Context-aware Adaptive Service Mashups

Christoph Dorn, Daniel Schall, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology, 1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

***Abstract*—Mashup tools are becoming increasingly important enabling users to compose services and processes on the Web. Most existing tools focus on Web-based interfaces, usability, and visual languages for creating mashups. A major challenge that has received limited attention is context-awareness and adaptivity of service mashups. In this paper we focus on two main aspects: First, a service capability model describing service characteristics that can be tracked and matched against the requirements associated with service mashups and second an algorithm to recommend refinements such as replacing services within mashups. We implemented a set of adaptation algorithms to validate our approach in real service-oriented systems.**

I. Introduction

Mashup tools have become increasingly popular supporting the end-user in composing services and aggregating data from various sources. Major industry players including Yahoo![1], or Microsoft[2] have developed a set of Web-centric tools for designing and executing mashups. The focus of these tools is to reduce the amount of programming effort and time needed to create compositions. We believe that context-awareness and adaptivity play an important role in the widespread proliferation of service mashups. Such compositions are often executed in highly dynamic environments such as the Web. Service availability changes dynamically and may render a mashup unusable. We propose semiautomatic reconfiguration and replaceability strategies to support the service mashup developer in the redesign of compositions. Our approach is based on a service capability model enabling a matching of suitable services satisfying requirements such as minimum reputation of a service or maximum cost. In the next step we detail the problem of replaceability in a concrete use case example.

Motivating Scenario

Consider a mashup comprising of services from multiple organizations. User Dave (the Mashup Developer) creates a service mashup to support collaboration and interactions in a software development team. Along with the services that are part of the composition, Dave selects the number of users to be involved in the development team. Based on the team members' skills, location, and roles he selects the suitable services such as: code versioning storage, central document repository, email list management, bug tracking, etc. As the initial development effort is expected to be low, Dave selects services supporting only a limited set of users, which is cheaper.

After the first release of the software product, the great success encourages the managers to further develop the software with support from external programmers. Team participants may thus be scattered around the globe. Changing conditions by transforming the team has consequently impact on the requirements such as the number of users that need to be supported. Also, additional aspects, for example, whether existing services can be used in an organizational structure make it very difficult for Dave to track the changing requirements and redesign the service mashup.

Dave needs to receive automatic recommendations which services to apply in the changed environment. This recommendations need to be based on service capabilities satisfying the demanded set of requirements in a given context.

Contributions and Outline

The main contributions of this paper are described in following sections. In Section II we present our methodology for context-aware adaptation. The service capability model is presented in Section III followed by Section IV on requirement rules and the gracefully degrading requirements filtering algorithm. The clustering algorithm in Section V applies service utility values to identify optimal constraint groups. Experiments are shown in Section VI, a discussion on related work in Section VII, and finally we conclude the paper in Section VIII.

II. Adaptation Methodology

Mashup adaptation relies on two complementary building blocks: capabilities and requirements. Capabilities describe non-functional service properties to determine a service's applicability in a specific mashup context. Requirements specify the necessary capabilities for a particular mashup.

The upper part of Figure 1 visualizes the dependencies between the various data models. The *Service Capability Meta Model* (a) introduces the fundamental capability concepts. The *Capability Descriptions* (b) detail common capability templates. These templates define a portfolio of common capability structures and properties independent of actual services. The *Service Metadata* (c) map particular capability templates to service operations.

[1] `http://pipes.yahoo.com/pipes/`
[2] `http://www.popfly.com/`

Mashup Context and Metrics (d) describe properties of the runtime environment. Metrics measure interaction and dependency aspects in-between users as well as services. *Requirement Rules* (e) are similar to context-dependent goals. They describe which capabilities—respectively constraints on capabilities—apply for a particular set of context conditions. Ultimately, the *Mashup Configuration* (f) lists the set of involved services and outlines which service capabilities fulfill the given set of requirement rules.

The adaptation process exhibits two main phases (Figure 1 lower part). *Mashup Monitoring* observes context changes and determines which mashup configurations are affected. A configuration needs reevaluation when associated requirement rules involve the observed context change. Requirements rules then generate updated capability constraints. We trigger adaptation when these constraints no longer sufficiently match the current service configuration.

Adaptive Management takes the set of requirements and determines the set of services with the best fitting capabilities. First, we compare each service that matches at least one requirement against all requirements. Each requirement specifies the utility function and evaluation parameters to calculate a service's score. Traditional selection approaches usually assume that one service has to fulfill all requirements. However, with increasingly specialized constraints, it is unlikely that a single service exhibits all required capabilities to sufficient degree. Splitting the services into multiple groups, and ranking them separately, will provide more useful service recommendations. The clustering process applies the obtained utility scores to determine the optimal assignment of requirements to these groups. *Candidate comparison* ranks the service within each of these groups and returns the sets of top rated services. The mashup developer selects amongst these services and reconfigures the mashup accordingly.

A. Mashup Context

We discuss a set of aspects influencing the adaptation of service mashups. A service mashup potentially needs to be reconfigured based on the changing properties (i.e., the context) of the environment. Let us consider the following examples:

Reputation of a service is the opinion expressed by consumers of the service. Opinion can be collected based on feedback ratings of users (e.g., see [1] for trust and reputation mechanisms in service-oriented systems) or by monitoring interactions (invocations technically speaking) between services, i.e., whether a service was part of a composition in a particular context.

Organization: Mashup users and services from multiple organizations are confronted with trust, hidden information, and control challenges. Measuring and tracking organizational properties is key to adapt, for example, access to resources and monitoring of work progress.

Location: The physical location — a well known metric for context-aware adaptation — can be used to select a service based on the physical proximity to the service consumer.

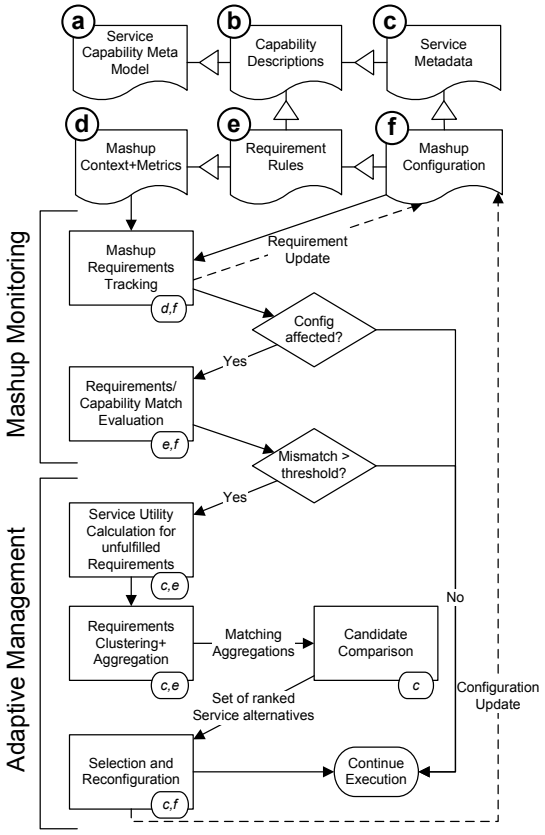

Fig. 1. Mashup Adaptation: the top icons represent data models (a-f), with arrows depicting dependencies. Throughout the adaptation process each step references the involved data.

Proximity to the service may reduce the response time which is important for highly reactive Web applications.

We also include specific metrics that describe the interactions and structural properties of collaborative e-professionals from our previous work [2].

III. Service Capability Model

Service capabilities describe behavior properties which cannot be directly derived from the service's WSDL interface. Example properties include limitations on simultaneous service use, supported resource access strategies, or reconfigurability. Capabilities usually change when a service undergoes major modification. Adding a new operation or extending service back-end resources provides new or better functionality. Services might also choose to reduce capabilities to remain available in spite of high load. This graceful degradation allows service clients to balance limited functionality and the cost of finding and invoking an alternative fully functional service.

Our *Service Capability Meta Model* (Fig. 2) shares some similarities with the *Composite Capability/Preference Profiles* (CC/PP) specification [3]. The original purpose of CC/PP foresees clients to transmit their capabilities in order to enable service providers to adapt delivered content accordingly. In contrast, our approach allow services to describe their capa-

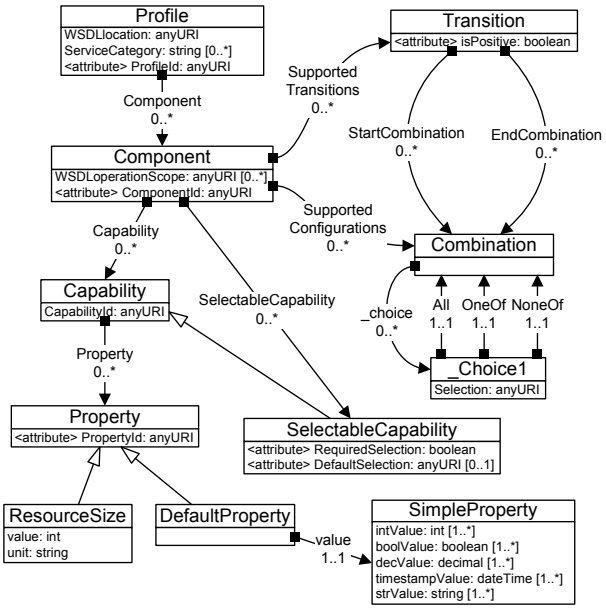

Fig. 2. Capability meta model UML class diagram

bilities to enable service clients to select the most suitable service.

The service capability model applies the concepts of *component* and *property* but goes beyond describing simple service characteristics. Selectable capabilities and supported capability reconfigurations are the main distinct differences to the CC/PP model. These properties are key to reconfiguration and adaptation. The capability meta model defines the following elements:

Profile: contains all capabilities of a single service. The *WSDLlocation* identifies the corresponding service instance. The *ServiceCategory* tags a service with (potentially domain-specific) categories. A profile consists of one or more *Components*.

Component: describes a certain functional or non-functional aspect of a service. A notification service, for example, will distinguish amongst publication related capabilities and subscription related capabilities. A component specifies regular capabilities, selectable capabilities, and supported configurations on the selectable capabilities.

Capability: comprises properties and optionally sub capabilities. Properties state the actual capability details while sub capabilities enable further structuring.

Property: identifies and provides details such as maximum number of requests per minute. The meta model defines five basic properties for integer, decimal, boolean, timestamp, and string values. FileSize is an example complex property comprising size unit (e.g., kB, mB) and size value.

SelectableCapability: describes capabilities that need selection and (optionally) configuration before they become available to the client. The list of alternative capabilities consists of regular capabilities or again selectable capabilities.

Combination: defines valid combinations of selectable capabilities. Note, the SelectableCapability element only defines the set of available choices. *Selection* identifies a SelectableCapability (i.e., representing a set of capabilities) or a single Capability.

Transition: describes valid reconfiguration paths. Specifically, a transition contains a minimum of one start configuration (i.e., a Combination) and a minimum of one reachable end configuration. A set of positive transitions explicitly lists allowed reconfigurations, and vice versa.

The *Combination* element exhibits some similarity with WS-Policy [3]. However, WS-Policy neither supports description of a service's utility, nor dynamic reconfiguration during runtime.

IV. Requirements Rules

Mashup requirements depend on the current context and define a desirable mashup configuration. To this end, we apply event-driven rules. A requirement rule describes metric conditions and subsequent constraints applied to a particular capability. Changes in mashup metrics trigger corresponding rules which then define optimum service capabilities (i.e., constraints). In the proceeding sections, we then compare deployed services with calculated constraints and compose the best reconfiguration plan given the available capabilities.

When designing rules, we have to consider a number of challenges. First, different mashups will exhibit different metrics. Thus, rules cannot rely on having all metrics available. Second, mashups have heterogeneous goals which reflect in customized additional rules and removal of nonessential rules. Rules must not rely on other rules being active or available. Given the complexity and heterogeneity of requirements, tight coupling of rules is not an option. Third, we need to provide the most fitting services regardless of the requirement fulfillment level available services exhibit. When services lack the required optimum capabilities, we need to find services that support the next most important requirements. Consequently, rules need to enable smooth degradation of provided capabilities.

To this end, we design loosely coupled, weighted rules. Rules depend only on metrics, they do not reference any other rule. Fine-grained rules do not override coarse-grained rules. Instead, they generate constraints of higher importance (i.e., constraints exhibiting a higher weight). When two requirements (not necessarily from the same rule) constrain the same capability, the more important one takes precedence. This mechanism is vital to smooth degradation. When the most significant constraint cannot be satisfied, the next most important constraint becomes active. An example requirements rule is available online (SOM [4]).

A requirement rule specifies the following elements:

Rule Identifier: enables requirement tracing. All constraints generated by the same rule carry the same rule identifier.

Metric Conditions: trigger the generation of constraints. Rules can aggregate any number of metrics, but must refrain from applying results generated by other rules.

[3] http://www.w3.org/TR/ws-policy-primer/

Capability Identifier & Property Identifier: determines the property within a capability.

Utility Function Identifier: defines the candidate comparison function. Supported functions include various set overlap and threshold-based calculations.

Utility Function Parameters: for linear functions, the parameters provide the limits. For set functions, the parameters list the required capability elements.

Weight: describes the importance of constraints. More specific constraints yield higher weights than general constraints.

We aggregate constraints on identical capabilities and then sort constraints in descending order of weight. Figure 3 visualizes the relations between metrics, rules, constraints, and constraint aggregation.

A. Requirements Filtering

Loose coupling of requirements rules renders the rule engine unaware of multiple requirements constraining the same capability. Matching capabilities requires, therefore, prior filtering of multiple—potentially conflicting—constraints on the same capability.

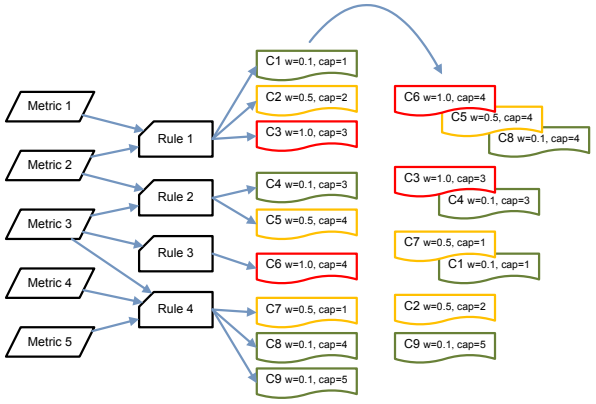

Fig. 3. Metrics triggering rules which in turn generate constraints on capabilities (*cap*) with weight *w*.

The *Gracefully Degrading Matching Algorithm* determines which constraint comes into operation. For sake of simplicity, suppose that each service profile consists of a single component. Further, let us define the set of candidate components $s \in S$ that we collect from all available service profiles. We capture the constraints aggregated for identical capabilities in $RL = \{R_1 \dots R_n\}$ such that all constraints $c \in R_i$ concern capability i. Each constraint c provides the details as outlined in the previous section.

We evaluate requirements in descending order of weight within each requirement list R. We store the utility values for each constraint and component in the utility matrix $\mathcal{U}$. If no capability fulfills the top requirement in R, we remove that requirement and evaluate the next highest. Once we have identified a requirement that can be fulfilled by at least one service, we drop all other less important constraints (i.e., those with lower weight) on the same capability. Ultimately, each requirement list $R \in RL$ contains only a single requirement for each capability. The top requirements in RL become the set of constraints in the subsequent requirements cluster analysis.

V. Capability Constraints Clustering

We extend the service matching and ranking approach to provide the optimum set of services matching the required constraints. When services exhibit complementary capabilities, two or more services compensate for their individual shortcomings. The challenge is to determine the optimum number and member of groups that best represent capability complementarity.

The goal of clustering algorithms is distributing data elements into a set of meaningful partitions. They fall into two main categories: assigning each data element to exactly one particular cluster (hard clustering), or assigning data elements to multiple clusters (soft clustering). We focus on the latter category of fuzzy clustering algorithms for grouping constraints according to implicit service groups.

Fuzzy C-Means (FCM) [5] associates each data element x_i to every cluster $k_{j=1\dots z}$. The data element x_i contains all service utility values for constraint i. We determine the best cluster count as suggested in [6], similarly giving equal weight to cluster compactness and cluster separation.

The membership matrix $\mathcal{M}_{ij}$ is the result of the clustering process. It describes the degree of data element x_i (i.e., constraint c_i) belonging to a particular cluster k_j, such that $\sum_{j=1\dots z} \mu_{ij} = 1$. We establish the importance weight v_i for each cluster by aggregating the individual constraints weights w_j according to their membership degree μ_{ij}. Thus clusters comprising of predominantly significant constraints yield in turn high importance.

We need to rank each service in each cluster to determine the top candidates. The scores reflect the clusters importance. Thus services performing well in important clusters yield higher scores than service fairing well in less significant clusters. To this end, we update the constraint weights w according to their membership degree μ and corresponding cluster importance v such that $biasedW_i = w_i * \mu_{ik} * v_k$. We thereby avoid having to defuzzify the clustering result. A constraint belonging equally to two clusters will thus influence the ranking result in both clusters to the same degree. We subsequently apply the basic logic scoring of preferences (LSP) ranking algorithm [7] to aggregate the utility values for each service according to the biased constraint weights. The resulting service scores establish the final ranking order within each cluster.

VI. Evaluation

A. System Design

We briefly outline the recommendation system's design before presenting the evaluating case study. The framework presented in [2] serves as basic context sensing component (Fig. 1d). Previous work in the inContext project[4] provides service registry and user interface components.

[4] http://www.in-context.eu/

ID	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	K_1	K_2	K_3	R_{non}	R_{K1}	R_{K2}	R_{K3}
c_1	80	90	50	0	0	0	5	5	0	0	0.03	0.03	**0.94**	S_7(41.5)	S_7(70.04)	S_4(70.29)	S_1(56.88)
c_7	50	70	90	0	0	0	0	0	0	0	0.05	0.04	**0.91**	S_1(32.0)	S_8(48.84)	S_6(57.22)	S_2(56.29)
c_8	60	40	30	0	0	0	0	10	0	5	0.08	0.06	**0.86**	S_4(32.0)	S_{10}(38.84)	S_5(56.64)	S_3(48.24)
c_2	0	0	0	20	5	5	90	90	70	60	**0.80**	0.12	0.08	S_8(28.5)	S_9(35.12)	S_7(39.3)	S_7(13.83)
c_3	0	10	10	0	0	0	70	30	0	40	**0.81**	0.09	0.10	S_{10}(27.5)	S_4(19.35)	S_{10}(34.13)	S_8(12.12)
c_6	0	0	0	0	0	0	70	50	20	20	**0.92**	0.04	0.04	S_5(26.0)	S_5(14.77)	S_9(30.25)	S_9(9.45)
c_4	70	20	30	60	95	30	70	60	90	40	0.33	0.47	0.20	S_6(25.5)	S_6(14.41)	S_1(26.88)	S_{10}(9.38)
c_5	30	5	0	60	20	80	80	20	50	60	0.34	0.54	0.12	S_9(25.0)	S_1(12.81)	S_8(23.41)	S_4(8.80)
c_9	30	10	0	90	60	60	30	20	0	30	0.04	**0.93**	0.03	S_2(24.5)	S_3(7.67)	S_2(9.54)	S_5(8.60)
c_{10}	0	0	0	90	80	80	0	0	20	20	0.09	**0.84**	0.07	S_3(21.0)	S_2(7.65)	S_3(6.96)	S_6(6.92)
										v	0.35	0.32	0.33	ρ	0.62	0.48	−0.01

TABLE I

SERVICE UTILITY VALUES u (FROM INTERVAL $[0, 100]$) , CONSTRAINT CLUSTERS (CRISP MEMBERSHIP VALUES μ IN BOLDFACE), AND RANKING RESULTS r (SCORES IN BRACKETS).

Our work in this paper adds (i) a capability management service for defining and providing the capability metadata on all registered services (Fig. 1b+c), (ii) a requirement management service for specifying context-dependent constraints (Fig. 1e), and (iii) a mashup configuration service for storing the applied services and constraint for a particular mashup instance (Fig. 1f).

B. Case Study

Our evaluation builds on the core adaptation step in our motivating scenario. We focus on a few context changes, corresponding requirements, and ultimate clustering result.

In the beginning, few constraints affect the selection of suitable services. The users share the same goal, exhibit high trust, and reside in close proximity. The mashup developer assembles services that yield little configuration overhead. Amongst the required capabilities, we find:

- Asynchronous communication capability (c_1).
- File versioning capability (c_2).
- Resource conflict resolution capability (c_3).
- Storage accounts for all users (c_4).
- Storage space for each user (c_5).

The mashup context considerably changes once the development effort includes users from multiple organizations that are physically distributed. Additional capabilities are required to handle the complexity that comes with the rising number of mashup users. This gives rise to additional constraints.

In distributed environments, storage services improve performance when they provide their *content in proximity* to the individual users (c_6). In addition, these users require *synchronous communication* capabilities (c_7). Team leaders have difficulty assigning and monitoring work tasks without dedicated *delegation support* (c_8). Involvement of multiple organizations demands services which can *virtually separate storage* places (c_9). (SOM [4] provides the corresponding requirement rule, respectively capability description). To this end, access to resources should be *organization-centric* (c_{10}).

Table I (left part) contains the utility values of services $S_1 \dots S_{10}$ partially matching the constraints $c_1 \dots c_{10}$. For this experiment, we assume equal constraints weights $w = 0.1$. Services $S_1 \dots S_3$ exhibit predominantly communication capabilities. Services $S_4 \dots S_6$ specialize in organization-aware access control, while services $S_7 \dots S_{10}$ provide mainly resource management capabilities.

The central part of Table I provides the constraint clustering result (applying fuzzy factor $m = 2$). We obtained three clusters of roughly the same importance (v). Communication and coordination capability constraints (c_1, c_7, c_8) yield crisp membership results. Similar, resource versioning, conflict resolution, and distributed storage (c_2, c_3, c_6) populate a distinct cluster, as do constraints c_9 and c_{10}. The real benefit of soft (as opposed to hard) clustering becomes apparent when we observe the cluster membership values for constraint c_4 and c_5. Every service exhibits to some degree capabilities that describe the number of supported users, and respective storage size. Consequently, these constraints lack a crisp cluster membership and subsequently have a similar effect on the final ranking process within all three clusters.

C. Discussion

We provide the final ranking results within the three clusters in the right part of Table I. Column R_{non} contains the non-clustered ranking scores. Clustering yields two main advantages:

First, the combination of top services from each cluster greatly outperforms the single top non-clustered service. The set of S_7, S_4, and S_1 yield an average utility score of 65.8 compared to 41.5 when selecting only S_7 from R_{non}.

Second, services that rank rather low in R_{non}—such as S_3 at position 10—potentially advance to the top candidates in a clustered ranking set (e.g., S_3 winds up at position 3 in R_{K3}). Thus, specialized services are more likely to become part of a mashup.

In our example, the three top services from R_{non} are the top clustered services. Confronted with no clustering information, however, the mashup developer would have to consult the service capabilities in detail to understand the differences of the top services. Furthermore, the mashup developer would remain unaware of the number of clusters and it's not guaranteed that the top elements always yield complementary capabilities.

Besides comparing ranking scores, we qualitatively evaluate the clustering result with Pearson's correlation coefficient ρ. ρ describes the correlation of the elements' sequence in two ranking sets. Sets with $\rho = 1$ yield identical order, while sets

with $\rho = -1$ exhibit inverse order. Table I provides the pairwise calculated coefficients between R_{non} and $R_{K1...K3}$. Cluster K_1 exhibits some, cluster K_2 little, and cluster K_3 no correlation. We can thus conclude that constraint clustering revealed a distinct structure in the service utility values that is beneficial for mashup adaptation.

VII. RelatedWork

The majority of mashup research efforts focuses on integration support [8]. Rosenberg et al. [9] introduce a lightweight composition language to integrate RESTful services into mashups. Maximilien et al. [10] describe a domain-specific language to create mashups that can be easily reused by other developers. Such collaborative design is also central to *expressFlow* by Vasko and Dustdar [11]. Their framework provides abstract composition models, which are then automatically transformed into various executable mashup languages. SOAlive [12] applies some of these ideas to service description, discovery, and composition.

To the best of our knowledge, dedicated recommendation support for mashup development has gained little attention yet. Blake and Nowlan [13] investigate service similarity measures based on syntactical message analysis to recommend suitable services. Their algorithm is, however, incapable of distinguishing between services that apply similar data structures but provide completely different capabilities. Ranabahu et al.[14] take this idea a step further and propose a faceted classification based approach.

More advanced adaptation 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 [15], or trust and reputation [16], [17] that equally apply to any service. Our capability model focuses rather on the gap between QoS and service interface descriptions.

Our combination of capabilities and requirement rules has similarities with goal-oriented service adaptation. Services get replaced when goals are no longer met. Examples such as [18], [19], [20], however, analyze context changes merely to determine when to reconfigure the composition. In contrast, we apply context to update the relevant set of goals.

VIII. Conclusion

Service mashups designers need context-aware adaptation recommendation in dynamic environments. Our approach matches service capabilities against dynamically updated capability requirements. The resulting utility values enable fuzzy constraint clustering to identify groups of complementary capability constraints. We have demonstrated that selecting the top elements from multiple clusters offers a benefit when two conditions hold: First, the clusters yield a distinctly different ranking sequence than the non-clustered ranking result; and second the average utility of the top clustered elements is significantly larger than the utility of the top non-clustered element. The presented clustering algorithm yields promising results, however, it does not consider aggregation costs yet. This will be subject to future work. In addition, we plan to automatically derive applicable requirements from the service operations that are invoked during mashup execution.

References

[1] F. Skopik, D. Schall, and S. Dustdar, "The cycle of trust in mixed service-oriented systems," August 2009.

[2] C. Dorn, H.-L. Truong, and S. Dustdar, "Measuring and analyzing emerging properties for autonomic collaboration service adaptation," in *5th International Conference on Autonomic and Trusted Computing (ATC)*. Springer LNCS, June 2008.

[3] C. Kiss, "Composite capability/preference profiles (cc/pp): Structure and vocabularies 2.0," http://www.w3.org/TR/CCPP-struct-vocab2/, W3C, Tech. Rep., 2007.

[4] "SOM: Supporting online material," http://www.infosys.tuwien.ac.at/Staff/dorn/SOM/APSCC.html.

[5] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981.

[6] J. He, A.-H. Tan, C.-L. Tan, and S.-Y. Sung, *On Quantitative Evaluation of Clustering Systems*. Kluwer Academic Publishers, 2003. [Online]. Available: citeseer.ist.psu.edu/he02quantitative.html

[7] J. J. Dujmovic, "Continuous preference logic for system evaluation," in *IEEE Transactions on Fuzzy Systems*, vol. 15, no. 6. IEEE Computer Society, 2007, pp. 1082–1099.

[8] D. Benslimane, S. Dustdar, and A. Sheth, "Services mashups: The new generation of web applications," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 13–15, Sept.-Oct. 2008.

[9] F. Rosenberg, F. Curbera, M. Duftler, and R. Khalaf, "Composing restful services and collaborative workflows: A lightweight approach," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 24–31, Sept.-Oct. 2008.

[10] E. Maximilien, A. Ranabahu, and K. Gomadam, "An online platform for web apis and service mashups," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 32–43, Sept.-Oct. 2008.

[11] M. Vasko and S. Dustdar, "Introducing Collaborative Service Mashup Design," in *Lightweight Integration on the Web (ComposableWeb'09*. CEUR - Workshop Proceedings, June 2009, pp. 51–62.

[12] I. Silva-Lepe, R. Subramanian, I. Rouvellou, T. Mikalsen, J. Diament, and A. Iyengar, "Soalive service catalog: A simplified approach to describing, discovering and composing situational enterprise services," in *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 422–437.

[13] M. Blake and M. Nowlan, "Predicting service mashup candidates using enhanced syntactical message management," in *Services Computing, 2008. SCC '08. IEEE International Conference on*, vol. 1, July 2008, pp. 229–236.

[14] A. Ranabahu, M. Nagarajan, A. P. Sheth, and K. Verma, "A faceted classification based approach to search and rank web apis," in *IEEE International Conference on Web Services (ICWS)*. IEEE Press, 2008, pp. 177–184.

[15] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar, "Towards composition as a service - a quality of service driven approach," 29 2009-April 2 2009, pp. 1733–1740.

[16] L.-H. Vu, M. Hauswirth, and K. Aberer, "Qos-based service selection and ranking with trust and reputation management," in *OTM Conferences (1)*, 2005, pp. 466–483.

[17] E. Maximilien and M. Singh, "Self-adjusting trust and selection for web services," June 2005, pp. 385–386.

[18] T. Yu and K.-J. Lin, "Adaptive algorithms for finding replacement services in autonomic distributed business processes," in *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, April 2005, pp. 427–434.

[19] D. Greenwood and G. Rimassa, "Autonomic goal-oriented business process management," in *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2007, p. 43.

[20] S. H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul, "Supporting the dynamic evolution of web service protocols in service-oriented architectures," *ACM Trans. Web*, vol. 2, no. 2, pp. 1–46, 2008.