A Domain-Specific Language for Web APIs and Services Mashups

E. Michael Maximilien[1], Hernan Wilkinson[2], Nirmit Desai[3], and Stefan Tai[1]

[1] IBM Research
{maxim, stai}@us.ibm.com
[2] Universidad de Buenos Aires
hernan.wilkinson@gmail.com
[3] N.C. State University
nvdesai@ncsu.edu

Abstract. Distributed programming has shifted from private networks to the public Internet and from using private and controlled services to increasingly using publicly available heterogeneous Web services (e.g., REST, SOAP, RSS, and Atom). This move enables the creation of innovative end-user-oriented composed services with user interfaces. These services *mashups* are typically point solutions to specific (specialized) problems; however, what is missing is a programming model that facilitates and accelerates creation and deployment of mashups of *diverse* services. In this paper we describe a domain-specific language that unifies the most common service models and facilitates service composition and integration into end-user-oriented Web applications. We demonstrate our approach with an implementation that leverages the Ruby on Rails framework.

1 Introduction

There are two paradigm shifts occurring on the Web that are changing the way software is developed. The first is the increasing availability of Web APIs (or Web services) in the form of Representational State Transfer (REST) [2] and SOAP services, as well as RSS and Atom data services. These Web APIs enable external partners (or software agents) to incorporate business data and processes of the service providers into their own Web application or Web client. Indeed, the proliferation of these Web APIs have resulted in various composed services with UIs, or *mashups*, which provide solutions to very specific and narrow problems. An example is Podbop.org, which combines the API and data retrieved from Eventful.com with MySpace.com, as well as other MP3 databases, to create a site for music lovers who want to sample music of new (unknown) artists performing in local bars and clubs.

The second paradigm shift is a movement to increasingly program Web applications using dynamic programming languages and frameworks, e.g., JavaScript with AJAX, Ruby with Ruby on Rails (RoR), Python with Zope, Smalltalk with Seaside, as well as PHP. These languages allow for rapid application development and testing; and not only afford programmers expressive and powerful

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 13–26, 2007.

frameworks, but they also lead to the use of high-level abstractions which are more representative of the domain in question.

In many ways these two paradigm shifts are complementary since they essentially help realize the vision of a *programmable Web*. However, frameworks focused directly on facilitating the creation and deployment of mashups of diverse Web APIs and services are missing. For instance, each type of service (REST, SOAP, RSS, and Atom) has heterogeneous means of exposing the service interface or none at all. Additionally, there is a need to help address common distributed systems issues that arise [3].

In this paper we present a domain specific language (DSL) for services mashups that alleviates some of these issues. In particular, our DSL (1) allows for a common interface representation among diverse service types, (2) facilitates exposing asynchronous and synchronous method invocations, (3) gives a uniform model for service data and service operations' interactions, and (4) enables basic service data caching. We demonstrate an implementation of our language using Ruby and the RoR framework.

1.1 Organization

The rest of this paper is organized as follows. Section 2 gives an overview of our platform architecture. We also provide a more thorough definition of services mashups. Section 3 gives a more precise definition of our language and some brief examples of the language in action. Section 4 describes our implementation. Finally, Section 5 follows with a discussion of our approach which includes related works and limitations.

2 Background and Architecture

In order to demonstrate our approach to mashups and how a DSL can facilitate mashup creation, it's useful to first have a more precise definition of mashups along with possible implementation approaches. We then illustrate our architecture with a brief overview of the base platform that we use.

2.1 What Are Mashups?

At its core, a mashup is a Web application that aggregates multiple services to achieve a new purpose. Conceptually, mashups are new Web applications used for repurposing existing Web resources and services. They include all three aspects of a typical Web application (model-view-controller) with additional functionality. For us, a mashup includes three primary components:

1. *Data mediation* involves converting, transforming, and combining the data elements from one or multiple services to meet the needs of the operations of another. For instance, mediating between data models of tags represented in both the Flickr [1] and the Eventful's APIs.

[1] http://api.flickr.com

2. *Process (or protocol) mediation* is essentially choreographing between the different services to create a new process. For instance, process mediation includes invoking the various service methods, waiting for asynchronous messages, and sending any necessary confirmation messages.
3. *User interface customization* is used to elicit user information as well as to display intermittent and final process information to the user. Depending on the domain, the user interface customization can be as simple as an HTML page, a more complex series of input forms, or an interactive AJAX UI.

2.2 Mashup Implementation Approaches

Two example technologies used to build mashups are the Google Web Toolkit (GWT) [2] and plain RoR. As an example, consider implementing a mashup that updates personal calendars from Atom feeds, and allows adding rating information for events. Assume, without loss of generality, that each user's calendar can be accessed via a REST API using a key parameter to authenticate users. The Atom feeds generate heterogeneous event entries for each up-coming talks. And finally, each user has an account in Eventful, which exposes a common data model for events and REST APIs to add, search, rate, and retrieve events.

Using GWT, two main components of the mashups are to represent the various talk feeds entries and converting them to the uniform *Event* data model of Eventful. This involves data mediation between the model for a *Talk* entry from the Atom feed to an the *Event* model in Eventful. For instance, the former may have a location data as a string which needs to be parsed into the different fields for location represented by the latter.

GWT does not have built-in libraries for accessing REST or Atom services. This means that for each service type, there is a need to find an appropriate Java ™library or creating one manually and binding and testing to the services in question.

Next, we need to mediate the protocols of the three services to achieve the goals of our mashup. For instance, assume that the first page of our mashup simply displays all up-coming events in the next two-weeks that are not already added to the user's calendar. One possible choreography between the three services to achieve this goal is:

1. Retrieve entries from all feeds for up-coming talks. Additional consideration for this step are: caching public entries for subsequent access or for other users and enabling asynchronous updates of the cache.
2. Query the user's calendar to get all talk entries for the next two weeks.
3. Mediate between the user's calendar entries and the feed entries. Decide on comparison criteria, e.g., time, date, location, and so on.
4. For each talk not present in the user's calendar, create a common representation of these talks as events in Eventful and add to the Eventful database via REST API. Eventful events include a model for speakers which also needs to

[2] http://code.google.com/webtoolkit

be mediated from the data feeds. If the talk is already present, then retrieve it and mediate between the reconciled event in previous step and this one.
5. Present a formatted page to the user with each new event with checkboxes and a button to enable the user to add events to her calendar.
6. Allow user to view events in her calendar. For each event: (1) display event data; (2) allow user to delete the event; and (3) allow user to indicate attendance.

It's worth noting that using GWT to implement the choreography above results in adding custom code for the data mediation steps, for resolving the choreography, as well as for any data caching. Additionally, there is no reuse of the various steps across mashups of the same services.

Using RoR is effectively similar to using GWT, though simpler for some aspects. For instance, RoR's built-in support for databases via *ActiveRecord* would facilitate caching the feed entries into a relational database. However, this would need to be manually done for each type of *Talk* feed added to the system.

2.3 Ruby on Rails

The Ruby on Rails (RoR) framework enables agile development of Web applications. The framework contains primitives to help efficiently implement all aspects of an Model-View-Controller (MVC) Web application. Each MVC Web application contains: (1) *Model* classes representing the data elements of the application's domain. Model objects can persist their state in a database using a series of conventions; (2) *View*s are the dynamic pages displayed to the user of the Web application. Each view file contains HTML and embedded Ruby code which is translated into JavaScript, HTML, and CSS on the server before being sent to the client (i.e., browser); and (3) *Controller*s constitute the middle layer between models and views. Controllers are classes whose names and methods map to the application URL path. Controller methods contain business logic by operating on model objects and accessing remote services.

Additionally, RoR also includes basic facilities to allow controllers to invoke external SOAP Web services and to access remote Web resources. However, the RoR Web API support lacks some key features needed to streamline and create mashups, e.g., lack of consistent and uniform representation for all different types of services, lack of support for asynchronous invocation of services' operations that can work across all service types, and lack of provisions for easily manipulating complex XML data (beyond parsing).

2.4 Architecture Overview

To address the above deficiencies (and others) as well as to provide a uniform model for building and sharing services mashup we created the *Swashup* platform. Our architecture extends the RoR architecture with a new DSL, supporting libraries, as well as associated platform models and services. Figure 1 illustrates the high-level components of our architecture.

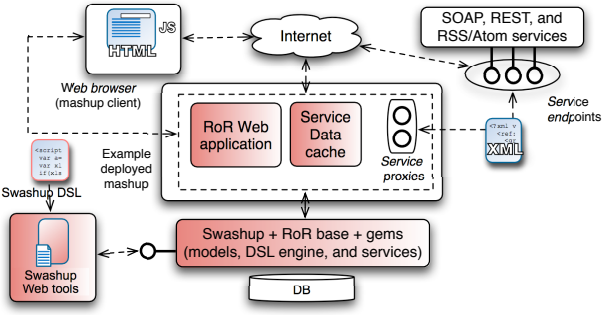

Fig. 1. Swashup high-level architecture

Using the Swashup Web UI tools, an end user creates, edits, and deploys a Swashup project which contains the necessary information for describing the services to be mashed up as well as the mashup information. Using the Swashup platform services, the Swashup project is deployed as a complete RoR Web application with all necessary service proxies, models, and initial views for each mashup.

3 Swashup DSL

We now introduce our language and discuss the main requirements for any DSL as well as some criteria for judging their value.

3.1 What Are DSLs?

A domain-specific language (DSL) is a 'mini' language built on top of a hosting language that provides a common syntax and semantics to represent concepts and behaviors in a particular domain. In general, using or designing a DSL helps achieve the following goals: (1) **Abstraction** by enabling programming at a level higher than what is available with the host programming language constructs or its libraries. A DSL allows the domain concepts, actions, and behaviors to be represented directly in the new syntax; (2) **Terse Code** as a side effect of programming in a higher-level of abstraction; (3) **Simple and Natural Syntax**, which leads to easy to write and read code; (4) **Ease of Programming**, which is desirable of any programming language and also somewhat difficult to judge. However, since a DSL enables the expression of constructs that map directly to a domain, it generally makes programming easier (for applications in the domain) than using the underlying language directly; and (5) **Code Generation** is how a DSL primarily functions. Essentially, the DSL statements are translated at runtime into code that uses the underlying language and its libraries. This can be either using metaprogramming techniques or by code generation of program files.

3.2 Language Overview

In the Swashup DSL, we directly represent in the syntax, the concepts necessary to cover the three main components of our conceptual model for mashups: (1) data and mediation; (2) service APIs, their protocols, and choreography; and (3) a means to generate Web applications with customized UIs for the resulting mashups. The following concepts form the main types of statements in our language.

- **data** describes a `data` element used in a service. A `data` element corresponds to an XML schema complex type. Each `data` element has a name and a series of `member` attributes. These attributes' types can be either regular XSD simple types or other data elements. Section 4 gives more details on our XML mapping approach including conventions and rules.
- **api** gives a complete description of a service's interface. This includes descriptions for the service's API, including operation names, parameters, and data types. An operation data type is either a simple type (e.g., string or integer) or refers to a `data` element. Section 4.1 discusses the conventions for creating `api` definitions for SOAP and REST services, as well as Atom and RSS services.
- **mediation** describes the transformation of one or multiple data elements to create a new one. Essentially, a `mediation` is a mapping between `data` elements with some possible transformations.
- **service** binds a service `api` with a concrete service. Part of the binding is to indicate the service's type (e.g., SOAP, REST, RSS, or Atom), the service's endpoint, as well as give an alias for the service instance.
- **recipe** constitutes a collection of `service`s and `mashup`s. A recipe also includes views for each of the mashup `wiring`. Some views are automatically generated and others are customized by the user.
  - **mashup** is a composition of one or multiple services. It comprises a collection of `wiring` declarations. Each `mashup` translates into a composed service which may be exposed externally and used for further mashups.
  - **mediate** invokes a `mediation` declaration with instances of the data elements to mediate. The result of a `mediate` call is a primitive type instance or another data element instance.
  - **wiring** which comprises two levels of granularities of connecting the services that are part of a `mashup`: (1) **:protocol** is a top-level structure of a mashup. It represents one or multiple operation `wiring`s and `step`s invocations. It also associates with views as specified in Section 3.3 and (2) **:operation** is the wiring of one or multiple services' operations. Operation wiring includes the ability to invoke services' operations in an asynchronous fashion by automatically setting up callbacks.
  - **step** constitutes one atomic step in a protocol mediation. A step can be invoked multiple times as part of a protocol wiring. A `step` is invoked by the `step`'s name as a method call.

– **tag** and **tags** allows users to annotate terms to the various components of a Swashup `recipe` as well as `data` and `api` definitions. These types of tagging allows for some level of comments and idiosyncratic semantics to the various components.

For brevity, a complete formalization of our language in BNF (Backus-Naur form) is not described in this paper.

3.3 Conventions

Following one of RoR's main philosophy, namely, *using conventions over configurations* [3] our Swashup DSL includes a series of conventions. The use of conventions is meant to simplify the language's usage and to make the resulting code more compact.

– *Naming* are added to most statements as the first parameter and as a Ruby symbol or string. Names use either a camel-case (e.g., :SomeDataElement) format or lower-case (e.g., :some_mashup) using underscore to separate words. The `data` and `api` require camel-case. Other language constructs accept either camel-case or lower-case with underscore, e.g., `wiring` constructs.
– *Variables* are always lower-case with underscore separating the words in the variable's name.
– *Recipes* when deployed are complete RoR Web applications with controllers matching each of the `mashup`.
– *Mashups* are converted to a RoR Web application controller and every protocol `wiring` translates into an action for the Web application. This allows the application to be exposed as a service as well as adding views.
– *Views* by RoR convention associate with a controller's action and therefore with a protocol `wiring`. Using an `async` parameter to operation `wiring` allows the views to be created with AJAX JavaScript that can check back with the controller for updated data and refresh the view's content.

3.4 Examples

To illustrate the power of our DSL we now give a complete example. Briefly, our example mashes the data and protocol of two available services: (1) Google's SOAP search Web service and (2) Yahoo! Flickr's photo REST API. The main purpose of our mashup is to allow users to search for a phrase or word using the Google search service and display the top results. Additionally, we display the top thumb nail photos associated with the searched words from Flickr by matching the tags that the Flickr community has used for the shared photos.

Our mashup's `recipe` is divided into four listings (Listings 1.1 to 1.4), each illustrating one aspect of the solution. Listing 1.1 shows how we use the DSL's `data` construct to represent the data coming from Flickr (starting line 1). The API definition starts at line 5.

[3] http://www.rubyonrails.org/

Listing 1.1. Example Swashup `data` and `api` definitions

```
data :Photo do
    member :url, :xml_text
    member :tags, [:string], :xml_text
end
api :FlickrApi do
  api_method :find_photos,
             :expects => [{:tags => [:string]}],
             :returns => [[:Photo]]
end
```

Listing 1.2 shows the start of our Google search SOAP API and Flickr REST API mashup `recipe`. We start by tagging the `recipe` in lines 11. Next we use the `service` construct to create a binding to the Flickr REST service, giving it an alias name of `f` and we would include similarly for all other services used.

The `service` construct unifies the different types of services supported in our DSL. It includes type specific parameters, e.g., `:wsdl` for a SOAP service, and type independent parameters, for instance, the `:endpoint` which is used for SOAP and REST services and used to indicate the RSS or Atom feed URL. The `service`'s `:api` parameter points to the defined API (Listing 1.1) for SOAP or REST services and is implicit for RSS and Atom feeds (see Section 4.1). However, RSS and Atom feeds require a `:entry` parameter to indicate the `data` definition for the expected entries of of the data feed.

Listing 1.2. Example Swashup `recipe` showing `tag`(s) and `service`(s) definitions

```
recipe :GoogleFlikcrRecipe do
  tag 'recipe',
      :synonyms => ['example', 'exemplar', 'pattern']
  service :flickr_service, :alias => :f
          :type => :rest,
          :api => :FlickrApi,
          :endpoint => 'http://rest.flickr.com/api'
    # service for Google search service
    # constants declarations, other service definitions ,$\ldots$
end
```

Next, we illustrate how to define `mediator`s, `wiring`s, and `step`s. These are shown in Listing 1.3. Our `extract_tags` mediator starts in Line 20 and takes a string input and divides it into a set of keywords by first filtering them.

Each `wiring` is converted into a method that can be called in the context of the `recipe`, e.g., *search 'flickr mashups'*, however, the value added for the creating wiring (besides the design values and potential for reuse) is the ability to automatically make the wiring invoke operations in an asynchronous fashion.

This is achieved by either passing a Ruby block that is called back with the result of the `wiring` when the operation completes or by passing a block or Ruby method taking one parameter using the automatically generated setter method named `search_callback=`. The result of last invocation of a `wiring` is also automatically added to an instance variable by the wiring name.

Listing 1.3. Example Swashup `recipe` and `mashup`

```
recipe :GoogleFlikcrRecipe do
19      # tag(s), tags, service(s), and CONSTANT(s)
        mediator(:extract_tags, :data) do |string|
21          keywords = []
        string.split.each do |s|
23          keywords << s unless NON_KEYWORDS.include?(s)
            end
25          return keywords
        end
27      wiring(:find_images, :operation, :async) do |words|
            @urls = []
29          words.each do |w|
                url = f.find_photo(w).url
31              @urls << url unless urls.include?(url)
            end
33          return @urls
        end
35    step :search_and_images do |string|
            @results = search(string)
37          @keywords = extract_tags(string)
            @urls = find_images(keywords)
39    end
        # other mashup(s)
41  end
    end
```

Listing 1.4 completes our example `recipe`. It illustrates how different `mashup`s are added to a `recipe` by adding different protocol `wiring`s. Each protocol `wiring` can accept parameters as a Ruby block parameters and can make calls to `step`s, `mediator`s, and operation `wiring`s. Importantly, each `mashup` can have it's protocol `wiring`s exposed as SOAP, REST, RSS, or Atom services. This is achieved using the `expose_operation` construct. For RSS and Atom services the protocol `expose_operation` uses an `:entry` parameter which binds to a `data` indicating the format of the RSS or Atom entry and instance variable that will contain the updated entry data.

Listing 1.4. Example Swashup `recipe` and `mashup`

```
recipe :GoogleFlikcrRecipe do
43     # tag, tags, service, and any CONSTANT(s)
       # mediator(s), wiring(s), and step(s)
45   mashup :spell_search_images_mashup do |g, f|
       tags ['mashup', 'spell']
47         wiring(:images_for_keywords, :protocol) do |words|
               expose_operation :soap,
49                              :expects => [{:keywords => :string}]
                                :returns => [[:Photo]]
51             find_images(words)
           end
53     wiring(:search_and_images, :protocol) do |string|
               expose_operation :atom,
55                      :entry_data => :GoogleSeearchResult,
                        :entries => @results,
57                      :atom_metadata => [{:author => 'Jane Doe'}]
               spelled = spell_search(string)
59         search_and_images(spelled)
           end
61   end # mashup
  end # recipe
63 end
```

3.5 Value of DSL

As mentioned in Section 3.1 our DSL enables mashup programming at a higher-level of abstraction than frameworks supporting Web application programming. This is primarily achieved by defining high-level constructs that facilitate mashup creations. Specifically:

1. *Uniform treatment of diverse services (REST, SOAP, RSS, and Atom).* This is especially useful for REST, RSS, and Atom services which do not have standard machine readable definitions (such as WSDL for SOAP services).
2. *Facilitate asynchronous operation calls.* For each wiring operation you can specify `:async` as an option which will add (via metaprogramming) all necessary code to call methods asynchronously and deal with callbacks and so on.
3. *Uniform treatment of service data elements.* This includes having a definition of the data elements passed and returned to the `service` constructs. Additionally, our `data` construct help: (1) *facilitate data mediation and reuse* and (2) *facilitate service data caching*

4. *Uniform design for mashups.* Using our language we give some structure to the design of service mashups while also enabling the full support of a modern language and framework for Web application development.
5. *Integrate into RoR.* First by using Ruby as the implementation language (which makes RoR integration seamless) but also in how to expose a `recipe` as a RoR Web application.

4 Implementation

Our Swashup platform is implemented completely in Ruby and RoR. We leverage the RoR framework by using and extending various aspects. Using Ruby's metaprogramming support and the rich view capabilities of the RoR platform every recipe is converted into a Web application that can be customized to create rich AJAX Web applications. In addition, every recipe's `mashup` can be exposed as a Web service (SOAP, REST, RSS, or Atom). This is achieved using the DSL constructs and a series of conventions.

Our metaprogramming approach is enabled using a series of class and object templates for the different constructs of our DSL. For instance, each `data` construct is translated into three classes: (1) a ROXML [4] class to enable parsing and generation of XML; (2) an *ActiveRecord* class to allow the data element to be cached in a relational database; and (3) a Ruby class that seamlessly aggregates the other two classes' functionalities.

For each `recipe` we generate a full RoR Web application with a controller class for each `mashup`. Each `api` construct translates into a RoR *ActionWebService* API classes that make use of the `data` classes. We extend the RoR classes to deal with REST and other types of services. Each `service` construct translates into an object that proxies the service it binds. The proxy exposes the `api` interface and is adjusted for each type of service supported.

The `mediator` and operation `wiring` translate into Ruby methods that are added to a `module` created for each `recipe`. This module includes the Swashup platform modules and is included itself into the generated controller classes for each of the `mashup` constructs. Finally for each `mashup` we also generate an API class with `api_method` for each protocol `wiring` that includes an `expose_operation` construct call. This is how a mashup is exposed as a service.

For each protocol `wiring` we generate the following view related artifacts:

1. A partial view that includes an HTML form for the parameters of the protocol `wiring`. If the protocol `wiring` does not have parameters then no partial view is generated. Using Ruby and *ActiveRecord* conventions we use text fields for strings, numbers, and `data` fields marked `xml_attribute`; and we use an HTML form for fields that point to other `data` element using `xml_object`.
2. An RHTML template view with the name of the protocol `wiring` that includes the partial views and with some default text to indicate that this view associates with the action and needs to be customized.

[4] http://roxml.rubyforge.org

3. An action method in the generated `mashup` controller class that uses the data from the partial view (if any is present) to call the protocol `wiring` method and displays the view page.

4.1 Details

We achieve uniform `data` and `service` and `api` descriptions by extending the RoR platform and using a series of conventions when describing services. First, the service `data` are described by using the XML schema. For SOAP services this schema is part of the WSDL and for REST, RSS, and Atom it can be inferred, by the human designer, from service's documentation, or from example input and output messages. The representation of the `api` for a service depends on the service's type.

- SOAP services are expected to have an associated WSDL which makes the API definition somewhat automatic. Each SOAP `portType` maps to an `api` definition. Each `operation` in a `portType` maps to an `api_method` in the associated `api` and uses the input messages as `expects` parameters and output messages for the `returns` hash [5]. The input and output message's XSD types translate one-to-one to a `data` definition. The `service`'s `endpoint` parameter maps to the SOAP endpoint in the WSDL's `service` section.
- REST services require additional conventions, especially since REST services do not have associated standard description languages. Each REST service specifies its endpoint as the root URI that is common across all of its operations. For instance, we use `http://api.evdb.com` for the Eventful's API since all REST operations have this root URI in common. The `api_method` for a REST `api` definition can also take a third `:http_method` parameter to specify either if this operation should be an HTTP `:get` (default), `:post`, `:put`, or `:delete`.
  REST operations use a simple convention to convert the path into a Ruby method. For path names that do not contain the underscore character (i.e., '_') in the operation's path elements translate into a Ruby method that uses underscore to separate its sections (if any). For instance, the path 'search/-customer' translates into the operation named 'search_customer'. If the path contains the underscore character then it is assumed that the path section translates into two underscores when converting to a Ruby method. For instance, the path 'search_all/customers' translates into the Ruby method 'search__all_customers'.
- RSS and Atom services follow the same `api` so it never needs to be specified. Figure 2 shows the UML class diagram for Atom services showing the operations available for any Atom service.
  Since RSS and Atom services are feeds that contains recurring elements, the type of the element must be specified in the `service` construct. That type is specified as a `data` construct which uses its *ActiveRecord* part to enable caching of the feed's data entries.

[5] A Ruby hash is equivalent to maps or dictionaries in other languages.

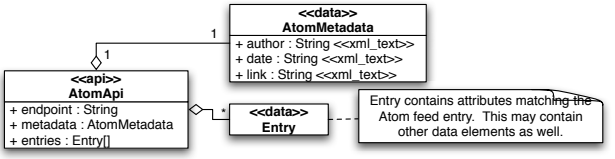

Fig. 2. Atom service API UML class diagram

5 Discussion

The Swashup DSL gives a high-level abstraction and language to create service mashups. Our initial implementation leverages and extends the RoR framework to create a set of tools that facilitate mashup creation as well as management.

5.1 Related Works

We divide related works into two main categories: mashup tools and frameworks and service compositions and service workflows.

Yahoo! Pipes [6] is an example of a mashup tool available on the Web. In Yahoo! Pipes, services (primarily RSS and Atom data services) can be 'piped' together (à la UNIX pipes) to create more complex data composition mashups. IBM's QEDWiki [7] is another example of a mashup tool. However, unlike Yahoo! Pipes, QEDWiki allows users to create widgets that access different services and data sources. Using the wiki metaphor, QEDWiki aims to make the composition process iterative and collaborative. While similar in objectives, both Pipes and QEDWiki differ from the Swashup platform, which focuses instead on giving common structures to mashups and creating a language to facilitate their creation and sharing.

Since Swashup, at some level, is essentially a platform for services composition, related works in services composition and workflows are important to note. BPEL is a workflow language adapted for services—instead of orchestrating a flow of activities, it orchestrates a flow of services [1]. Although BPEL has gained currency as a services composition solution, it is not geared toward UI-ready situational applications and mashups.

5.2 Directions

While with the current Swashup DSL we are able to create `recipe`s which encompass different types of services and somewhat complex data and protocol mediation, there is a need to test our language and platform with even more complex services and mediations. For instance, the types of mediations necessary for back-end enterprise integration. For examples, services such as the ones available in the SalesForce.com's AppExchange [8] platform.

[6] http://pipes.yahoo.com
[7] http://services.alphaworks.ibm.com/qedwiki
[8] http://www.salesforce.com/appexchange

In addition to tooling enabling users of the Swashup platform to program in our DSL, there is also a real need to directly facilitate the UI customization aspects of mashups. Currently, this is achieved using the RoR platform's UI primitives by using RHTML and AJAX library tags (e.g., prototype, script.aculo.us, and others). One possible direction is to add support for UI customization directly in our mashup DSL which could make `recipe`s more complete at the point of their creation.

Another direction is enabling the system and platform for collaboration [4]. We started in that direction by enabling the various components of a `recipe` to be tagged with information. In addition we would like to explore adding directly the ability to share, reuse, copy, restrict, and measure the effectiveness of recipes in our tools. This may result in some changes to the DSL, especially in the area of restricting access to recipes for instance. Additionally, with enough usage the tags in the recipes may form a folksonomy, [9] which might help users discover recipes and reuse them.

References

1. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.0 (2002), `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`
2. Fielding, R.T.: Software Architectural Styles for Network-based Applications. Ph.D. thesis, University of California, Irvine, CA (January 2000)
3. Goff, M.K.: Network Distributed Computing: Fitscapes and Fallacies. Prentice Hall, Upper Saddle River, NJ (2003)
4. Tai, S., Desai, N., Mazzoleni, P.: Service communities: applications and middleware. In: SEM-06. Proceedings of the 6th International Workshop on Software Engineering and Middleware, Portland, OR, pp. 17–22 (2006)

[9] http://en.wikipedia.org/wiki/Folksonomy