
 9     new String[]{},
10     SetUtilityOne.class.getSimpleName(),
11     SetUtilityOne.UTILITY_TYPE,
12     0.6d);
13
14 TSimpleRequirement req2 =
15 RequirementsFactory.getConstraint(
16     "Minimum required transfer file size",
17     "http://.../StorageTransfer",
18     "http://.../StorageTransfer/MaxFileSize",
19     ValueUtility.UTILITY_TYPE,
20     ValueUtility.class.getSimpleName(),
21     new double[]{
22     new Unit(Mb,  80),
23     new Unit(Mb, 100),
24     new Unit(Mb, Double.MAX_VALUE),
25     new Unit(Mb, Double.MAX_VALUE)},
26     0.4d);
27
28 TQoSRequirement reg3 =
29 RequirementsFactory.getQoSConstraint(
30     "Minimum responsetime for file list retrieval",
31     "http://.../StorageContent",
32     QoSMetric.RESPONSETIME,
33     ValueUtility.UTILITY_TYPE,
34     ValueUtility.class.getSimpleName(),
35     new double[]{
36     new Unit(Millis, 0),
37     new Unit(Millis, 0),
38     new Unit(Millis, 200),
39     new Unit(Millis, 1000)},
40     1.0d);
41
42 reqList.add(req1);
43 reqList.add(req2);
44 reqList.add(req3);
45
46 QoSManager qos = new QoSManager();
47 qos.join(new Requirement[]{reg1, reg2});
48 qos.split(new Requirement[]{reg4, reg5});
```

Listing 2 Requirements on dynamic storage space adaptation, minimum transferable file size, and file list response time

independent, a service failing a single (out of potentially many) QoS requirement is no longer eligible for clustering.

In addition to performance requirements, service level agreements and other legal business aspects potentially constrain the formation of service compositions. Such aggregation constraints might restrict certain capabilities to be provided by the same service instance, forbid aggregation of two services from different providers, or limit the number of services within the composition. We discuss any restrictions that are dealt with before the clustering process here, and postpone the remaining description to Sect. 5.8. Currently we support the following pre-clustering aggregation constraints:

Collocated capabilities i.e., two or more capabilities need to be provided on the same service instance. We need to ensure that these capabilities do not end up in different clusters. Hence, we create a virtual capability that provides a single utility values based on the individual requirements and service capabilities. See Listing 2 line 47 as an example for joining two requirements in one virtual capability. Later on, only the virtual capability is used for clustering.

Separated capabilities i.e., two or more capabilities must not be provided by the same service instance. We need to ensure that these capabilities end up in different clusters. Any service that provides multiple of these capabilities is split into virtual services, one for each capability (see Listing 2 line 48 as an example). This reduces the likelihood of the separated capabilities ending up in the same cluster. There is an alternative if the capabilities still become collocated and when the aggregation size is not a concern. In this case, we select for each capability a separate service from that cluster, thereby increasing the number of aggregated services.

Aggregation size i.e., the aggregation must consist of minimal/maximal $\times$ services. The clustering process should not create too many or too few clusters (even though the cluster quality metric suggests to do so). For example, experience tells us to use at least three services, but the cluster quality is highest for two clusters. Thus, we simply limit the range of the cluster count z to observe this lower limit. Ultimately we accept the clustering result (with $z > 2$) that has the highest clustering quality.

4.3 Utility calculation

For the scope of this paper, it is sufficient to assume that there exists a matching function *eval* that maps each service $s_n \in \mathcal{S}$ and required capability $c \in \mathcal{C}$ to a utility value $u_{i,n}$ in the interval [0, 100], where 100 is the maximum score. The resulting utility matrix $\mathcal{U}$ contains for every combination of capability c_i and service s_n the corresponding utility value. Each row x_i in $\mathcal{U}$ describes the feature vector of capability c_i used for clustering.

The overall impact ω_i of capability c_i during the clustering process is given by the initial capability requirement importance τ_i (with $\sum \tau_i = 1$), and the support by the available services as measured by the fulfillment metric f_i.

$$\omega_i = \frac{f_i * \tau_i}{\sum f_i * \tau_i} \quad \text{where} \quad f_i = \frac{\sum_n u_{i,n}}{\sum u} \tag{1}$$

Subsequently, we ensure that services determine the clustering result proportional to their utility. To this end, we transform the utility matrix ($\mathcal{U}$) before clustering to reflect the preliminary service rank $r_n = \sum_i u_{i,n} * \tau_i \;/\; \sum r_n$. We multiply each $u_{i,n}$ with the service rank $r_n \in \mathcal{R}$ and normalize the matrix again such that services with average utility $u_i = \bar{u}$ maintain their utility value ($\mathcal{U}_\omega = \mathcal{U} \times \mathcal{R} \times |\mathcal{R}|$). We thereby exploit the FCM's sensitivity toward outliers. After the transformation, the weighted utility matrix $\mathcal{U}_\omega$ contains higher utility values for better ranked services compared with lower ranked services. Hence, above average services will have more impact during the clustering process as described in the following section.

5 Capability clustering

5.1 Fuzzy clustering basics

The underlying principle in fuzzy clustering is assigning data elements to multiple clusters with varying degree of membership. Algorithm 1 details the steps to obtain the best clustering for a particular number of clusters z. We refer to the listing when explaining the individual steps.

For our problem, the basic FCM [1,2] associates each capability feature vector x_i with every cluster $k_j \in \mathcal{K}$. The membership table $\mathcal{M}_{ij}$ describes the degree of capability x_i belonging to a particular cluster k_j, such that $\sum_j \mu_{ij} = 1$. Elements close to the cluster center (i.e., the centroid) have higher membership values for that particular cluster than elements farther away.

The clustering algorithm's objective is minimizing the overall distance of data elements to the cluster centers (line 11). This *within-class least squared-error function* is defined as:

$$J_m = \sum_{i=1}^{|\mathcal{C}|} \sum_{j=1}^{|\mathcal{K}|} \mu_{ij}^m * \|x_i, k_j\|^2 \tag{2}$$

where $m > 1$ is the fuzzy factor and $\| \bullet \|$ is a distance measurement between data element x and the cluster center k.

Algorithm 1 Weighted FCM clustering algorithm $WFCM(\mathcal{U}_\omega, z, m, \varepsilon, maxIt, \mathcal{W}, \beta)$.

```
1: function PERFORMCLUSTERING($\mathcal{U}_\omega, z, m, \varepsilon, maxIt, \mathcal{W}, \beta$)
2:     $\mathcal{M} \leftarrow$ initRandomMembership($\mathcal{U}_\omega, z$)
3:     /* Weight membership according to importance. */
4:     $\mathcal{M}_b \leftarrow \mathcal{M} * \mathcal{W}$
5:     $lastJ \leftarrow 0$
6:     for $round = 1 \ldots maxIt$ do
7:         $K \leftarrow$ calculateClusterCenters($z, m$)
8:         $\mathcal{M} \leftarrow$ updateClusterMembership($\mathcal{U}_\omega, \mathcal{K}, m$)
9:         /* Recalculating the cluster membership resets $\sum \mu_{ij} = 1$,
              therefore update membership again according to importance. */
10:        $\mathcal{M}_\omega \leftarrow \mathcal{M} * \mathcal{W}$
11:        $J$ = calculateObjectiveFct($\mathcal{U}_\omega, \mathcal{K}, m$)
12:        if $|J - lastJ| < \varepsilon$ then
13:            break
14:        else
15:            $lastJ \leftarrow J$
16:        end if
17:    end for
18:    $maxDist \leftarrow$ calculateTotalDistance($\mathcal{U}_\omega, \mathcal{W}$)
19:    calculateQuality($\mathcal{U}_\omega, \mathcal{W}, K, \beta, \mathcal{M}_\omega, maxDist, m$)
20:    $\mathcal{M} \leftarrow$ normalizeMembership($\mathcal{M}_\omega$)
21:    return $\mathcal{M}$
22: end function
```

Table 3 Fulfillment cluster center and size

	K_L	K_H
C_6		
Center	5.23	67.86
Size	7.05	2.95
C_9		
Center	83.18	96.57
Size	5.66	4.34

FCM iteratively recalculates cluster centers (line 7) and membership degree (line 8) until the objective function converges (line 12) $|J_m^t - J_m^{t-1}| < \varepsilon$ (where ε denotes the convergence limit) or until the maximum number of iterations $maxIt$ is reached (lines 6–17). For our purpose, the distance function is the euclidian distance, defined as:

$$\text{distance}(x, k) = \left(\sum_{d=1}^{n} |x(d) - k(d)|^2 \right)^{1/2} \tag{3}$$

where n is the dimension of both the capability vectors x and the centroid k, i.e., equal to the number of service candidates.

FCM applies the fuzzy factor m to define the crispness of membership degree. In general, high values of m implicate very fuzzy cluster boundaries whereas low values result in clear cluster limits. For m close to 1, FCM replicates the behavior of K-means clustering [14]. With $m = 2$, distance measurements are normalized linearly, and for $m \rightarrow \infty$, elements will belong to every cluster with equal degree.

5.2 Detecting generic capabilities

We automatically detect generic capabilities by analyzing the support distribution across all services. To test a singe capability c_i, we cluster the corresponding utility values ($\mathcal{U}_i$) into two groups: high support K_H and low support K_L. We then analyze the respective cluster centroids. A typical capability exhibits a clear separation. Capability C_6, for example, has $K_L = 5.23$ and $K_H = 67.86$ (see also Table 3). In addition, the size of K_H ($\sum \mu_H$) is smaller as K_L as usually only a subset of all services will support a particular capability. As for the generic capability C_9, both centers are comparatively high and also the clusters are of roughly equal size. We make use of these properties and define a capability as generic according to the following condition:

$$isGeneric = \begin{cases} \texttt{true} & \texttt{if } K_L > x \wedge K_H > x \\ \texttt{true} & \texttt{if } K_H > x \wedge \sum \mu_H > \sum \mu_L \\ \texttt{false} & \texttt{otherwise} \end{cases} \tag{4}$$

where x is a configurable threshold. A sensible value for x is 60 as any generic capability with both cluster centers below that threshold receives little to no boosting by the fulfillment metric f_i. The condition states that any capability is considered generic when both cluster centers $K_{H,L}$ are above the threshold or when size of K_H exceeds the size of K_L (given than K_H exceeds the threshold). Any generic capability that is not captured by these conditions receives medium to low support by the fulfillment metric f_i and thus is not likely to populate its own cluster but rather ends up assigned to multiple clusters. For our example, this clearly identifies C_9 and C_{10} as generic capabilities. In a second step, we decide whether to remove those from the clustering process or preferably opt to significantly reduce their weights (e.g., $\tau = 0.0001$).

5.3 Weighted fuzzy clustering

The basic FCM algorithm considers all data elements of equal importance. Thus when calculating the cluster centroid, the impact of a capability feature vector x is determined by its distance and membership degree. Data elements further away—thus having lower membership degree—yield lower impact on the center than closer elements. We adapt the FCM algorithm considering also the capability importance. Note that with $\sum \mu_i = 1$, every element is considered equally important. We therefore drop the condition that $\sum \mu_i = 1$. To this end, we weight the membership according to the importance vector $\mathcal{W}$ (line 4). After multiplying the membership table with the importance vector ($\mathcal{M}_\omega = \mathcal{M} \times \mathcal{W}$), less significant capabilities have little impact when calculating the cluster center as they yield a lower μ value. The weighted centroid k_j with importance vector $\mathcal{W}$ and fuzzy factor m is defined as:

$$k_j = \frac{\sum_i \mu_{ij,\omega}^m * x_i}{\sum_i \mu_{ij,\omega}^m} \tag{5}$$

The membership of a capability x belonging to a particular cluster k depends on the ratio of distance between x and k and the distance from x to all centroids $\mathcal{K}$:

$$\mu_{ij} = \left(\sum_{l=1}^{|K|} \left(\frac{\|x_i, k_j\|}{\|x_i, k_l\|} \right)^{2/(m-1)} \right)^{-1} \tag{6}$$

The reevaluation of membership degree in each iteration resets $\sum \mu_i = 1$, and hence, we need to recalculate the membership table in every iteration (line 10).

5.4 Cluster quality

The clustering procedure itself does not give any indication on the optimal number of clusters (the ‘c’ in FCM). Instead, we require some quality measurements to determine which number of clusters provides the most sensible results. A rule of thumb [16] recommends selecting $\max_z \approx i^{1/2}$ with i the number of data elements. A computationally more intensive approach calculates the clustering quality for increasing number of clusters until reaching maximum quality [29] propose a combination of cluster compactness and cluster separation for crisp clustering as a viable overall quality measure. Compactness describes how well the clusters explain the variance in the data. Cluster separation describes the heterogeneity between clusters. Clusters further apart exhibit more distinct elements than clusters close together.

First we update the definition of variance to account for weighted data elements. The weighted variance v_ω of a set of capability constraints and importance vector $\mathcal{W}$ is defined as:

$$v_\omega(\mathcal{U}_\omega) = \sqrt{\sum_{i=1}^{n} \left(\|x_i, \bar{x_\omega}\|^2 * \omega_i^2 \right) * \left(\sum_i \omega_i^2 \right)^{-1}} \tag{7}$$

$$\bar{x_\omega} = \sum_i \frac{x_i * \omega_i^{\varpi}}{\sum_i \omega_i^{\varpi}} \tag{8}$$

where $\|x_i, \bar{x_\omega}\|$ computes the distance of x_i to the weighted mean $\bar{x_\omega}$ of all elements in X. The less dispersed the elements, the smaller the variance.

Next, we compare the variance found in each cluster to the overall variance. We alter the definition of compactness cmp to consider fuzzyness:

$$\text{cmp} = \frac{1}{z} \sum_{j=1}^{z} \sqrt{\frac{\sum_i \mu_{ij} * \|x_i, k_j\|^2}{\sum_i \mu_{ij}}} * v_\omega(\mathcal{U}_\omega)^{-1} \tag{9}$$

where $\sqrt{\bullet}$ calculates the variance of elements weighted by their degree of membership in cluster k_j. Compactness is 1 for one cluster. With increasing clusters, the compactness eventually decreases to 0 at which point each data element resides in a separate cluster. We prefer lower compactness (i.e., clusters describe the variance increasingly well), but we need to avoid introducing too many clusters. To this end, we reuse the cluster separation metric by [29].

Separation is the coefficient of total pairwise distance between cluster centers and maximum possible distance. Separation reaches its maximum (sep $= 1$) when each cluster contains exactly one element. When one cluster comprises all elements, separation is zero. We update the function for calculating the total distance between constraints accordingly. Distance between important constraints gains significance, while distance between less important or mixed important elements has little effect on the overall distance.

$$\text{dist}_{\mathcal{U}} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \|x_i, x_j\|^2 * \frac{\omega_i + \omega_j}{2} \tag{10}$$

For cluster separation, we have to adapt the distance measurement between clusters. For each cluster, we compute the importance of the contained elements and apply the same weighted distance function as introduced above.

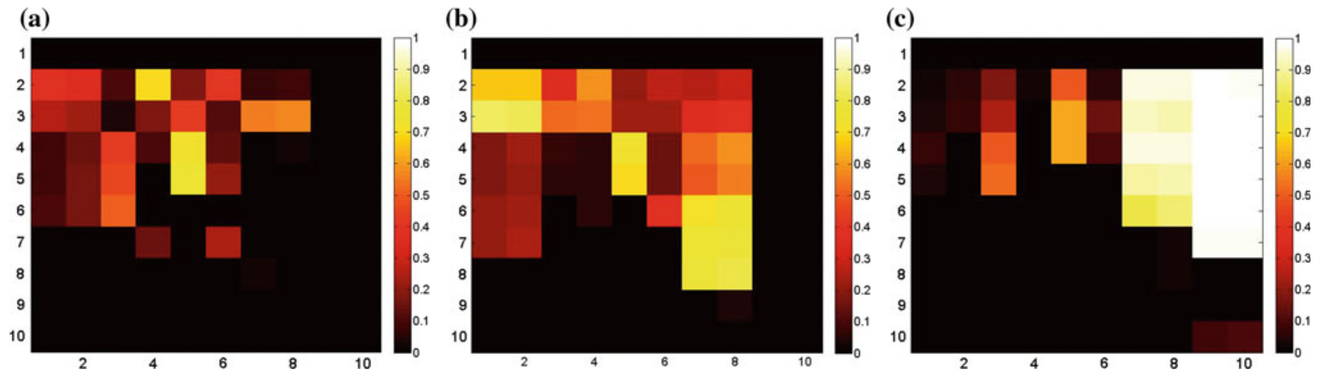

Fig. 2 Cluster entropy H_k for unweighted (**a**); weighted, equal importance (**b**); and weighted, variable importance (**c**) clustering. Capabilities 1 to 10 along the x-axis, cluster size z along the y-axis

$$\text{sep}_\omega = \sum_{j=1}^{z-1} \sum_{p=j+1}^{z} \left(\|k_i, k_j\|^2 * \frac{\sum_i (\omega_i * \mu_{ij}) + \sum_i (\omega_i * \mu_{ip})}{2} \right) * \text{dist}_{\mathcal{U}}^{-1} \tag{11}$$

where $\sum_i \omega_i * \mu_{ij}$ defines the importance of cluster j. The sum of pairwise distance between all elements yields computational complexity $\mathcal{O}(|C|^2)$. However, the distance remains unchanged for all iterations of cluster counts $z = 1 \ldots |C|$ and thus needs computation only once (line 18).

The combined metrics identify the maximum clustering quality. For one cluster, compactness equals 1 and separation equals 0. For all elements in individual clusters, compactness yields 0 and separation 1. The quality function $q(\beta)$ identifies the number of clusters that best describe the underlying distribution (line 19):

$$q(\beta) = 1 - (\beta * \text{cmp} + (1 - \beta) * \text{sep}) \tag{12}$$

where β defines a preference on compactness or separation. A β value below 0.5 assigns more weight on distinct clusters (sep) than on (lower) intra-cluster variance (cmp) and vice versa. The maximum quality value identifies the best number of clusters.

5.5 Effect of weights on cluster formation

One phenomena when applying the importance vector $\mathcal{W}$ is having the most significant capabilities rapidly split up into separate clusters. Clusters of less important capabilities form comparatively late.

Figure 2 compares normalized cluster entropy for unweighted (a); weighted, equal importance (b); and weighted, variable importance (c), with $m = 2$. Cluster entropy measures for each element the membership degree distribution across all available clusters. Low entropy values (dark colors) indicate focus on one or a few clusters. Bright colors highlight elements that (equally) belong to many clusters. Each column comprises the entropy values for a single capability. The top row contains the entropy values for $z = 1$ cluster, respectively the bottom row for $z = 10$ clusters.

The regular FCM algorithm places the general capabilities $9 + 10$ immediately into their own cluster while the remaining capabilities $1 \ldots 8$ display cluster membership of changing crispness. Rows 2, 3, and 4 in Fig. 2a result from the membership values in Table 2. Note the fluctuation of membership values between crisp and fuzzy as visualized by alternating bright and dark colors within a column.

In the weighted, equal importance case (Fig. 2b), we observe capabilities $9+10$ still exhibiting crisp membership across all clustering iterations. Hardly supported capabilities $7+8$, however, cease to occupy a separate cluster and display increasingly fuzzy membership values.

Finally, when correcting for the general capabilities in the weighted, variable importance case (Fig. 2c), capabilities $7 \ldots 10$ belong equally to an increasing number of cluster until after row seven, they all end up populating individual clusters. At the same time, capabilities $1 \ldots 6$ display comparatively crisp membership values also for low cluster count values ($1 < z < 4$).

5.6 Effect of weights on compactness and separation

Figure 3 displays compactness and separation for $m = [1.5; 2; 3]$ with unweighted, weighted equal importance, and weighted variable importance clustering side by side.

We notice an early, sharp decline in compactness opposed to a consistent increase in separation. Compactness is minimal when elements populate individual clusters. As observed above, the most significant elements quickly scatter into separate groups. If insignificant elements eventually occupy their own cluster, they barely reduce compactness.

The same effect causes a late steep incline of cluster separation when the weights clearly distinguish between distinct, well-supported capabilities and ill-supported or general capabilities. In the weighted, equal-importance case (Fig. 3b), capabilities $9 + 10$ dominate the importance subsequently

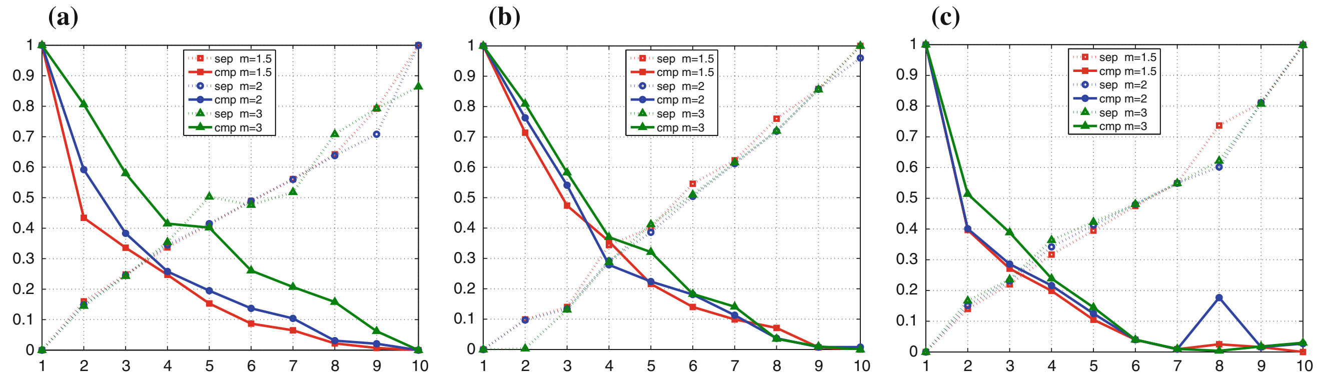

Fig. 3 Compactness and separation for unweighted (**a**); weighted, equal importance (**b**); and weighted, variable importance (**c**) clustering

leaving little difference between regular capabilities $1 \rightarrow 6$ and the unsupported capabilities $7 + 8$. This difference becomes more significant once we reduce the importance of the general capabilities (Fig. 3c). Separation is maximal when each cluster contains a single element. As long as clusters of significant elements split into increasingly smaller clusters, the centroids remain close together, adding little to separation. The distance between centroids grows once less significant elements form individual clusters clearly separated from the existing cluster centers.

Comparing unweighted and both types of weighted clustering, we notice weighted compactness reaching its minimum once all important elements (i.e., all capabilities with comparatively high weights) reside in separate clusters. Unweighted compactness drops similarly fast at the beginning, but then phases out, reaching its minimum at $z = |C|$. This effect is most visible in Fig. 3c where compactness almost reaches zero already for $z = 7$.

5.7 Intra-cluster service ranking

The score for service j within cluster k is determined by the original utility value, capability importance, and capability cluster membership:

$$r_{jk} = \frac{\sum_i u_{ij} * \tau_i * \mu_{ik}}{\sum_i \tau_i * \mu_{ik}} \quad (13)$$

Ultimately, arrive at the service rankings in Table 4 for the scenario cluster result (left side in Table 2). The optimum aggregation consists then of service S_7 (providing capabilities C_4, C_5, C_6) and service S_2 (providing capabilities C_1, C_2, C_3). Note that the scores in both clusters remain comparatively low as services lack fulfillment of capabilities C_7 and C_8. Although no service supports these capabilities, they nevertheless remain equally important as $C1 \rightarrow C6$ (as specified by the user weights τ).

Table 4 Final ranking results for the scenario

Rank	$r_{j,1}$	$r_{j,2}$
1	S_7 (57.24)	S_2 (46.38)
2	S_8 (46.05)	S_1 (44.71)
3	S_9 (40.86)	S_3 (42.06)
4	S_{10} (31.66)	S_8 (5.74)
5	S_3 (5.22)	S_7 (3.08)
6	S_5 (4.04)	S_{10} (2.78)
7	S_2 (2.90)	S_9 (0.90)
8	S_1 (1.33)	S_5 (0.74)
9	S_6 (0.34)	S_6 (0.44)
10	S_4 (0.02)	S_4 (0.01)

5.8 Post-clustering aggregation constraints

The optimal service aggregation consists of selecting the top ranked service within each clusters. However, such an aggregation might be subject to conditions. Aggregation constraints that limit the selection of specific services or service provider come into play once service ranking is completed. Any QoS constraints and certain aggregation constraints were addressed already before clustering (see Sect. 4.2). In the scope of this paper, we support service aggregations that respect dependencies between service instances and between services of particular providers:

Joint service dependency (JSD) i.e., if service S_A is included in the composition, then also service S_B must be included. This constraint is only enforceable when both services provide capabilities from different clusters.

Disjoint service dependency (DSD) i.e., if service S_A is included in the composition, then service S_B must not be included.

Joint provider dependency (JPD) i.e., two or more capabilities must be provided by services of the same provider.

Disjoint provider dependency (DPD) i.e., two or more capabilities must be provided by services of different providers.

Algorithm 2 Greedy aggregation algorithm GAA $(RankingResults\ RR, AggregationConstraints\ AC)$.

```
function AGGREGATE($RR$, $AC$)
    /* Initialize service aggregation and constraint violations */
    $\mathcal{A} \leftarrow$ selectTopRankedServices($RR$)
    $\mathcal{V} \leftarrow$ evaluateConstraints($\mathcal{A}$, $AC$)
    while $\mathcal{V} \neq \emptyset$ do
        sortViolations($\mathcal{V}$)
        $v \leftarrow top(\mathcal{V})$
        /* Extract violating service combination */
        $S_H \leftarrow$ higherScore($v$)
        $S_L \leftarrow$ lowerScore($v$)
        if $type(v) = JPD \wedge type(v) = DPD$ then
            /* Create copies of RR that include only desired providers
               before calling $SelectAlternative$ */
        end if
        $S_{N1} \leftarrow$ SelectAlternative($S_H$, $S_L$, $RR$, $AC$)
        $util1 \leftarrow$ evaluateScore($S_H$, $S_{N1}$)
        $S_{N2} \leftarrow$ SelectAlternative($S_L$, $S_H$, $RR$, $AC$)
        $util2 \leftarrow$ evaluateScore($S_L$, $S_{N2}$)
        if $util1 \geq util2$ then
            replace($\mathcal{A}$, $S_L$, $S_{N1}$)
        else
            replace($\mathcal{A}$, $S_H$, $S_{N2}$)
        end if
        $\mathcal{V} \leftarrow$ evaluateConstraints($\mathcal{A}$, $AC$)
    end while
    return $\mathcal{A}$
end function
function SELECTALTERNATIVE($S_X$, $S_Y$, $RR$, $AC$)
    $cl_y \leftarrow$ getCluster($S_Y$)
    $S_N \leftarrow$ top($RR[cl_y]$)
    /* Select the best combination that is not violating any constraint
       */
    while doesViolate($S_X$, $S_N$, $AC$) do
        $S_N \leftarrow$ next($RR[cl_y]$)
    end while
    return $S_N$
end function
```

This constraint is only enforceable when all the affected capabilities populate distinct clusters.

We propose the following greedy algorithm to determine the optimum aggregation. Algorithm 2 processes only simple constraints involving exactly two capabilities, services, or providers. Any more complex constraints involving three or more entities can be broken down into a set of pairwise constraints.

The algorithm starts out with the top ranked service in each cluster (line 3) and checking this initial aggregation for constraint violations (line 4). As long as the set of violations is not empty, we sort the violations by the highest ranked service causing the violation (line 8). From the top violation, we extract the higher (S_H) and lower (S_H) ranked service. The basic principle for resolving the violation is keeping one of the involved services fixed at a time and searching for an alternative second service (lines 14–17). The function `SelectAlternative` (lines 27–35) iterates descending through the ranking results to find the highest combination (as defined by the sum of service scores) that does not violate any constraint. Whichever combination yields better scores is included in the aggregation (lines 18–22). This basic procedure is immediately applicable for JSD and DSD constraints. For JPD and DPD, we need to filter the service ranks according to the underlying provider dependencies before searching for alternatives (lines 11–13). Once a violation is resolved, the aggregation is checked again (line 23), and the while loop (lines 5–24) is repeated for any other remaining violation.

Note that this algorithm is unaware of conflicting aggregation constraints that potentially prevent the algorithm to find a valid composition. Detecting and resolving such conflicts is, however, outside the scope of this article. We refer instead to existing literature (e.g., [10,19]).

6 Evaluation

We focus on analyzing the following aspects of the weighted FCM clustering algorithm:

- The algorithm produces suitable clusters, i.e., well-supported capabilities dominate the cluster formation, while less supported capabilities exhibit fuzzy membership across (all) existing clusters.
- The generated clusters are indeed distinct and promote specialized services to the top of each cluster. Without clustering, these services rank rather low and thus would not be considered for aggregation.
- Cluster results remain overall stable when changing from equal to variable capability requirements weights. Although the promoted capabilities tend to form new/changed clusters, the remaining capability groups remain stable.
- Service aggregations exhibit higher averaged service scores than a single best service providing all capabilities.

To the best of our knowledge, there exists no data set describing the utility, respectively the score, of software-based web service operations. Most collections consist of services with a single capability or provide only Quality of Service data; both types are unsuitable for evaluating our clustering approach.

In the domain of Human-provided Services [27] (HpS), however, experts join service compositions to provide their skills. For HpS, capabilities describe the level of expertise or level of participation. Also the notion of QoS applies in the form of response time or throughput. We evade the problem of having no data on software services by focusing on data sets describing HpS. Hence, in the scope of this evaluation, we describe the process of capability matching, filtering, clustering, and ranking for ultimately determining the best aggregation of HpS for a particular set of skills (i.e., capabilities) based on a real-world data set from the Slashdot community.

Slashdot[1] is a user-driven news portal focusing on various aspects of information technology. News fall into multiple categories (i.e., subdomains). Slashdot exhibits similar characteristics as large-scale complex service systems. Some entities remain consistently active throughout all subdomains. Other entities join in an ad hoc manner, participate for a limited period, and then vanish again. In Slashdot, users are interested in providing their knowledge to improve the quality and information content of a story (i.e., they fulfill a task). They rarely engage in direct communication with other users [8,28]. Hence, we claim that Slashdot users accurately mimic the behavior of Human-provided Services.

Slashdot postings are subject to a moderation system. Postings receive scores between -1 (low quality) and $+5$ (high quality). Predicates enable the classification of postings according to $insightful$, $interesting$, $informative$, $funny$, etc. content. We define two posting metric to measure the skill level: total scores (i.e., *Score*) and total posting counts (i.e., *Count*). Specifically, we derive an HpS' total posting count and the total score for every subdomain and predicate combination. We prefer to keep these two capabilities separated, as relying on a single average score favors HpS with very few postings which were lucky to receive high ratings. In contrast, HpS contributing regularly are unlikely to receive continuously high scores. We consider total scores to be equally important to total postings throughout our experiments. Thus, requirement weights for pairs of these statistics (respective to predicates) are always identical. For each capability requirement, the HpS with the highest total score (respectively total posting count) receives a utility value of 100, with the worst HpS having utility 0. The underlying data set contains postings from February 1 to July 1, 2008.

6.1 Experiment setup

Given the set of predicates P, subdomains SD, and two posting metrics, we arrive at the global set of available capability requirements set C of size $|P| * |SD| * 2$. In our experiments, we focus on an HpS aggregation that provides skills for three subdomains—Ask, $Entertainment$, and $Mobile$—and focus in particular on the scores of $funny$, $interesting$, and $insightful$ postings. Thus, there are 18 requirements (i.e., $Ask-Fun-Count$, $Ask-Fun-Score$, $Ask-Ins-Count$, etc.) as input to our clustering algorithm.

First, we apply a QoS metric to identify only regular HpS instances. In particular, we filter out all HpS that did not achieve at least 5 posting in the desired subdomains of score 2 or higher. We treat HpS below this QoS constraint as services exhibiting different skills/capabilities we are not interested in. For each HpS, we then derive the utility values for all requirements. As a side effect, reducing the initial set of candidates (here 257 users) reduces the duration of the clustering process.[2]

6.2 Unweighted clustering results

First, we analyze unweighted clustering, where we ignore requirement importance and work with the unweighted utility matrix $\mathcal{U}$. The cluster quality metric identifies 12 clusters to optimally describe the implicit capability groups. Specifically, the resulting cluster membership places *Count* and *Score* requirements of every subdomain and predicate in the same cluster except for Ask-Insightful, Entertainment-Funny and Entertainment-Interesting which populate individual clusters. Cluster membership μ is larger than 0.9 for all constraints.

We calculate the pairwise Jaccard similarity between any two clusters for the top 50 HpS. Figure 4a visualizes the resulting similarity matrix. Row 13 and column 13 contain the unclustered ranking set. We notice that some clusters yield extremely high similarity. Specifically clusters 3 and 5, 4 and 10, as well as cluster 6 and 11 have many HpS in common. Although all clusters yield significant differences to the unclustered ranking, the clustering process has created three pairs of clusters that should be merged. Incidentally, these pairs comprise of the above-mentioned constraints, where Score and Count of the same subdomain and predicate end up in different clusters (Ask-Insightful, Entertainment-Funny and Entertainment-Interesting). Merging these cluster pairs would not significantly reduce the overall clustering quality.

6.3 Weighted, equal importance clustering results

Second, we cluster again with equal requirement importance but weighted utility matrix $\mathcal{U}_\omega$. Capabilities that exhibit low fulfillment support have less impact during clustering than well-supported requirements. Table 5 provides the capability membership in the resulting six clusters. Most capabilities yield crisp cluster membership ($\mu > 0.9$) with exception to Entertainment-Funny, Entertainment-Interesting, and Mobile-Funny which do not strongly belong to any cluster. The weight vector $\mathcal{W}$ is a good indicator on which capability requirements are likely to yield crisp clusters. A low weight value by itself, however, is not sufficient. The constraint Ask-Funny-Score ($\omega = 0.804$) exhibits lower weight than Entertainment-Funny-Count ($\omega = 0.836$) but ends up clearly assigned to cluster 1. Here, the close correlation between count and score values is decisive.

[1] http://www.slashdot.org.

[2] The data set containing the 257 users is available as a csv file from: http://www.infosys.tuwien.ac.at/staff/dorn/soca-dataset257.csv.

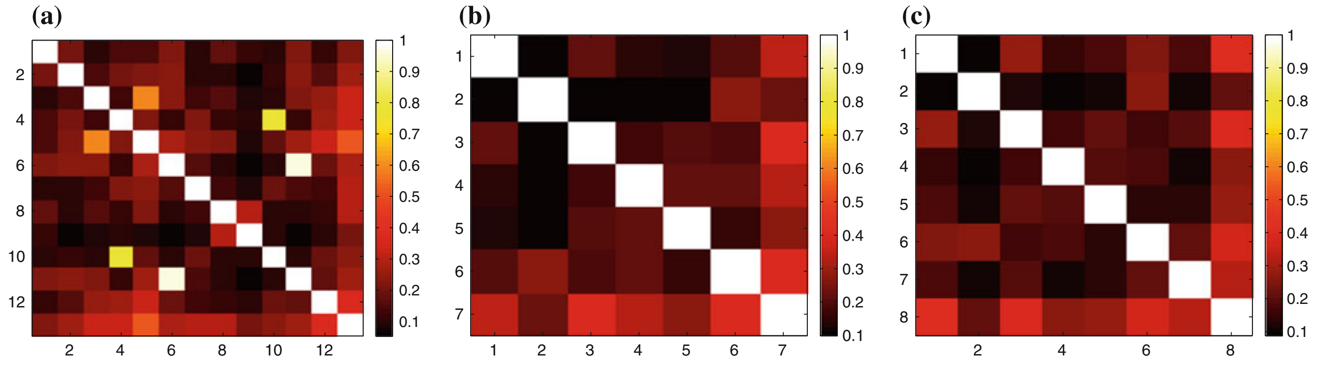

Fig. 4 Cluster Jaccard similarity for Top 50 HpS for unweighted (**a**); weighted, equal importance (**b**); and weighted, variable importance (**c**) capability constraints

Table 5 Cluster membership and weight vector $\mathcal{W}$ for 18 capability requirements from Count, Score, with subdomains Ask, Entertainment, and Mobile and predicates Funny, Insightful, and Interesting

Constr.	ω	K_1	K_2	K_3	K_4	K_5	K_6	τ	ω	K_1	K_2	K_3	K_4	K_5	K_6	K_7
A-Fun-C	0.864	**0.959**	0.005	0.011	0.012	0.008	0.006	0.05	0.822	0.253	0.070	0.169	0.186	0.117	0.092	0.114
A-Fun-S	0.805	**0.968**	0.004	0.009	0.009	0.006	0.005	0.05	0.766	0.260	0.067	0.166	0.183	0.115	0.091	0.118
A-Ins-C	1.394	0.019	0.010	0.023	0.025	**0.914**	0.010	0.05	1.325	0.019	0.011	0.024	0.027	**0.901**	0.01	0.009
A-Ins-S	1.735	0.005	0.003	0.008	0.007	**0.975**	0.003	0.05	1.649	0.005	0.003	0.007	0.007	**0.973**	0.003	0.002
A-Int-C	1.185	0.009	0.004	0.006	**0.969**	0.008	0.005	0.05	0.901	0.008	0.004	0.007	**0.961**	0.009	0.006	0.004
A-Int-S	1.226	0.011	0.004	0.007	**0.964**	0.009	0.005	0.05	0.932	0.008	0.004	0.007	**0.961**	0.010	0.006	0.004
E-Fun-C	0.836	0.240	0.120	0.176	0.158	0.104	0.202	0.05	0.795	0.167	0.099	0.143	0.134	0.087	0.155	0.215
E-Fun-S	0.702	0.253	0.116	0.170	0.146	0.099	0.215	0.05	0.667	0.165	0.092	0.133	0.120	0.080	0.156	0.253
E-Ins-C	1.035	0.010	0.004	**0.965**	0.007	0.010	0.004	0.05	0.984	0.008	0.004	**0.967**	0.006	0.008	0.004	0.004
E-Ins-S	1.008	0.007	0.003	**0.976**	0.004	0.007	0.003	0.05	0.958	0.008	0.003	**0.970**	0.005	0.007	0.003	0.004
E-Int-C	0.513	0.338	0.090	0.179	0.156	0.105	0.132	0.08	0.732	**0.799**	0.023	0.045	0.041	0.027	0.032	0.032
E-Int-S	0.748	0.291	0.085	0.215	0.158	0.143	0.108	0.08	1.066	**0.945**	0.006	0.014	0.011	0.010	0.007	0.007
M-Fun-C	0.563	0.197	0.136	0.123	0.112	0.087	0.346	0.07	0.91	0.002	0.002	0.002	0.002	0.001	0.004	**0.987**
M-Fun-S	0.563	0.205	0.130	0.127	0.110	0.087	0.341	0.07	0.911	0.002	0.002	0.001	0.001	0.001	0.003	**0.990**
M-Ins-C	1.459	0.003	**0.980**	0.003	0.003	0.003	0.007	0.05	1.387	0.003	**0.976**	0.003	0.003	0.003	0.008	0.004
M-Ins-S	1.503	0.003	**0.983**	0.003	0.003	0.003	0.007	0.05	1.429	0.003	**0.980**	0.003	0.003	0.003	0.007	0.003
M-Int-C	0.814	0.006	0.011	0.005	0.006	0.004	**0.969**	0.05	0.774	0.008	0.014	0.006	0.007	0.005	**0.943**	0.018
M-Int-S	1.046	0.008	0.016	0.006	0.008	0.005	**0.957**	0.05	0.994	0.003	0.007	0.003	0.003	0.002	**0.976**	0.006

Crisp cluster membership in bold font

We test whether clusters provide more specialized HpS than the unclustered ranking result by pairwise comparing the top-k HpS with Pearson's correlation coefficient ($\rho = [-1, 1]$) and Jaccard similarity ($J = [0, 1]$). Table 6 lists the ranking differences of the top 10, 50, and 100 HpS. The Jaccard similarity measures the set overlap of HpS regardless of their rank. Complete overlap results in $J = 1$, while no overlap results in $J = 0$. Pearson's coefficient requires both sets to contain the same elements. We therefore extend beyond the top [10;50;100] HpS and take the union of elements from both rankings (given in brackets in Table 6) and then compute ρ. We have $\rho = -1$ for perfect negative correlation, $\rho = 0$ for no correlation, and $\rho = 1$ for perfect positive correlation.

Average Jaccard similarity remains low for the top 10, 50, and 100 HpS. On average, only 34% of the top-50 non-clustered HpS are also listed in individual clusters. The average Pearson's coefficient stresses the ranking differences even more. We observe no correlation in ranks for up to the top 50 HpS and only a small correlation for the top 100 HpS.

6.4 Weighted, variable importance clustering results

Finally, we apply variable requirements weights to analyze their effect on the clustering result. In this third experiment, we increase the importance of following four constraints: Entertainment-Interesting-[Count|Score] ($\tau = 0.08$) and Mobile-Funny-[Count|Score] ($\tau = 0.07$). The remaining

Table 6 Ranking differences of top [10;50;100] HpS between each cluster and the unclustered ranking order measured with Pearson's correlation coefficient (ρ) and Jaccard similarity (J)

Clus.	Top 10		Top 50		Top 100		Top 10		Top 50		Top 100	
	ρ	J	ρ	J	ρ	J	ρ	J	ρ	J	ρ	J
K_1	−0.38 (16)	0.25	0.02 (74)	0.35	0.11 (144)	0.39	−0.10 (15)	0.33	0.22 (71)	0.41	0.26 (138)	0.45
K_2	−0.60 (17)	0.18	−0.15 (81)	0.24	−0.01 (147)	0.36	−0.64 (17)	0.18	−0.20 (83)	0.21	−0.06 (147)	0.36
K_3	0.31 (15)	0.33	0.19 (71)	0.41	0.22 (139)	0.44	0.21 (15)	0.33	0.10 (72)	0.39	0.16 (142)	0.41
K_4	−0.32 (17)	0.18	0.01 (75)	0.33	0.01 (143)	0.40	−0.37 (17)	0.18	−0.04 (79)	0.27	−0.05 (150)	0.33
K_5	0.07 (16)	0.25	0.03 (78)	0.28	0.12 (138)	0.45	0.00 (16)	0.25	−0.06 (78)	0.28	0.01 (146)	0.37
K_6	−0.37 (17)	0.18	−0.06 (71)	0.41	0.19 (138)	0.45	−0.62 (18)	0.11	−0.16 (73)	0.37	0.10 (142)	0.41
K_7							−0.39 (18)	0.11	−0.15 (76)	0.32	0.12 (136)	0.47
Avg	−0.21	0.23	0.01	0.34	0.11	0.41	−0.27	0.21	−0.04	0.32	0.08	0.40

Columns 2–7: weighted, equal importance (ω) clustering; Columns 8–13: weighted, variable importance ($\omega + \tau$) clustering

requirements exhibit identical weights ($\tau = 0.05$) so the sum of weights remains 1.

Clustering this configuration produces one more cluster (Table 5 right side). Also cluster membership has changed for some requirements. Both, Entertainment-Interesting and Mobile-Funny populate now their own cluster exhibiting high crispness. On the other hand, Ask-Funny looses its clear membership in a single cluster, now yielding fuzzy membership across all clusters. Entertainment-Funny maintains its fuzzyness but shares its largest membership with Mobile-Funny instead of Entertainment-Interesting.

Again, we pairwise compare the non-clustered ranking and each cluster for ranking differences. Compared to the previous experiment, Jaccard similarity is similarly low (average $\sim=$ [0.21; 0.40]) and Pearson's coefficient supports the distinction in clusters as there is no correlation of the top 10, 50, and 100 between non-clustered HpS and clustered HpS. Cluster 7 emerges not only due to the changed importance. We evaluate the pairwise cluster similarity to ensure that the underlying data justifies this additional cluster. Figure 4 provides the similarity matrix including the non-clustered set in the last row and column. Cluster 7 remains distinctively different from the other clusters for the top 50 HpS. Hence, the additional cluster allows for further specialization compared to the previous experiment.

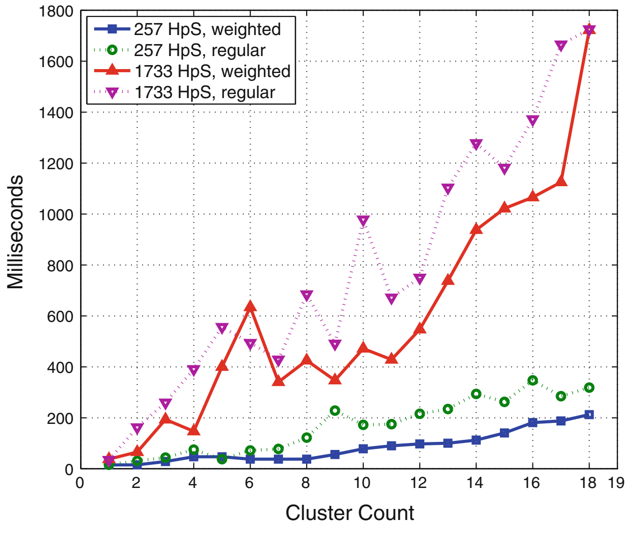

Fig. 5 Cluster algorithm performance in milliseconds for 18 capabilities (i.e., $c = 1$ to 18): *dashed lines* for the regular FCM , *full lines* for the weighted FCM, with 257 HpS (*circles* and *squares*) and 1,733 HpS (*triangles*)

6.5 Performance evaluation

We measure the run-time performance of the clustering algorithm for the described experiment of 257 HpS and a larger set of 1,733 HpS (using subdomains: *Mobile*, *Tech*, and *Science*). The performance tests were carried out on a Windows XP Intel core2duo at 2.5 GHz laptop with 3.5 GB RAM. We implemented the weighted (as well as reference regular) FCM algorithm in Java 1.6. As the focus of this research was mostly on producing sensible clusters rather than on maximal performance, we did not rely on dedicated matrix libraries nor other speed-up mechanisms such as multicore support. Thus, we expect that the following performance results can easily be reduced by a factor of 10.

Figure 5 displays the average runtime in milliseconds for completing one instance of the weighted and regular FCM algorithm for $c = 1 \rightarrow 18$ clusters (for 18 capabilities). Our weighted FCM algorithm, for example, takes on average 340 ms to determine the best 7 clusters for 1,733 HpS and 425 ms for the best 8 clusters. When checking the complete spectrum for the best capability clusters (here 1 to 18), the weighted FCM takes 1.5 s to complete with 257 HpS, respectively, the regular FCM algorithm 3 s. For the larger data set of 1,733 HpS, the runtime increases to 10.6 and 14.2 s, respectively. Our weighted FCM is consistently faster

Table 7 Utility comparison of top 10 HpS of each cluster and the unclustered ranking order

Cluster weight w_C provides the importance of all capabilities in that cluster according to capability membership

	Non	K_1	K_2	K_3	K_4	K_5	K_6	K_7	Impr (%)
	w_C	0.196	0.150	0.167	0.159	0.143	0.185		
1	60.7	76.2	73.3	76.3	78.2	84.1	61.9		23
2	39.3	48.8	71.7	71.9	70.1	79.0	49.2		62
3	35.7	47.5	67.4	60.3	47.1	61.0	42.2		50
10	30.5	32.8	50.3	39.3	38.8	50.6	31.1		30
	w_C	0.177	0.119	0.135	0.115	0.119	0.126	0.209	
1	58.2	84.4	82.6	84.5	85.0	90.9	77.5	85.8	45
2	43.6	83.7	78.1	76.7	71.2	81.5	47.9	49.8	58
3	34.3	55.7	75.7	64.4	48.0	65.7	44.7	48.7	66
10	29.5	39.0	55.7	41.5	37.8	53.0	36.6	33.3	40

than the regular FCM. The assignment of low impact entities to specific clusters has little impact on the convergence condition and thus enables our weighted FCM to settle earlier. The overhead of weighting the membership matrix is thus more than compensated through earlier termination. The runtime of weighted and regular FCM is linear with the number of clusters to check where the vector dimension (i.e, number of HpS) determines the steepness. Besides the integration of high-performance matrix libraries, we propose to limit the number of clusters to check to sensible values (and not the complete spectrum). In our example, restricting the search to half the capabilities (i.e., $c_{\max} = 9$) reduces the overall runtime of our weighted FCM to 0.3 s for 257 HpS, and 2.6 seconds for 1,733 HpS, respectively.

6.6 Discussion of clustering results

The three experiments have shown how requirement weights influence the clustering result for the same underlying utility data. Weighted, equal importance clustering highlights the requirements that are fulfilled by most HpS. Subsequently weighted, variable importance clustering successfully shifts the focus onto the capability requirements considered more important. The general cluster structure, however, remains stable. Observe that only one one 'old' cluster experienced significant membership changes, and one new cluster was created. New, or changed, clusters emerge only when the preferred requirements indeed meet a distinct difference in the underlying capability data set when compared to the remaining requirements.

Regular clustering—as tested in the unweighted experiment—produced too many clusters. In contrast, both weighted experiments exhibited very distinct clusters. The inter-cluster Jaccard similarity remains in the range of $[0 \leftrightarrow 0.25]$, $[0.02 \leftrightarrow 0.28]$, and $[0.13 \leftrightarrow 0.42]$ for the top-10, top-50, and top-100 HpS (out of 255), respectively. The corresponding similarity between the unranked result and each cluster resulted in slightly higher values, namely $[0.11 \leftrightarrow 0.33]$, $[0.21 \leftrightarrow 0.41]$, and $[0.33 \leftrightarrow 0.47]$ for the top-10, top-50, and top-100 HpS.

Our new weighted FCM clustering algorithm correctly identifies collocated capabilities. For all of our experiments with the WFMC algorithm, the posting metrics of *Score* and *Count* always ended up as pairs in the same cluster.

Clustering also promotes HpS to the top elements in a cluster which are badly ranked in the non-clustered set. The Pearson's correlation coefficient emphasized the element positioning difference between non-clustered and clustered ranking order for both weighted experiments. We observed no correlation for the top-10 not top-50 HpS and only a slight positive correlation for the top-100 HpS.

Table 7 compares the utility values of the single best unclustered service to an aggregation of services from the various clusters. The average utility benefit for selecting the top-10 HpS set from every cluster amounts to a 38% (equal importance) to 51% (variable importance) utility increase compared to the unclustered ranking result.

Compared to existing techniques that map each required capability to a dedicated service, our approach greatly reduces the number of required services. In the weighted examples only 6, respectively 7, services are required compared with the 18 capability requirements.

7 Related work

Numerous papers improve the FCM algorithm to achieve robustness (e.g., [4,13,33]). These techniques apply data distribution intrinsic metrics to identify and mitigate the effect of outliers and noise. We focus on achieving optimum clusters where significant capability constraints and capability support by actual service instances should influence the result more than insignificant constraints or services.

In the SOA domain, clustering techniques have been applied to compute the similarity of web services based on key words extracted from the service interface descriptions

(WSDL). Dong et al. [6] propose a Web service search engine that applies keywords as well as input and output matching to determine matching services. They apply term clustering to determine sets of parameter names to give web service operations a semantic meaning. Nayak and Lee [18] transform WSDL descriptions into the OWL-S format en enhance the result with information found in human readable service descriptions. Hierarchical agglomerative clustering produces then groups of semantically similar services. Platzer et al. [21] enhance a distributed vector space search engine to achieve rapid, scalable retrieval of services. Services that match the users query are then clustered according to term similarity. In the domain of recommendation support for mashup development, Blake and Nowlan [3] investigate service similarity measures based on syntactical message analysis to recommend suitable services. Their algorithm is, however, not laid out for distinguishing between services that apply similar data structures but provide completely different capabilities. Ranabahu et al. [22] take this idea a step further and propose a faceted classification-based approach. These approaches aim to find similar services given a particular query. In contrast, we extract a set of complementary service clusters, from which an aggregation of individual services provides the required capabilities. Also, these approaches observe only keywords, but not consider a service's fitness to provide a certain capability.

More advanced techniques exist in the broader domain of service composition. Work focusing on QoS-centric service selection, recommendation, and composition is orthogonal to our approach. QoS metrics describe parameters such as throughput and latency [24], or trust and reputation [15,31] that apply equally to any service. In contrast, our aggregation mechanism builds on capabilities which reside between QoS and service interface descriptions. As we have demonstrated, they extend our core approach to improve the initial selection of service candidates, and the final process of selecting the actual services that create a composition. The notion of capabilities is not new. It shares some similarities with the *Composite Capability/Preference Profiles* (CC/PP) specification [11]. The purpose of CC/PP foresees web clients to transmit their capabilities (e.g., screen size) in order to enable service providers (e.g., web servers) to adapt the delivered content accordingly. In contrast, services describe their capabilities to enable service clients to select the most suitable service. An example alternative to describing service capabilities is the semantic modeling approach through WSMO [23]. More recent efforts in the semantic web services domain provide more light-weight description languages such as WSMO-lite [30] (targeted to SOAP-based service) and hREST [12] (addressing REST-based services and Web APIs). Pedrinaci and Domingue describe how these descriptions relate to each other and can be unified in the Minimal Service Model [20]. However, independent of the description language used, a service provider would have to create dedicated capability profiles. While our approach is intentionally more pragmatic and subsequently more straight-forward to realize in a real-world environment, we are aware that there is added benefit for representing our capability model in RDF for integration with a semantic service description language. The *wl:NonFunctionalParameter* from the WSMO-lite model can potentially serve as an entry point. Automatic web service mediation techniques based on mediation spaces [5] could then assist in matching differences in the capability descriptions. Such efforts to overcome heterogeneous service provides and consumers are, however, complementary to the mechanism presented in this paper and thus remain future work.

Other complementary techniques to service composition are goal-driven approaches such as [9,26,32]. These techniques, however, focus on optimal replacement of individual services to maintain desirable, global composition quality metrics. Integration of those techniques after the capability clustering procedure is expected to be very effective for maintaining a service composition.

8 Conclusion

We have presented a weighted fuzzy clustering approach to exploit implicit service groups. Our technique produces service aggregations that significantly reduce the number of actual involved services (compared to the number of capability requirements) while taking advantage of service specialization. As a side effect, discovered service clusters enable the rapid replacement of individual services without having to analyze the overall composition. Analysis of capability support among the services, and capability requirement weights allow for fine-grained cluster formation control. Integration of required QoS levels allows for further refinement of candidate services. Post-clustering aggregation constraints control the ultimate service aggregation to observe service and provider dependencies. We successfully demonstrated the effectiveness of our approach based on a real-world data set.

Currently all service capabilities are assumed to be at the same level of granularity. Fine-grained analysis of capabilities, their support by services, and the subsequent impact requires the consideration of capability hierarchies. The capability model already provides a hierarchical structure that enables reasoning on the relationship between capabilities. We expect this analysis to improve the detection of general and specific capabilities and thus adjust the corresponding weights more accurately. In the future, we also plan to include feedback from the actual composition orchestration and deployment process to learn which services turn out to be difficult or even impossible to integrate.

References

1. Bezdek JC (1981) Pattern recognition with fuzzy objective function algorithms. Kluwer, Norwell
2. Bezdek JC, Ehrlich R, Full W (1984) Fcm: the fuzzy c-means clustering algorithm. Comput Geosci 10(2–3):191–203. doi:10.1016/0098-3004(84)90020-7. http://www.sciencedirect.com/science/article/pii/0098300484900207
3. Blake M, Nowlan M (2008) Predicting service mashup candidates using enhanced syntactical message management. In: Proceedings of IEEE international conference on services computing, 2008. SCC '08, vol 1, pp 229–236. doi:10.1109/SCC.2008.68
4. Chintalapudi K, Kam M (1998) A noise-resistant fuzzy c means algorithm for clustering. In: The 1998 IEEE international conference on fuzzy systems proceedings, 1998. IEEE world congress on computational intelligence, vol 2, pp 1458–1463. doi:10.1109/FUZZY.1998.686334
5. Dietze S, Gugliotta A, Domingue J, Yu H, Mrissa M (2010) An automated approach to semantic web services mediation. Serv Orient Comput Appl 4:261–275. doi:10.1007/s11761-010-0070-7
6. Dong X, Halevy A, Madhavan J, Nemes E, Zhang J (2004) Similarity search for web services. In: VLDB '04: proceedings of the thirtieth international conference on very large data bases. VLDB Endowment. pp 372–383
7. Dorn C, Schall D, Dustdar S (2009) Context-aware adaptive service mashups. Services Computing Conference, APSCC, IEEE Asia-Pacific. pp 301–306. doi:10.1109/APSCC.2009.5394107
8. Gómez V, Kaltenbrunner A, López V (2008) Statistical analysis of the social network and discussion threads in slashdot. In: WWW '08: proceeding of the 17th international conference on world wide web. ACM, New York, NY, USA. pp 645–654. doi:10.1145/1367497.1367585
9. Greenwood D, Rimassa G (2007) Autonomic goal-oriented business process management. In: ICAS '07: proceedings of the third international conference on autonomic and autonomous systems. IEEE Computer Society, Washington, DC, USA. p 43. doi:10.1109/CONIELECOMP.2007.61
10. Hausmann J, Heckel R, Taentzer G (2002) Detection of conflicting functional requirements in a use case-driven approach. In: Proceedings of the 24rd international conference on software engineering, 2002. ICSE 2002
11. Kiss C (2007) Composite capability/preference profiles (cc/pp): Structure and vocabularies 2.0. Tech. rep., W3C
12. Kopecký J, Gomadam K, Vitvar T (2008) Hrests: an html microformat for describing restful web services. In: Proceedings of the 2008 IEEE/WIC/ACM international conference on web intelligence and intelligent agent technology—vol 01. IEEE Computer Society, Washington, DC, USA. pp 619–625. doi:10.1109/WIIAT.2008.379. http://portal.acm.org/citation.cfm?id=1486927.1486962
13. Leski J (2003) Towards a robust fuzzy clustering. Fuzzy Sets Syst 137(2):215–233. doi:10.1016/S0165-0114(02)00372-X
14. Macqueen JB (1967) Some methods of classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, pp 281–297
15. Maximilien E, Singh M (2005) Self-adjusting trust and selection for web services. pp 385-386. doi:10.1109/ICAC.2005.53
16. McBratney A, De Gruijter J (1992) A continuum approach to soil classification by modified fuzzy k-means with extragrades. J Soil Sci 43:159–175
17. Mrissa M, Ghedira C, Benslimane D, Maamar Z, Rosenberg F, Dustdar S (2007) A context-based mediation approach to compose semantic web services. ACM Trans Internet Technol 8(1):4. doi:10.1145/1294148.1294152
18. Nayak R, Lee B (2007) Web service discovery with additional semantics and clustering. In: WI '07: Proceedings of the IEEE/WIC/ACM international conference on web intelligence. IEEE Computer Society, Washington, DC, USA. pp 555–558. doi:10.1109/WI.2007.112
19. Nepal S, Zic J (2008) A conflict neighbouring negotiation algorithm for resource services in dynamic collaborations. IEEE international conference on services computing, vol 2, pp 283–290. http://doi.ieeecomputersociety.org/10.1109/SCC.2008.18
20. Pedrinaci C, Domingue J (2010) Toward the next wave of services: linked services for the web of data 16(13):1694–1719
21. Platzer C, Rosenberg F, Dustdar S (2009) Web service clustering using multidimensional angles as proximity measures. ACM Trans Internet Technol 9(3):1–26. doi:10.1145/1552291.1552294
22. Ranabahu A, Nagarajan M, Sheth AP, Verma K (2008) A faceted classification based approach to search and rank web apis. In: IEEE International Conference on Web Services (ICWS). IEEE Press. pp 177–184
23. Roman D, Keller U, Lausen H, de Bruijn J, Lara R, Stollberg M, Polleres A, Feier C, Bussler C, Fensel D (2005) Web service modeling ontology. Appl Ontol 1(1):77–106. http://iospress.metapress.com/content/dccb867p347xxebx
24. Rosenberg F, Leitner P, Michlmayr A, Celikovic P, Dustdar S (2009) Towards composition as a service—a quality of service driven approach. pp 1733–1740. doi:10.1109/ICDE.2009.153
25. Rosenberg F, Platzer C, Dustdar S (2006) Bootstrapping performance and dependability attributes of web services. In: Proceedings of the IEEE international conference on web services (ICWS'06). IEEE Computer Society. pp 205–212
26. Ryu SH, Casati F, Skogsrud H, Benatallah B, Saint-Paul R (2008) Supporting the dynamic evolution of web service protocols in service-oriented architectures. ACM Trans Web 2(2):1–46. doi:10.1145/1346237.1346241
27. Schall D, Truong HL, Dustdar S (2008) Unifying human and software services in web-scale collaborations. IEEE Internet Comput 12(3):62–68
28. Skopik F, Truong HL, Dustdar S (2009) Trust and reputation mining in professional virtual communities. In: 9th international conference on web engineering (ICWE). Springer
29. Tan AH, Tan CL, Sung SY (2003) On quantitative evaluation of clustering systems. In: Wu W, Xiong H, Shekhar S (eds) Clustering and information retrieval. Kluwer, Dordrecht, pp 105–133
30. Vitvar T, Kopecký J, Viskova J, Fensel D (2008) Wsmo-lite annotations for web services. In: Proceedings of the 5th European semantic web conference on the semantic web: research and applications, ESWC'08. Springer, Berlin, Heidelberg. pp 674–689. http://portal.acm.org/citation.cfm?id=1789394.1789455
31. Vu LH, Hauswirth M, Aberer K (2005) Qos-based service selection and ranking with trust and reputation management. In: OTM conferences (1), pp 466–483
32. Yu T, Lin KJ (2005) Adaptive algorithms for finding replacement services in autonomic distributed business processes. In: Proceedings of the autonomous decentralized systems, 2005. ISADS 2005. pp 427–434. doi:10.1109/ISADS.2005.1452105
33. Zhang JS, Leung YW (2004) Improved possibilistic c-means clustering algorithms. IEEE Trans Fuzzy Syst 12(2):209–217. doi:10.1109/TFUZZ.2004.825079