Domain-specific Mashups: From All to All You Need

Stefano Soi and Marcos Baez
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento
Via Sommarive, 14 38123
Trento, Italy
{soi,baez}@disi.unitn.it

Abstract. Last years, aside the proliferation of Web 2.0, we assisted to the drastic growth of the mashup market. An increasing number of different mashup solutions and platforms emerged, some focusing on data integration (a la Yahoo! Pipes), others on user interface (UI) integration and some trying to integrate both UI and data. Most of proposed solutions have a common characteristic: they aim at providing non-programmers with a flexible and intuitive general-purpose development environment. While these generic environments could be useful for web users to develop simple applications, they are often too generic to address domain-specific needs and to allow users to develop real-life complex applications. In particular, proposed mashup mechanisms do not reflect those specific concepts that are proper of a given domain, which domain-experts are familiar with and could autonomously manage. We argue the need for domain-specific mashup architectures, also going beyond today's enterprise platforms, in which standard mashup mechanisms and components are driven by an underlying domain-specific layer. This layer will provide a service and component ecosystem built upon a shared and uniform conceptual model specific for the given domain. This way, domain experts will be provided with mashup components and mechanisms, following those well-known concepts and rules proper of the domain they belong to, that they are able to understand, use and, finally, profitably compose. In this paper, we will show the necessity of such an architecture through a real-life use case in the context of scientific publications and journals.

Keywords: domain specific mashups, vertical mashups, end-user centric mashups

1 Introduction

During last decade a vast amount of functionalities have been made available as online services, in form of Web Services, APIs, RSS/Atom feeds and so on. While these services can also be used independently from one other, putting them together to create a value-adding combination could lead to much more fruitful results, as described in [1]. This is exactly what mashup solutions try to achieve. In addition,

most of available mashup platforms aim at giving the possibility to develop such composite applications to domain experts, i.e. users with very limited programming skills but deep knowledge of the domain being the context of the problem to be solved. A number of studies (e.g., [2], [3]) discuss about benefits of moving the development of this kind of composite applications from IT-experts to non-programmers. This would be a radical paradigm shift bringing two main advantages, that is, first, avoiding requirement transfer from domain-experts to IT-experts and, moreover, allowing to face the development of situational applications[1], that is applications addressing transient or very specific needs for which the standard development lifecycle is not adequate since it would not be time- and cost-effective.

Current mashup building tools have the - non trivial - target of enabling domain-experts to develop such mashed up applications without - almost - any programming skill or any intervention of expert developers. This is often the main claim of available mashup platforms but, from our studies and experience in the mashup field, we found that this claim is only partially fulfilled. In particular, available mashup solutions provide easy mechanisms, suited for non-programmers, allowing them to produce very simple applications (often just "toy applications") or covering only some aspects of the integration needs of the users [4]. When users' needs go beyond this complexity level, available solutions show up their limitations.

In our opinion, main reasons underlying difficulties in overcoming these complexity limits are related to the fact that current mashup platforms (like [5],[6] and similar) have the ambitious goal of "integrating all the Web". In other words, they are not targeted at a specific domain but aim to give the possibility to make interacting user interfaces (UIs) and services coming from completely different domains and producers. For the time being, we think that "integrating all the Web" is a too ambitious goal. The lack of widely adopted - official or de facto - standards in this area make the integration of highly heterogeneous components a complex task that, at the end, is not sufficiently supported with mechanisms well-suited for non-programmers.

Starting from these considerations, we argue that there is the need for domain-specific mashup solutions. In particular, we propose to place the mashup system upon a layer defining concepts and policies of the given domain. We will see that this layer should include all the knowledge about the specific domain that could be then used to ease the mashup development, making actually possible to move the mashup application development from IT-experts to domain-experts.

Summarizing, the main contributions of this work are:

- stating the need for moving from horizontal general-purpose mashup solutions to vertical domain-specific ones, allowing domain-experts to autonomously compose their applications playing in their well-know playground
- proposing a modular architecture that makes a net separation between mashup layer and – pluggable – domain layer
- giving a first characterization of the “domain” concept, in terms of domain entities and rules, and first proposals on how the mashup platform should adapt to these.

[1] For details and references about the concept of situational application we refer to the Wikipedia page: http://en.wikipedia.org/wiki/Situational_application

To make our proposal clearer, we will explain the proposed solutions with the help of a use case, taken from the scientific publications context. In particular, we will make reference to a project our group is working on, called *LiquidPub* (http://liquidpub.org/), and we will show how the solution we propose could be profitably applied in that context. This will be the domain we will try to characterize and on which verticalize. We will base our example on a mashup tool coming from another project of our group, called *mashArt* (http://mashart.org/), which provides a complete mashup platform allowing for *Universal Integration* [7], that is seamless integration of data, services and user interfaces (UIs), targeted to non-programmer web users.

The rest of this paper is structured as follows. Section 2 will introduce and describe the motivating scenario, with particular reference to the LiquidPub project. In Section 3 we will see in more detail how generic mashup platforms work and what are their limitations. Section 4 will propose an architecture trying to overcome the issues presented in the previous section. In Section 6 will be presented the final conclusions and future work.

2 Motivating Scenario: Knowledge Dissemination

The Web has pushed forward technological and social changes in different areas and the scientific domain has not been the exception. It has opened a brand new world of possibilities for how the scientific knowledge can be consumed, produced, shared and disseminated. This has motivated an extensive research on how to exploit these opportunities, leading to novel *models*, new forms of *scientific contributions*, *metrics*, *services* and *sources of information*. Having a virtually infinite number of possibilities also implies that there could be different ways of consuming /disseminating /evaluating the scientific research work. The selection of the right configuration in terms of type of content (peer-reviewed papers, preprints, blogs, datasets...), the metrics (h-index, citation count, pagerank,..), sources (reputed publishers, open archives or the whole web) and the actual process will probably depend on the final usage scenario and the believes of the community. Implementing a particular dissemination model would normally require programming knowledge to produce the required code (e.g., in java, perl, ruby…), following a particular development process. Considering that everybody in the scientific domain has different thoughts on how to do this, it will be unnecessarily limiting to restrict this to programmers. It is clearly something end-users, or domain experts, should be able to do, and not only programmers.

Mashups provide the foundations for supporting such a scenario. However, current mashup platforms provide rather generic components, and so, at a level the scientist (non-programmer) cannot manage. For example, if a scientist strongly believes that i) blogs and open archives are valid dissemination venues, ii) sharing and consumption should be the main goal of a dissemination model, and therefore iii) sharing data should be used as base to evaluate researchers; presenting her a component that connects to a web service as primary tool does not help her in the composition of the dissemination model she believes in. Configuring and wiring components would become extremely complex (e.g., setting up a web service connection, or even

selecting the right web service), and the user would be required to provide the mapping in terms of I/O among heterogeneous web services (when the whole concept of "mapping" is probably obscure to her), to reflect a flow and a process at such low level. Maintaining and reasoning over such a composition would also be a complex task, not to mention that there is no way to ensure that the final outcome is actually a dissemination model. Hence, in this context, defining the desired dissemination model would not only be complex but it would require programming skills to specify the mashup, regardless how fancy the user interface might be.

Thus, domain concepts and processes (e.g., notions of publication, review, paper, venue,…) should be exploited in order to really assist her in the composition. Taking this as reference scenario, we discuss the limitations and required extensions to current mashup platforms, in the following sections.

3 Understanding Current Mashups and their Limitations

As introduced in Section 1, the mashup approach should provide domain-experts with no programming skills with suitable mechanisms allowing them to develop autonomously their situational applications, leading to the above mentioned advantages in terms of responsiveness and effectiveness of solutions.

What makes possible moving application development from IT-experts to domain-experts, is probably the complete separation of roles, and thus of required skills, among component and composition developer, as depicted in Figure 1.

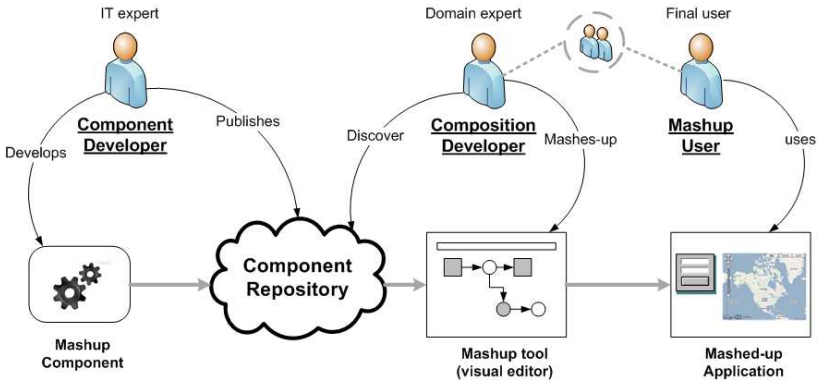

Figure. 1. Separation of roles among component and composition developer. Dotted representation among composition developer and mashup user, indicates that both roles can be covered by the same person, as typically happens in the context of situational applications. The figure is an adaptation of the one presented in [15].

The former is responsible to create and publish the building blocks that will be glued together to realize the final composite application. These components will implement or wrap some services or will represent a user interface. Complexity is primarily pushed into components, leaving compositions simpler and lightweight. It is clear that component development requires specific programming skills, in particular

in the web-programming field, since mashup platforms are typically offered as web applications, following the Software as a Service (SaaS) approach [8]. To this end, a number of web tools have been proposed to help and simplify the creation of services and components (e.g., data extraction from web pages, web clipping) [9]. Some examples are OpenKapow and Dapper. Both this tools provide simple mechanisms to grab contents from web pages and expose extracted data as web services or RSS feeds. However, this kind of tools still requires a not negligible knowledge of programming concepts to be effectively used.

Assuming that components have been developed and made available, composition developers, now, only need to define the business logic addressing their needs, connecting available components, usually through simple visual mechanism (e.g., drag and drop). This operation should not need any particular programming knowledge or complex operation and should be tackled by advanced web users that have a deep knowledge in their domain but no skills in programming. Typically, in the context of situational applications development, the domain-expert plays both the mashup developer and mashup consumer role (as indicated by dotted representation in Figure 1), since she develops compositions to automate processes covering her situational needs.

Providing autonomy in mashups composition to advanced web users is the big claim of most mashup platforms, but our experience and studies showed that it is not actually fulfilled in general. Proposed solutions allow domain-experts to compose their applications without the need to write programming code, but this does not mean that the composition process is easily and intuitively affordable by non-programmers [10]. In the example of Section 2, this would the case for the scientist trying to select publication venues but that find herself with a web service connector. Analyzing available mashup tools, we concluded that they are often confusing for domain-experts, starting from the components selection that, given the vast amount of available possibilities, could be time-consuming and error prone. For example, if we analyze two popular visual environments for "Consumer mashup" composition, Yahoo! Pipes and Microsoft PopFly[2], we can see that they provide users with about 50 components for Yahoo! solution and more than 300 for Microsoft one. Moreover, when the needs require more complex mashup solutions many tools either are no more sufficient or start requiring to the composition developer (domain expert) a deeper and deeper understanding of programming concepts. In fact, a significant part of offered components provide functionalities that can be exploited only by those users that have good programming knowledge (e.g., regular expressions, loops). Another important lack of currently available solutions is that almost none of them provide *universal integration*, that is, as discussed in [7], the seamless integration of data, application, and user interface (UI) components, characteristic that we consider necessary to actually enable end-users to develop their situational applications. For instance, Yahoo! Pipes is mainly oriented toward data integration while Intel MashMaker mainly focuses on UI integration, but to build real-life complex applications both ingredients are needed. In the field of the "Enterprise mashup" tools many efforts is being done to address some business-critical issues, like security,

[2] Microsoft PopFly was discontinued on August 2009, but still remain an important mashup platform example that attracted thousands of developers.

privacy, reliability and accountability. From the point of view of domain-experts usability, enterprise solutions suffer of the same problems of consumer ones. In particular, although there exist powerful and complete mashup solutions, they are usually targeted at programming-skilled users. A noticeable example is the Tibco[3] suite, providing users with a vast amount of different components and mechanisms, covering every possible need, but often strictly related to programming concept that domain-experts could completely ignore or, however, difficultly manage.

All the generic components and mechanisms that available mashup solutions, both consumer and enterprise, provide and their programming-nature limit the possibilities of composition of domain-experts to "toy applications".

We argue that the main reasons for these limitations regarding most of the available tools need to be searched in their aim to be generic-application building tools. This general purpose attitude make it difficult for domain-experts to get familiar with components, functionalities and mechanisms representing concepts and entities they are not acquainted - and which they are not interested in. Moreover, such an approach aims at integrating components from different sources belonging to different domains, so, very often, making possible the communication among different components is a complex task still requiring specific programming efforts. Back to our example of Section 2, this would be the case for the scientist who wants to aggregate, according to her, valid venues of scientific resources (e.g., publishers, blogs, eprints) to incorporate them in her model. This would lead to complex mappings requiring, most probably, some programming skills.

Overcoming these issues requires a different mashup platform architecture allowing domain experts to work in their natural playground, where they are familiar with concepts and issues, so that they can tackle the development of their situational applications. To the best of our knowledge, there is no related work actually exploring the concept of domain-specific mashup. Next section will describe our proposed solution and architecture, aiming to enable real-life application development for domain-experts.

4 Domain-specific Mashups

Domain-specific mashups is our proposal to exploit domain concepts at the mashup composition level in order to put domain-experts at the center by providing an environment that can really assist them in the composition of domain-specific mashups. So, we need specific solutions pushing domain concepts up to the composition editor level, so that users can play in their well-known conceptual environment. To achieve such a system, we propose a modular architecture including two main layers, as depicted in Figure 2.

[3] http://www.tibco.com

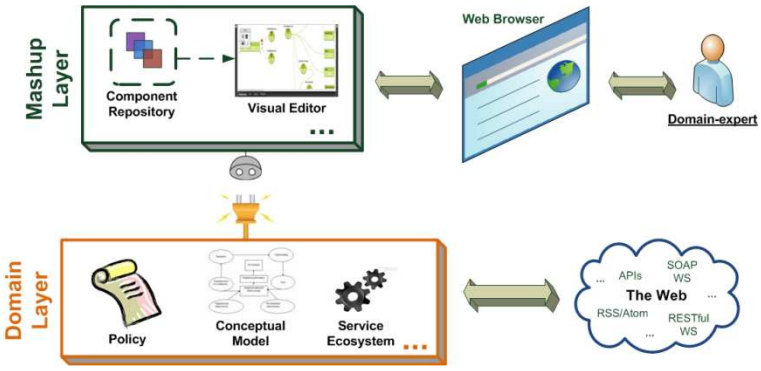

Figure 2. High-level architecture of a domain-specific mashup platform.

The upper layer is the actual *Mashup Layer*, that is, a mashup tool similar to some available today. In particular, this layer should - at least - include a composition visual editor, providing end-user friendly mechanisms for composition development through a common web browser, and a component repository, providing all the components that could be useful for building compositions in a given domain. In addition, other parts should be included at this level, like a runtime environment able to run the produced composition and other implementation-specific components, but those go beyond the scope of this work that is trying to focus on the composition-development phase seen from the domain-experts point of view. Our proposal to actually help and enable domain-experts to autonomously create their composite applications is to transform our generic mashup tool into a domain-specific one injecting domain related concepts into the development environment. Aiming at this, we create a *Domain Layer* that will be then plugged into the *Mashup Layer*. This lower layer is responsible for the domain characterization. In other words, it will define all the concepts and entities proper of the domain, their representation and general rules regulating the interactions among them. Furthermore, this layer will provide the domain related ecosystem of services, either implemented inside the layer or wrapping web-sourced services.

In this section we provide the two aspects covered by our proposal: modeling and characterizing the domain and its projection to the mashup platform.

4.1 Characterizing the Domain: Domain Layer

In order to leverage domain-experts knowledge in the composition, we need to understand the concepts, properties, rules and processes that make the domain. The definition of these elements is key to the selection of the right level of abstraction for users. To this end, we rely on the definition of the conceptual model, the business level operations and the domain rules.

Conceptual model. In the context of a particular domain, there are concepts and relations among these concepts that are known in the domain and familiar to domain experts. These concepts are commonly represented in a conceptual model. For

instance, in Figure 3, we show a possible conceptual model for the example introduced in Section 2.

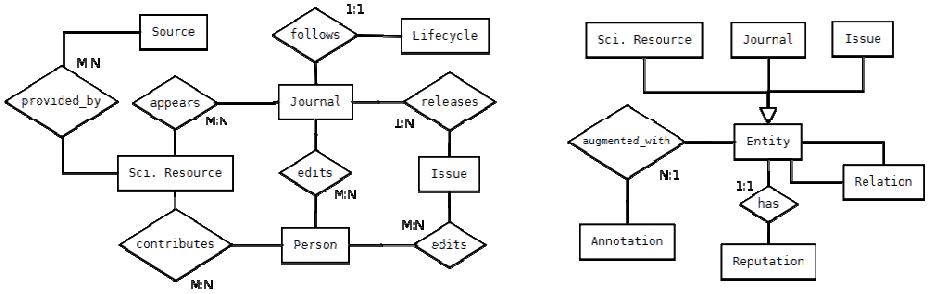

Figure 3. Liquid journals conceptual model

The Figure above captures the concepts in the knowledge dissemination domain. It is based on the liquid journal model, which represents a family of models from the traditional ones to the ones more social and web-aware [11]. In this model we can see that *journal* is a first-class entity composed of *scientific resources* (papers, blogs, datasets, ...), organized in *journal issues*, driven by *editors* and that follows a certain lifecycle. We can also see that *scientific resources* belong to certain sources (venues).

Business-level operations. Operations and processes that affect the shared concepts are highly relevant. These relate to users' every-day life and as such provide the level at which the expert can better reason. Following our example, these are the operations which are meaningful in the domain such as *publish*, *evaluate*, *review*, *submit*, and other more social such as *share*, *annotate*, *search*, etc.

Business rules. Business rules are well known by domain experts. They are very important as they give shape to the business logic and processes. In our example, we could establish as a business rule that whatever publication model we follow, we need to first select/review a paper before publishing it in journal.

The information we have described above is present in the domain but not exploited in the mashup composition. Mashup composition environments strongly rely on the domain expert to build application from usually low-level components, and so they do little or nothing to assist users. To inject these elements into the composition environment we propose to build a *Domain Layer* as a way of hiding the unnecessary complexity which is currently exposed to users. Although we are convinced that the complexity could be pushed into the component design, having such a platform will make component development much easier and would ensure consistency and, finally, smooth composition. Another good reason for having such a layer is the increasing existence of domain specific ecosystems. Thus, in practical terms, concepts and operations are captured typically by a platform exposing an API. In our example, the *liquid journal* platform provides the services and key entities via

RESTful services[4]. This platform builds on existing sources of information (e.g., Google Scholar, eprints, DBLP) and allows upper layer to access them using the common conceptual model in Figure 3. Solving the heterogeneity at this level has the advantage of not only providing homogeneous programmatic access, but also of avoiding taking complex mappings to the mashup environment by preparing the components according to the shared concepts. Of course, taking these and all the domain elements described here requires some work to make it happen. We discuss these and other related issues in the next subsection.

4.2 Taking the Domain into the Mashup Platform

Injecting the domain information provided by the ecosystem into the composition environment requires some work on both the component development and the mashup platform. More precisely, it requires (i) a proper design of the components in what regards the level of abstraction and composition, (ii) making the composition environment aware (to some extent) of the business rules by defining some composition rules, and thus taking to the mashup environment what is already on the backend, and (iii) making the environment aware of the coupling among the components (and concepts managed by the components) in order to assist users in the component selection.

In what follows we describe our approach using mashArt as reference platform, albeit the ideas described here could be applied to other mashup platforms.

Building domain components. Good quality components are key to the success of any mashup platform and derived applications, and so is the case for domain-specific components. Thus, in addition to known practices for component development (e.g., [12]), building domain components puts some extra usability requirements. To reach users, we must select the right level of abstraction for the components and composition. It should be the one at which users find in the environment only concepts they know, expressed with the same terminology, i.e., components should be meaningful to the domain expert and be related to the business-level *concepts and operations*. In practice, we pass from components that represent just technology (e.g., a component connecting to a service) to components that have a precise semantic that is familiar to the domain expert (e.g., the component that publishes a paper). This is a conceptual shift.

Components should also be designed for smooth composition. Composing components should be straightforward to domain experts and complex mappings avoided to the possible extent. To this end, components events and operations I/O should be presented in terms of the domain concepts. In practical terms, this means that a domain-specific *namespace* should be made available to the component definition. Additionally, to ease and, at the same time, check the component development process, the platform could possibly provide a domain-specific component editor able to guide developers in the generation of new components

[4]http://docs.google.com/Doc?docid=0ARoLwpXLTjBGZGt6Mng0cl8yZG00N3Y4Y24&hl=en

(particularly for their descriptors), based on the knowledge coming from the *Domain Layer* (e.g., available entities and their representations).

In Listing 1, we illustrate the definition of a component designed taking into account a domain-level operation for providing familiar functionality, and domain-concepts in the I/O to ease the composition.

```
<?xml version="1.0" encoding="utf-8" ?>
<mdl version="0.1" xmlns:lj=" http://liquidjournal.org/schema/liquidjournal.xsd">
 <component name="Publish" binding="component/UI" stateful="yes"
  url="http://mashart.org/registry/X/Publish/">
  <event name="Paper published" ref="onPublish">
    <output name="Published Entity" type="lj:entity"></output>
  </event>
  <operation name="Publish paper" ref="doPublish">
    <input name="Entity" type="lj:entity"></input>
  </operation>
 </component>
</mdl>
```

Listing 1. MDL of the component Publish

As seen in the *Publish* component I/O refers to the type *entity*, an abstraction introduced in the conceptual model. A strong point of this approach is that it does not introduce any change into the MDL (mashArt Description Language, used to define each component of the mashArt platform) but it introduces higher level types based on the conceptual model. In Listing 2 we show part of the definition of the XSD used to make the mashup platform aware of the conceptual model[5].

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema ... xmlns:tns="http://liquidjournal.org/schema/liquidjournal.xsd">
...
 <xs:complexType name="entity">
  <xs:choice>
   <xs:element ref="tns:liquidjournal" />
   <xs:element ref="tns:issue" />
   <xs:element ref="tns:sciResource" />
  </xs:choice>
 </xs:complexType>
 ...
</xs:schema>
```

Listing 2. XSD Definition of domain-level concepts

Business-level composition policies. As mentioned before, business rules are important and usually enforced at the backend. In order to ensure some patterns in the output and help users in the definition of consistent mashups we believe it is important to abstract these rules and take them to the level of composition. Of course, domain rules can be very complex to be completely pushed to the composition, so we target composition policies as a tool for "assistance" rather than "enforcement". There

[5] The complete XSD can be found here https://dev.liquidpub.org/svn/liquidpub/prototype/ljdemo/server/resources/meta/liquidjournal.xsd

could be different ways of defining such policies, but in this paper we consider syntactic constraints based on category of components as simple starting strategy (that will be then extended in future). Components providing some common functionality could be categorized and category-level policies defined on them. Thus, policies allowing/denying cross-category couplings could be defined. Taking our scenario as an example, we could define categories such as *review processes*, *selection modes*, *publication modes*, *sources and venues*, *people*, *metrics* and *entities*, and on these define policies such as publication cannot be performed before the selection. Note that categorization is not mandatory and so "free" components not regulated by the policies are perfectly allowed. Implementation-wise, policies can be defined using XML and the mashup editor can check and guide the user during the composition based on the rules regulating that domain.

Mashup composition environment. The information about the domain components and policies should finally be reflected on the mashup composition environment. Domain components provide the opportunity to make meaningful suggestions based on components coupling (I/O matching) and so introduce a basic yet useful proactive behavior in the component selection. In its simplest, we could see composition as a domino where the domain-expert select components among the compatible ones. Domain policies provide even richer information. Policies will have higher priority over coupling based suggestions, filtering out and ranking eligible components. In addition to this, the mashup composition environment should provide intuitive UI representation for components and the connections (e.g., meaningful icons for components) to ease the selection.

Finally, having composition information at this level will enable further improvements in the composition environment. It would make it easier to extract usage information that could be used to improve the selection process (by reusing past experiences), since all components are defined at business level, making possible to extract the semantics of the compositions and usage.

5 Conclusion

In this paper we have introduced domain-specific mashups as a way to inject domain knowledge into the mashup composition, with the ultimate goal of providing domain-experts with the tools to compose mashup applications from familiar domain concepts. Our approach rather than proposing a technological change, proposes a paradigm shift in going from generic platforms with mainly low level (and technological-oriented) components to domain-specific vertical extensions. To this end, we have introduced a layered architecture in which we distinguish the mashup layer from a domain layer that can be plugged in. The definition of this domain layer is what allows us to describe the level of abstraction familiar to the user. In addition, the separation among the two layers allows the same mashup tool to be reused for different domain verticals, simply replacing the underlying domain layer.

As immediate future work we need to investigate more on how to define and model the domain characteristics (entities and rules) such that they are at the same

time useful to help the composition phase but also not too rigid, guaranteeing the needed flexibility. Then, we plan to go from the conceptual modeling to the actual implementation of the extensions to the mashup environment. In particular, we plan to do that working on the mashArt platform and on the domain of scientific publishing, as introduced in this paper.

6 References

[1] A. Jhingran. Enterprise information mashups: integrating information, simply. In VLDB '06, pages 3–4. VLDB Endowment, 2006.
[2] I. Floyd, M. Jones, D. Rathi, M. Twidale. Web Mash-ups and Patchwork Prototyping: User-driven technological innovation with Web 2.0 and Open Source Software. Proc. Of HICCS '07
[3] S. Bitzer, M. Schumann. Mashups: An Approach to Overcoming the Business/IT Gap in Service-Oriented Architectures. Proc. of AMCIS 2009.
[4] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. Proc. of the SIGCHI '07, pag. 1435-1444, 2007.
[5] Yahoo! Pipes project. [Online] http://pipes.yahoo.com/.
[6] Intel MashMaker project. [Online] http://mashmaker.intel.com/.
[7] F. Daniel, F. Casati, B. Benatallah, M. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. Proc. of ER'09, Pages 428-443.
[8] M. Shan. Software as a Service(SaaS) The challenges of application service hosting, Proc. of Int. Conference on Web Engineering, Como, Italy, July 2007.
[9] F. Daniel, S. Soi, F. Casati. Search Computing - Challenges and Directions, edited by S. Ceri and M. Brambilla, LNCS, Volume 5950, March 2010, Springer, Pages 72-93.
[10] R. Tuchinda, P. Szekely, C. A. Knoblock. Building Mashups by example. In Proc. of the 13th international Conference on intelligent User interfaces IUI '08, p. 139-148
[11] M. Baez, F. Casati, A. Birukou, M. Marchese. Liquid journals: Knowledge dissemination in the Web Era. http://eprints.biblio.unitn.it/archive/00001814/
[12] Daniel, F., Matera, M.: Turning Web applications into mashup components: issues, models and solutions. Proc. of ICWE'2009.