

Quality-based Recommendations for Mashup Composition

Matteo Picozzi, Marta Rodolfi, Cinzia Cappiello, and Maristella Matera

DEI - Politecnico di Milano
Via Ponzio 34/5, 20133 Milano, Italy
[matteo.picozzi,marta.rodolfi]@mail.polimi.it;
[cappiell,matera]@elet.polimi.it

Abstract. When composing mashups, the selection of suitable services is mainly based on functional requirements and does not consider the quality of the single services. In this paper, we show that the quality of component services can drive the production of recommendations that can help building quality mashups. We capitalize on a quality model for mashup services and discuss the concept of *mashability*, a multi-dimension quality property that expresses the capability of a component to maximize the quality of a mashup, and the concept of *role-based composition quality*, i.e., the quality of mashup compositions weighted according to specific roles that the composed services play within the mashup. We then show how such concepts can enable the production of quality-based recommendations for the mashup design.

1 Introduction

Web mashups are a new generation of applications that support the “composition” of applications starting from contents and services provided by third parties and made available on the Web [6, 17]. Mashups were initially conceived in the context of the consumer Web, as a way for end-users to create their own applications. More recently, enterprise mashups started to emerge, as instruments for the average (i.e., not necessarily technically-skilled) users to easily and quickly create *situational* applications that can serve, even for a short period, to support their decisional processes [12]. Both the previous contexts are characterized by the involvement of end users. However, with very few exceptions, so far the research in the mashup field has focused on the technical issues, such as formats for data exchange, interoperability, etc., while very little attention has been devoted to easing the development process. In many cases, indeed, mashup composition still results into the manual programming of the service integration.

Recently, some approaches have been proposed to ease the mashup composition task. On the one hand, some tools try to reduce the efforts required to program the service composition, by proposing high-level abstractions, also sustained by visual easy-to-use notations [9, 5]. On the other hand, some other approaches focus on the production of recommendations about the component services and the composition patterns to be adopted [8, 16]. The approach we

discuss in this paper goes into the direction of easing the mashup composition by means of recommendations; its characterizing feature is that it takes into account quality, both component services quality and the overall mashup quality, as a key factor to drive the production of recommendations.

We capitalize on a previous model for the quality of mashup components [2], and identify how the quality of single components, expressed along several quality attributes, can be used to guide the selection of services and compositions. We in particular discuss the concept of *mashability*, a multi-dimension quality property that expresses the capability of a component to maximize the quality of the final mashup, and the concept of *role-based composition quality*, i.e., the quality of mashup compositions weighted according to specific roles that the composed services play within the mashup. We then show how these concepts can support the production of recommendations within a “quality-aware” mashup development.

The paper is organized as follows. In Section 2 we illustrate the typical scenario for mashup development, with the purpose of highlighting different quality issues, and the consequent need for quality-based recommendations. We then illustrate our model for mashup quality: we summarize our previous model [2] for the quality of mashup components (Section 3), and define the new perspectives of *mashability* (Section 4) and *role-based composition quality* (Section 5), which are the basis for the generation of quality-aware recommendations. Inspired by the recommendation model proposed in [8], in Section 6 we then show how our quality model can support the production of recommendations. We finally describe the most relevant related works (Section 7), and draw our conclusions (Section 8).

2 Quality issues in the mashup development process

There is a lack of proposals for the quality of mashups. In a sense, this is because, being Web applications, mashups can be mainly characterized by the external quality-in-use perspective, which is exhaustively covered by the huge research on Web application usability [13]. We however believe that, beyond quality in use, other issues must be considered, which are strictly related to the activities that characterize the mashup development process.

The typical scenario for mashup development spans from the production of single *mashup components* to the integration of selected components into a final *mashup composition*. Given the nature of mashups as applications integrating other resources, throughout the whole process the quality of the component resources and the adopted composition patterns play a fundamental role. We identify three stages in which the quality of component services comes into play.

The component developer creates component services for mashups. We assume that developers correctly implement the service functionality, taking into account well-known principles, best practices and methodologies for guaranteeing the internal quality of the code. However, when used in a mashup composition, component services can be selected by considering especially some external prop-

erties. Since our aim is to investigate the quality of the final mashups, this is the perspective we are interested more, which is related to aspects such as the architectural style (e.g., SOAP services vs. RESTful services vs. widget APIs), the adopted programming language (e.g., client side such as JavaScript vs. server side such as Ruby), the data representation (e.g., XML vs. JSON), the component operability and interoperability (e.g., the multiplicity of APIs targeting different technologies). Such external aspects indeed affect the “appeal” of the component from the mashup composer perspective. The component developer should try to maximize them, and should also make these quality properties visible for the mashup composers, who can therefore base on them the choice of components. The component developer therefore builds the component having in mind the quality properties to be maximized. She can also document such quality properties by means of suitable component descriptors, possibly complementing a documentation for the component use (at least, ideally).

When composing a mashup, *mashup composers discover components*, directly from the Web or from repositories accessed from the adopted mashup tool, taking into account the fitness of each component for its purpose within the mashup (i.e., its functional requirements), but also the complexity of its technological properties (e.g., a simple programming API, languages and data formats enhancing operability and interoperability), as well as the richness and completeness of the provided data. The quality of single services can drive their selection, so enabling the composer to select components based on the provided quality.

At the end of the service selection phase, when a first committed composition is ready, quality assessment can take advantage of the defined composition model, to identify the role and importance that component services play within the composition. The quality of mashups can thus be revisited and quantified as an aggregation of the quality of the single components, weighted however on the basis of the component role and the adopted composition patterns. Therefore, once a composition model is available, some recommendations can be issued, about adding further components, or about alternative *similar* compositions able to increase the quality of the final mashup.

Addressing the previous quality concerns requires the definition of sound models, first of all focusing on the quality of components as stand-alone ingredients, but also on their attitude to generate quality mashups when combined with other components.

3 Quality of Mashup Components

Publishing mashup components through APIs or services hides their internal details and gives more importance to their external properties [18]. In line with this black-box view, in [2] we proposed a quality model for mashup components that privileges properties of the component APIs - this is indeed the perspective that is most relevant to the mashup composer or the mashup user. The model is based on both our own experience with the development of components and mashups [17], and on experimental evidence gathered by analyzing data from

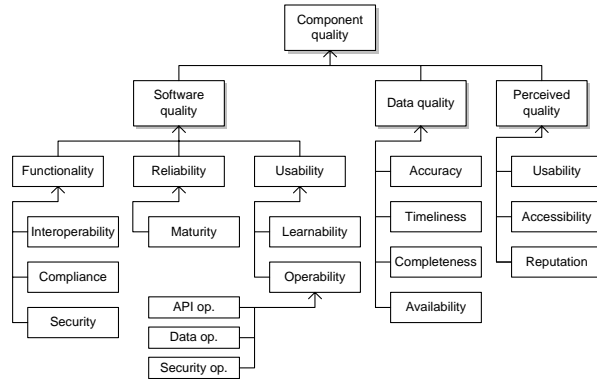


Fig. 1. The quality model for mashup components [2].

programmableweb.com [2]. It is organized along three main dimensions recalling the traditional stratification of Web applications into data, application logic and presentation layer:

- *Data quality* focuses on the suitability of the data provided by the component in terms of accuracy, completeness, timeliness, and availability.
- *API quality* refers to software characteristics that can be evaluated directly on the component API. We split API quality into *functionality*, *reliability*, and *API usability*.
- *Presentation quality* addresses the user experience, with attributes such as *presentation usability*, *accessibility*, and *reputation*.

Figure 1 summarizes the quality dimensions, their attributes, and the finer-grained characteristics addressed by our model. For more details on the associated metrics the reader is deferred to [2].

4 Mashability

The previous model focuses on single components. *Mashability* is still a property of single components, but it expresses the measure in which the quality characteristics of a component would improve the mashup being created. It therefore puts the quality of each single component into the dynamic perspective of the mashup composition, and can be seen as the combination of *compatibility* and *aggregated quality*.

Compatibility estimates whether the combination of one component with those already included in a composition is possible, based on:

- *Technology compatibility*, to assess whether two or more components are compatible from the point of view of the adopted protocols, languages and data formats;

- *Syntactic compatibility*, to check the type compatibility between the input/output parameters exposed by the component services;
- *Semantic compatibility*, to check whether input/output parameters and involved operations belong to the same or similar semantic categories, assuming that the syntactic compatibility is satisfied.

Compatibility is not properly an expression of the quality of the final mashup; it is however a source of recommendations for the mashup composer about the selection of compatible component services, so as a correct composition is produced. The notion of *aggregated quality* is therefore needed, as an estimation of the final mashup quality achieved by aggregating the quality of individual component services. It is in this context that the quality of component services comes into play. As compatible services are selected and added into the mashup, their quality indicators are aggregated to estimate in which way they influence the quality of the final mashup.

In summary, mashability helps ranking the available components, based on their capability to be syntactically combined with previously selected components and to maximize the quality of the overall mashup. It is therefore a dynamic estimation of quality, which progressively takes into account the component services' attitude to be composed into quality mashups.

5 Role-based Composition Quality

Mashup quality is not simply an aggregation of the quality of the individual components; therefore assessing and aggregating the quality of each component service is not enough. Mashup quality depends on the particular combination of components into a composite logic, layout and, hence, user experience. Component services, especially those provided with a user interface and therefore visible in the mashup, may play different roles that affect the perception of the quality of the final integration, and must therefore be carefully taken into account. By analyzing the most popular mashups published on programmableweb.com we have identified three typical roles [4]:

- *Master*: Even if a mashup integrates multiple components in one page, in most cases one component is more important than the others, being the service the user interacts with the most. It usually is the starting point of the user interaction that causes the other components to react and synchronize accordingly.
- *Slave*: The behavior of a slave component depends on another component; its state is mainly modified by events originating in another (master) component. Many mashups also allow the user to interact with slave components. However, the content items displayed by slave components are selected via the user's interaction with the master component, and by automatically propagating synchronization information from the master to the slaves.
- *Filter*: Filter components allow the users to specify conditions over the content shown by the other components. They provide (possibly hierarchical)

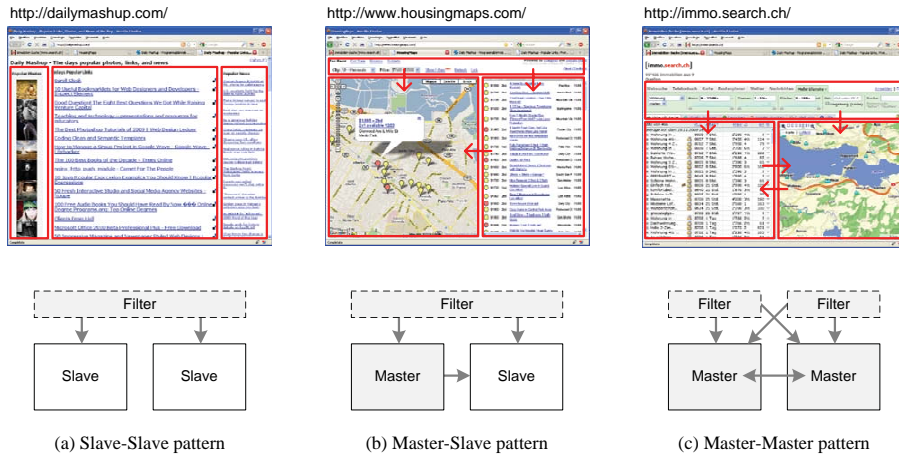


Fig. 2. Basic mashup development patterns [4].

access mechanisms that allow the users to incrementally select the contents they want to see. They also reduce the size of the data set shown by the other components, thus improving the mashup understandability and ease of use. In most cases, filter conditions are specified over the data set of master components, while slaves are synchronized and, hence, their content is automatically filtered by the integration logic.

Figure 2 highlights three basic patterns based on the previous roles of component services. The roles impact mashup quality. For example, in the slave-slave pattern represented in Figure 2(a), filters have minimal influence over the perceived quality of the components: being oftentimes developed by the mashup developer, they do not allow the specification of filter conditions that cannot be satisfied by the slave component. In other composition patterns, such as the master-slave and the master-master patterns depicted in Figure 2(b) and 2(c), the master is the major responsible of the final quality; it could even degrade the quality of the other components. The perceived quality of the final composition can be therefore achieved as an aggregate of the quality of individual component services, but weighted taking into account the services roles.

In order to describe the roles of component services, a mashup composition can be modelled as an oriented weighted graph $G = (V, E, W)$, where each vertex $v_i \in V$ represents a component service; each arc $e_{ij} \in E$ is associated with a binary value that represents that one or more bindings are defined between the two connected components to couple v_i output parameters with v_j input parameters; the arc weight, $w_{ij} \in W$, represents the actual number of mappings defined between the two connected components. Informally, the role, and thus the relevance, of a specific vertex v_i in a specific mashup can be estimated by considering all its direct and indirect bindings in all the paths that link v_i to a vertex v_k . In fact, we assume that the influence of one component on the total

quality of the mashup depends on the number of its total bindings towards other components. More details about the identification of roles and the computation of quality weights can be found in [3].

6 The model for mashup recommendation

Preliminaries. Based on the quality concepts introduced in the previous sections, in this section we propose a model for the production of recommendations able to support the mashup composer in the recognition of the most suitable components to use from a functional and non-functional point of view. Let us assume that the mashup composer could access a component registry C in which each component $c_i \in C$ is associated with a *component descriptor* and a *quality vector*. The component descriptor lists all the offered operations and related parameters and all the technical details needed to evaluate the quality dimensions described in Section 3. The quality vector $QD_i = [qd_{i1}, qd_{i2}, \dots, qd_{in}]$ contains the list of metrics associated with the quality dimensions. Starting from the quality vector, it is possible to define the value of the component quality cq_i as the aggregated quality index for the i -th component.

Quality assessment may play a fundamental role when selecting component services during the mashup development process. In fact, in case two components c_i and c_j are similar, that is functionally equivalent, quality can be a valuable discriminant factor. For this reason, it is worth to introduce the concept of *quality distance* between two mashup components as the absolute value of the difference between the quality of c_i and the quality of c_j :

$$Dist(c_i, c_j) = |cq_i - cq_j|$$

Besides the component registry, the mashup composer could also access a mashup registry M , in which previously composed “ready-to-use” mashups are stored. For each mashup $m_k \in M$, also assuming the availability of a composition descriptor, it is possible to access the list of the components $C_k \subset C$ used to create it. Additionally, given the description of the composition, it is possible to derive the composition graph and identify the pattern on which the composition is based, as described in Section 5. According to the approach presented in [8], each mashup could also be associated with an *Importance* index Imp_k that defines the mashup’s reputation. Importance is calculated taking into account the *mashup popularity* mp_k , that is the frequency of adoption by the community of users, and the *user rating* mr_k as proposed in [8]. We refine the notion of Importance, also introducing the *mashup quality* mq_k ; therefore, given a mashup, its Importance is given by:

$$Imp_k = \alpha \cdot mq_k + \beta \cdot mp_k + \gamma \cdot mr_k$$

where α , β and γ are weights that express the relevance of each dimension, and $\alpha + \beta + \gamma = 1$.

The *mashup quality* is the characteristic feature of our approach. It is an aggregated quality measure that is calculated on the basis of the quality of the mashup components and of the patterns used to connect them [4].

The recommender algorithm. Considering the mashup development process described in Section 2, in this paper we aim at providing support to the mashup composer at the end of the component selection phase, or when a preliminary composition is available, i.e., when some components have been already included within the mashup. This allows us to identify the goal of the composition. In fact, our approach analyzes a mashup under construction m^* and provides recommendations about possible alternative compositions and/or extensions.

The main steps of the quality-driven recommender algorithm are four: (i) Discovery and evaluation of similar components; (ii) Discovery of similar mashups; (iii) Quality distance evaluation; (iv) Mashups ranking.

Discovery and evaluation of similar components. As soon as the mashup designer defines a composition m^* (final or intermediate), the first step of the quality-aware recommender algorithm has the goal to retrieve all the components that are similar to the components used in the mashups $C^* = [c_1^*, c_2^*, \dots, c_j^*]$. In detail, for each component c_j^* , the component registry C is analyzed in order to identify all the components that satisfy the mashability property, i.e., are syntactically and semantically compatible to c_j^* , and also improve the quality of the overall mashup. In this way, for each component c_j^* , an array of similar components $SC_j^* \subset C$ is created. Considering the set given by $c_j^* \cup SC_j^*$, it is possible to define the *ideal component* ic_j^* as the component that is associated with the maximum quality factor. The set of the ideal components IC_j^* then corresponds to the set of J components used in the *Ideal Mashup* im^* , that is the best mashup similar to the mashup m^* that the user can build according to his/her needs.

Discovery of similar mashups. The composition m^* is further analyzed for the retrieval of similar mashups. In details, the service registry M is analyzed in order to identify a set of mashups $SM^* \subset M$ in which each mashup $m_z \in SM^*$ contains at least J components equal or similar to the ones contained in m^* . A mashup $m_z \in SM^*$ could include also other components; it is however necessary that a similarity-based correspondence between the components $c_j^* \in m^*$ and the components $c_{zi} \in m_z$ is maintained.

Quality distance evaluation. As represented in Figure 3, for each mashup m_z in SM^* , the distance of a m_z from the ideal mashup is represented by the distance between a n -dimensional point p_{m_z} and the origin of axes, which is the ideal mashup. The point p_{m_z} describes the position of the particular mashup in the n -dimensional space and ranks the recommendations.

$$p_{m_z} = [1 - Imp(m_z), w_{z1} \cdot Dist(c_{z1}, ic_1^*), w_{z2} \cdot Dist(c_{z2}, ic_2^*), \dots, w_{zj} \cdot Dist(c_{zj}, ic_j^*)]$$

where $c_{zj} \in C$ are the components of m_z , ic_j^* are the ideal corresponding components, and w_{zj} are weights that express the following properties of components and composition patterns:

- the role of components in the mashup m_z as illustrated in Section 5;
- the similarity between c_{zj} , the m_z 's components, and c_j^* , the m^* components;

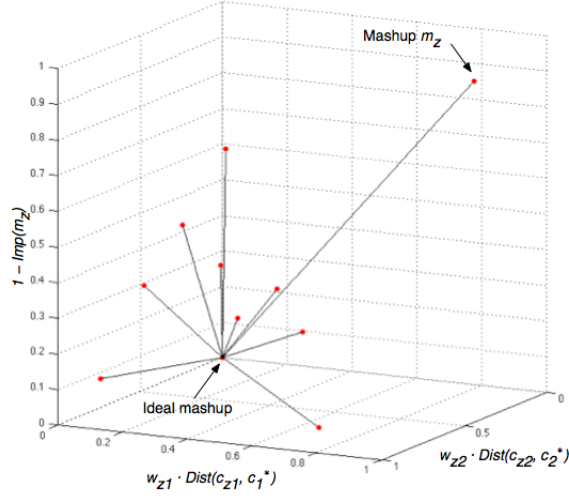


Fig. 3. An example of the vector space of similar mashups for a 2-component mashup. Segments represent the distances of the similar mashups from the ideal mashup.

- the position of each c_{zj} in the composition (e.g., let $c_1, c_2, c_3 \in C$, let \rightarrow the binding between two components, some possible compositions are $c_1 \rightarrow c_2 \rightarrow c_3$, $c_3 \rightarrow c_1 \rightarrow c_2$, $c_1 \rightarrow c_3 \rightarrow c_2$), so as to consider also different component bindings with respect to those defined in m^* , which however need to be downgraded through appropriate weights;
- the operations involved in the components binding in m^* with respect to the operation among components in m_z .

For the quality distance evaluation we need to consider: P_m the set of the p_{m_z} points, SC_j^* the set of the components similar to the m^* ones, and AC_z the set of the additional components which are the components present in m_z but not in m^* . We have introduced the two sets SC_z^* and AC_z to compute the weights w_{zj} , in order to give different importance to the components similar to the m^* ones, with respect to the others, because we want to penalize the components not in m^* , that do not match exactly the user's needs. Differently from the approach presented in [8] we however think that super-compositions should be considered.

Mashups ranking Each m_z is associated with a point p_{m_z} in a multidimensional space where the origin of axes is the ideal mashup. The distances of each point from the origin can be calculated as the L2-norm $\|p_{m_z}\|_2$. The resulting ranking will provide information about the quality of all the mashups that the users could use to obtain the same functionality of the composition m^* .

7 Related Works

Several works have proposed quality models for traditional Web applications (see for example [1, 14, 13]). Few proposals also concentrate on modern Web 2.0 applications. For example, in [15], the authors extend the ISO 9126-1 standard, and discuss the internal quality, external quality, and quality in use of Web 2.0 applications. To our knowledge, there are no works explicitly addressing the quality of mashup component services and mashup compositions.

Our model for component quality is derived from the quality attributes defined by the ISO quality standard. We however add a specific perspective, which allows us to concentrate on the external quality of components, i.e., on the set of properties that affect the component's quality as perceived by the mashup composer (not necessarily the final mashup user). Other works focused on API quality in the more general SOA (Service-Oriented Architecture) domain, by specifically addressing the set of external factors that increase the ease of use of an API (the so-called *API usability*) [7, 11]. Our approach capitalizes on these contributions but tries to go beyond, since it considers a broader set of external quality factors – not only usability –, all having impact on the success of mashup components.

Another novelty of our approach is that quality is the driving factor used for producing recommendations for mashup design. For service-based applications the literature already provides some approaches in which quality is the main driver for service selection and composition. For example, in [10] authors assess the overall quality of the final applications by aggregating the quality of the composing services. However, none of these approaches concentrates on the peculiarities of the mashup ecosystem.

Very few works have recently concentrated on *top-k* ranking of mashups (see for example [8, 16]), to help designers in the choice of components and composition patterns. In particular, in [8] the authors propose a technique for ranking ready-to-use mashups, to support the development process through a novel auto-completion mechanism. The approach is based on the discovery of the common properties (i.e., components and compositions) that characterize the mashups developed by different users, and identifies classes of mashups that are similar to the one under development based on the “syntactic” inheritance of component services and composition patterns. Similarities is thus exploited to predict the most likely potential completions. Our approach is inspired by this work, but extends it in several directions. The main innovation is the promotion of the quality of components and compositions, which is adopted to identify the *ideal mashup*, and to compute the distance-based ranking of similar mashups. The concept of similarity is also extended by means of the semantic compatibility of components and composition - not only the syntactic compatibility.

8 Conclusion

In this paper we have discussed the need of addressing quality issues in the mashup development process. We have illustrated how a quality model for mashup

component services can drive the selection of quality components, and can also be used to assess the quality of the overall mashup by means of new concepts, such as *mashability* and *role-based composition quality*. We have then shown how the new concepts can be exploited to generate design recommendations.

The approach described in this paper is still at an early phase. In order to prove the feasibility of the most fundamental assumptions, we have partly implemented it as an extension of the MashArt platform [5]. We have in particular defined descriptions for mashup components that provide values for the quality metrics computation and, starting from them, we have implemented the method for ranking component services based on the mashability property¹ [3]. We have also implemented the graph-based algorithm to analyze the quality of the composition based on the component roles. Our current and future work is devoted to finalizing the implementation of the recommendation technique, conducting experiments (also involving real users, i.e., mashup designers) to assess the effectiveness and efficiency of the enriched development process, as well as measuring the performance of the proposed algorithms.

References

1. C. Calero, J. Ruiz, and M. Piattini. A Web Metrics Survey Using WQM. In N. Koch, P. Fraternali, and M. Wirsing, editors, *ICWE*, volume 3140 of *Lecture Notes in Computer Science*, pages 147–160. Springer, 2004.
2. C. Cappiello, F. Daniel, and M. Matera. A quality model for mashup components. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *ICWE*, volume 5648 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2009.
3. C. Cappiello, F. Daniel, and M. Matera. Assessing mashup quality by looking at composition models and patterns. Technical report, Politecnico di Milano, 2010.
4. C. Cappiello, F. Daniel, M. Matera, and C. Pautasso. Information quality in mashups. *IEEE Internet Computing*, in print, 2010.
5. F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. Hosted universal composition: Models, languages and infrastructure in mashart. In A. H. F. Laender, S. Castano, U. Dayal, F. Casati, and J. P. M. de Oliveira, editors, *ER*, volume 5829 of *Lecture Notes in Computer Science*, pages 428–443. Springer, 2009.
6. F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007.
7. B. Ellis, J. Stylos, and B. A. Myers. The Factory Pattern in API Design: A Usability Evaluation. In *ICSE*, pages 302–312. IEEE Computer Society, 2007.
8. O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *PVLDB*, 2(1):538–549, 2009.
9. IBM. IBM Mashup center. www.ibm.com/software/info/mashup-center/.
10. M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl. Qos aggregation in web service compositions. In *EEE*, pages 181–185. IEEE Computer Society, 2005.
11. S. Y. Jeong, Y. Xie, J. Beaton, B. Myers, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse. Improving Documentation for eSOA APIs through

¹ Semantic similarity assessment is based on the use of the Wordnet 3.0 dictionary and the Pellet 2.1.0 ontology reasoner.

- User Studies. In *Proc. of the Second International Symposium on End User Development (IS-EUD09), March 2-4 2009, Siegen, Germany*.
12. A. Jhingran. Enterprise information mashups: Integrating information, simply. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 3-4. ACM, 2006.
 13. M. Matera, F. Rizzo, and G. T. Carughi. Web Usability: Principles and Evaluation Methods. In *Web Engineering*, pages 109-142. Springer, 2005.
 14. L. Olsina, G. Covella, and G. Rossi. Web Quality. In *Web Engineering*, pages 109-142. Springer, 2005.
 15. L. Olsina, R. Sassano, and L. Mich. Specifying Quality Requirements for the Web 2.0 Applications. In *Proc. of IWOST'08*, pages 56-62.
 16. M. A. Soliman, M. Saleeb, and I. F. Ilyas. Mashrank: Towards uncertainty-aware and rank-aware mashups. In *ICDE*, pages 1137-1140. IEEE, 2010.
 17. J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 923-932. ACM, 2007.
 18. S. Yu and C. J. Woodard. Innovation in the programmable web: Characterizing the mashup ecosystem. In G. Feuerlicht and W. Lamersdorf, editors, *ICSOC Workshops*, volume 5472 of *Lecture Notes in Computer Science*, pages 136-147. Springer, 2008.