The Mashup Component Description Language

Saeed Aghaee
Faculty of Informatics, University of Lugano (USI)
via Buffi 13, 6900 Lugano, Switzerland
saeed.aghaee@usi.ch

Cesare Pautasso
Faculty of Informatics, University of Lugano (USI)
via Buffi 13, 6900 Lugano, Switzerland
c.pautasso@ieee.org

ABSTRACT

Mashups can be seen as the result of software composition applied to the Web. One of the characteristics of mashup development is the heterogeneity of its building components in terms of logical layering (e.g., user interface, application logic, and data), access method (e.g., REST, SOAP), and composition technique (e.g., scraping vs. clipping, synchronous vs. asynchronous interaction, discrete vs. streaming). This poses a challenge towards the design of mashup tools aiming at lowering the barriers of mashup development, as this heterogeneity needs to be abstracted. In this paper, we address this challenge by proposing a new JSON-based domain-specific language for describing heterogeneous mashup components, called the Mashup Component Description Language (MCDL). MCDL lies at the core of a meta-model for mashup component modeling, and can be used for component discovery and classification but also for user-centric mashup development as it decouples the interface of a mashup component from its underlying implementation technologies.

Categories and Subject Descriptors

H.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based Services*

General Terms

Standardization, Languages

Keywords

Mashup, Mashup Components, Component Model, Language

1. INTRODUCTION

Mashups are built by composing multiple components of different types: Web services, Web data sources and Web widgets. Not only mashup components may span all tiers (user interface, application logic, and data) of a Web information system [23], but within each tier they need to be composed using various techniques (e.g., Web scraping [18], Web clipping [19], synchronous remote/local service invocation, asynchronous interaction or subscription to feeds and data streams). Furthermore, different mashup components can be made accessible through different access methods and technologies (e.g., POX, REST [17], or SOAP). To build a mashup out of such heterogeneous components, all these skills thus should be mastered by the developer.

For this reason, one of the main challenges of mashup tools consists of abstracting away such heterogeneity behind a uniform, and easy-to-understand component meta-model [1]. A uniform meta-model adapts heterogeneous mashup component interfaces to a common interface, and its aspect of being easy-to-understand is thus defined by how much its interface conceals the information about the underlying component technologies. This challenge is not fully addressed by the state-of-the art mashup tools as the majority of them do not support all types of mashup components [2].

To address these challenges, in this paper we propose a meta-modeling approach for heterogeneous mashup components that forms the basis for a novel domain specific language, called the Mashup Component Description Language (MCDL). MCDL is rendered using the JSON syntax and supports various composition styles of mashup components. Moreover, it allows to completely encapsulate an executable representation of a component into an abstract interface with only minimal technical aspects.

The above characteristics position MCDL as a comprehensive specification that can potentially be used as a standard by component providers. With the increasing number of existing mashup tools(e.g., [9], [7], and [13]), a standard component meta-model can potentially facilitate portability of components between them. With this, component discovery will not be bound to the local library of a single mashup tool. Instead, a shared repository, that can be easily accessed by component providers, can be established.

The rest of this paper is organized as follows. In the next section we present our meta-model for mashup components. Based on the proposed meta-model, in Section 3 we introduce and explain our MCDL language. Section 4 is dedicated to review the related work before we conclude the paper in Section 5.

2. MODELING MASHUP COMPONENTS

A model is a "representation of reality intended for some definite purposes" [15]. The power of models lies in their ability to abstract and unify heterogeneity of the target reality [5]. A meta-model, in turn, is a language that defines

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
iiWAS2011, 5-7 December, 2011, Ho Chi Minh City, Vietnam.
Copyright 2011 ACM 978-1-4503-0784-0/11/12 ...$10.00.

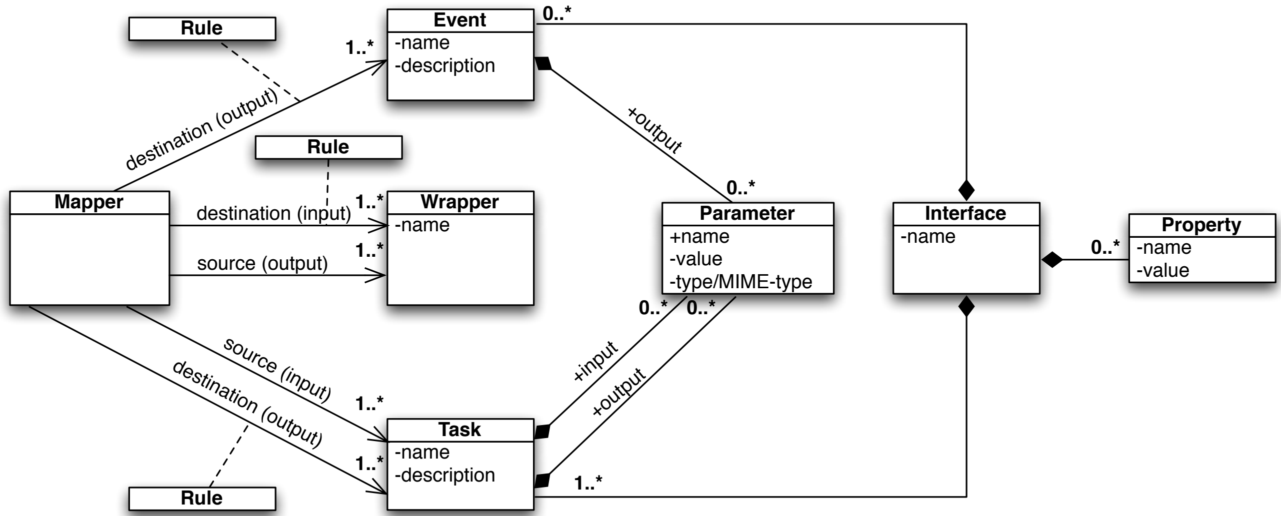

Figure 1: A meta-model for mashup components

the characteristics of a model. One important characteristic of models that conform to a meta-model is that they can be composed with each other as they follow the same ontology [4].

We depict the meta-model as a UML class diagram (Figure 1), whose main classes as well as the containment and reference relationships existing between them will be described in the rest of this section. The six classes of the meta-model (interface, task, event, parameter, mapper, and wrapper) form two layers of abstraction on top of a given mashup component. The first and underlying layer is provided by the wrapper class which models various invocation and composition styles of mashup components. The second layer is achieved through the interface class which contains the event and task classes, which, in turn, contain the parameter class. This layer provides a homogeneous, easy-to-understand interface concealing the underlying technological heterogeneity. The connection between these two layers is thus modeled with the mapper class.

2.1 Interface

The *interface* class is the main element of the meta-model, which mediates interactions between mashup components and their environment (i.e., the mashup that composes the components). This is done through a set of *tasks*, and *events* attached to their interfaces. An interface can also have a set of properties which jointly identify the mashup component that is supposed to be modeled. For instance, such properties of a given component are exemplified by its `component publisher`, a brief human readable `description` of its functionality, and the `version` under which it is modeled.

2.2 Task and Event

The *task* and *event* classes represent the passive and active behavior of a given mashup component. The task class models the functional behavior of a mashup component. It both represents the synchronous invocation of a component's functionality (such as a call to a remote Web API) or a data source that generates some output given a set of input data. From the control flow point of view, such behavior is considered passive inside a mashup, as the corresponding component is executed only when the control reaches it. The task class may contain abstract *parameters* associated to its input and output ports. Input parameters should be filled with values before the invocation. The output parameters will eventually store the result of the invocation if it does not throw any exception.

Whereas the execution of tasks is driven by the mashup, events fire independently of the mashup. They can be used to represent sources of streaming data as well as user interactions with a Web widget included in the mashup. Events may produce output data in their associated output port parameters, even if it is possible to model signal-like events which do not carry data but are nevertheless used to trigger the execution of other tasks.

2.3 Parameter

The input and output data parameters bound to tasks and events are represented by the *parameter* class. This class includes a mandatory attribute holding the `name` of the parameter. The optional `value` attribute contains the default value for the parameter. The parameter class has also a `type` attribute whose value can be one of the basic JSON types (`string`, `boolean`, `number`, `array`, `object`, and `null`). Alternatively, it can be used to store the MIME type of the data. The MIME types are registered in the IANA (Internet Assigned Numbers Authority) media types repository. For example: `text/plain`, or `application/rss+xml` exemplify valid MIME types. Parameter types can be further extended to model semantic meta-data.

2.4 Wrapper

The *wrapper* class is responsible for modeling different types of mashup components, distinguished by the technology for their invocation and utilization (Figure 2). The wrapper classes are organized into a hierarchal inheritance

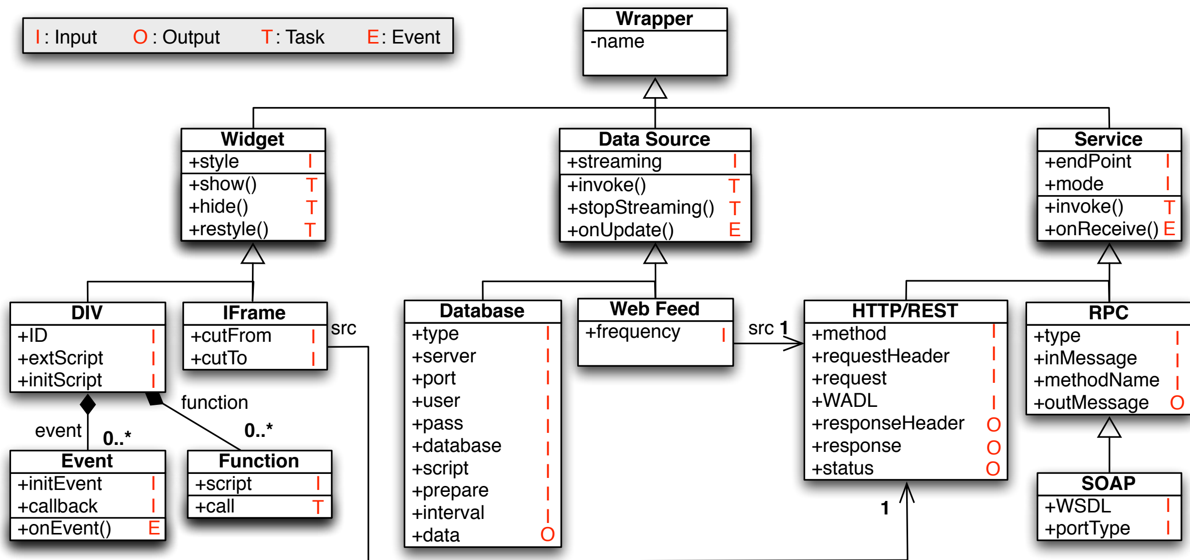

Figure 2: Wrapper hierarchy

structure. At the top of the hierarchy is the abstract wrapper class, which is inherited by three subclasses each of which corresponds to a different type of mashup component: *widget*, *data source*, and *service*.

The three wrappers (widget, service, and data source) have, in turn, a set of subclasses that are called *wrapper types*. Each of these types is associated with a single specific technology. For instance, the `database` class, which is a wrapper type defined under *data source*, is used to model the interaction with a database. To fully model a mashup component may require more than one wrapper types depending on how many different technologies are required for invoking the target mashup component.

The wrapper types store specialized configuration data structures adhering to the characteristics defining the interaction with different mashup components at runtime. Such data structures include as a set of attributes, that are conceptually grouped into *input*, and *output*, as well as a number of operations, that operate on the attributes, and are interpreted as either *task* or *event*. A task operation operates on both input and output attributes. For instance, most of the wrapper types have the `invoke` operation which takes the input attributes, invokes the corresponding mashup component, and will return the output attributes if the component invocation succeeds. A task can initiate (as a class contractor) or terminate (as a class desctructor) its corresponding component life-cycle, which is a subset of the target mashup execution life-cycle.

An event operation, on the other hand, operates only on output attributes. Basically, it is bound to a wrapper internal action or event whose occurrence causes its execution. Therefore, the implementation and interpretation of an event may differ from one wrapper type to another. For instance, in the database wrapper, the acquisition of new data triggers the execution of `onUpdate` event, whereas in the DIV wrapper an event (`onEvent`) is triggered by the user interaction with the component visual interface, and is implemented using the callback mechanism in JavaScript. Events are only triggered during the active part of a component life-cycle.

It should be noted that the attributes and operations of a wrapper type are predefined according to the technology the wrapper is intended to model, as opposed to task, event, and parameter objects that are defined by the user, though they will be eventually mapped to the wrapper types through the mapper class.

2.4.1 Widget

A Web widget is a form of mashup components providing a mashup with independent visual presentation of its underlying aggregate data. Some widgets may also give a mashup access to reusable pieces of functionality or data. A widget life-cycle starts with calling the `show` task, which initializes all input attributes with value, and ends when the `hide` task is called. During this life-cycle, the `restyle` task can be called multiple times to change the value of the `style` attribute containing a valid CSS stylesheet.

- **DIV.** A DIV widget is created on the client-side using external and internal JavaScript code. This kind of widgets is well exemplified by Google Maps APIs offering an external JavaScript library to build customized map visualizations. The `ID` attribute of the wrapper type specifies the actual ID of the DIV element that is supposed to contain the widget. The URL pointing to the external JavaScript libraries can be inserted as the value of the `extScript` attribute. The `InitScript` attribute needs to be set by the JavaScript code that actually wraps the widget in the DIV element.

The `Function` class models JavaScript functions. This class has a task operation (`call`) that executes the script specified in the `script` attribute. The operation can be then associated to a n interface task whose parameters can be mapped and used inside the code as regular variables using the template rule (see Section 2.5).

In addition to functions, a DIV wrapper can have a set of event objects which represent events in JavaScript. In JavaScript, events are handled through the callback mechanism, in which a callback function is called whenever its corresponding event has occurred. The declaration and implementation of the callback function is the value of `callback` attribute. The script that attaches the callback function to the event should be then inserted as value of `initEvent` attribute.

• **IFrame.** IFrame widgets are used to embed HTML-based User Interface (UI) contents from a remote Website, partially (i.e., Web clipping) or completely. The value of `src` attribute is a HTTP wrapper used to fetch the Website (with `Get` method) that is to be embedded inside the target container. In case of Web clipping, the `CutFrom` attribute contains the starting point of the clipping process that can be a piece of string which matches a part of the target Website HTML code. Likewise, the `CutTo` attribute represents the ending point of the cutting process. For instance, cutting a table from a website in the simplest case requires setting `<table` and `table>` as the values of respectively the `cutFrom` and `cutTo` attributes.

2.4.2 Data Source

A data source provides remotely accessed content on the Web. Data coming from remote sources can also have a real-time essence. In this case, the wrapper allows the streaming of data from a data source by setting the `streaming` attribute to `true`. This should be done through the first call to the `invoke` task which initiates all the input attributes and starts the component life-cycle. If streaming is not enabled, the component life-cycle ends as soon as the response from the `invoke` task arrives. Otherwise, it will only end by calling the `stopStreaming` task. Moreover, while streaming is activated, the `onUpdate` event is fired whenever a new data stream item is available.

• **Database.** The support for databases is achieved through the use of the `database` class. A database can also be used to generate a stream of data. In the simplest case (which is the only case supported by our meta-model), this can be done by running a query (`script`) in a given time interval. This interval is a value of the `interval` attribute, and its length is a factor of how fast the target data changes.

• **Web Feed.** Web feeds are popular components for building various data mashups, which are usually delivered in RSS/Atom formats. Feeds can be consumed as a continuous stream of data using a technique called polling. Using this technique, the feeds are periodically checked for updates. The feed wrapper type uses the `frequency` attribute to specify the update interval.

2.4.3 Service

Services are reusable modular business logic that are made available to remote clients through a URL (`"endPoint"`). The interaction between a client and a service provider (`mode`) can be either blocking, in which the invocation of the service (by calling `invoke()`) must wait until the response is ready before it can proceed, or non-blocking, in which the client invokes the service and registers a callback (`onReceive()`) to be triggered whenever the response comes back.

• **RPC/SOAP.** The three popular forms of Remote Procedure Call (RPC) are SOAP, XML-RPC [21], and JSON-RPC [11]. All of these protocols run over the HTTP or secure HTTP (HTTPS), except for SOAP messages that can be also transmitted over SMTP and other protocols. The protocol in use, therefore, can be found at the URL to the service endpoint (`"endPoint"`). To communicate requests (`inMessage`) and responses (`outMessage`), JSON-RPC messages use JSON, whereas SOAP and XML-RPC messages are encoded as XML. To determine the type of the RPC service, the `type` attribute can be set to `"SOAP"`, `"XML-RPC"`, or `"JSON-RPC"`. In case of SOAP, a service usually comes with a Web Services Description Language (WSDL) document (`WSDL` attribute), which defines the service interface. In case that the interface contains more than one port types, `PortType` specifies the desired one.

• **HTTP/REST.** HTTP is the communication protocol used on the Web. Following the constraints of the REST architectural style, it is used for stateless client-server interaction with Web servers by sending and receiving hypermedia documents (e.g., XML, HTML, etc.) via four `methods` (GET, POST, PUT, and DELETE). The HTTP communication mechanism is based on request and response messages, each of which is decomposed into a header (`requestHeader` and `responseHeader`) and a body (`request` and `response`). HTTP/REST services may have a Web Application Description Language (WADL) document (`WADL` attribute), in a similar manner as SOAP services, with the difference that it is not yet widely accepted as WSDL.

2.5 Mapper and Rule

The *mapper* class serves as a mechanism providing a link between the predefined tasks and events of a wrapper type and the tasks and events attached to an interface by defining how their corresponding wrapper attributes and abstract parameters are mapped to each other. This class contains two attributes: `destination`, and `source`, which hold a reference to parameters and wrapper attributes respectively or vise versa, depending on the mapping direction.

Using the mapper class, abstract parameters can be passed from an abstract interface as input to a wrapper task operation which controls the technology-dependent invocation of its corresponding mashup component. Likewise, the attributes holding the result of the invocation task or the data associated with an event operation are transformed to the abstract interface as abstract parameters. To do so, the *rule* class is used to designate how the mapping from a source to its destination should be accomplished. This class contains three subclasses each of which represents a specific rule prescribed for a particular purpose: *correspond*, *template*, and *parser*.

• **Correspond.** A one-to-one association between a source and its destination can be enabled by the *correspond rule*. In doing so, the value of the source is exactly copied to its destination parameter. This requires both source and destination have the same data type, or otherwise be implicitly convertible. For instance, the numerical type or the majority of Media types (such as XML, JSON, etc.) can be converted to the string type. For example, `User` and `Pass` attributes of `database` wrapper can be set at run-time by an abstract task taking two input parameters of string type, which are mapped with a one-to-one correspondence to these attributes.

• **Template.** The *template* rule can be selected to correspond multiple sources to single destination through a template. This template string consists of the value which is

```
{ "Interface": {
    "name":"",
    "property":[
      {"name":"", "value":""
    }],

    "task": [
      {"name": "",
       "description":"",
       "input": [
         {"name": "", "value": "",
          "type": "" | "mime-type": ""}
       ],
       "output": [
         {"name": "", "value": "",
          "type": "" | "mime-type": ""}
       ]
    }],

    "event": [
      {"name": "", "description":"",
       "output": [
         {"name": "", "value": "",
          "type": "" | "mime-type": ""}
       ]
      }]
    },

    "service_wrapper": [
      {"type":"", "name":"",
       "input": [
         {"name": "", "value": ""}
       ]
    }],

    "datasource_wrapper": [
      {"type":"", "name":"",
       "input": [
         {"name": "", "value": ""}
       ]
    }],

    "widget_wrapper": [
      {"type":"", name":"",
       "input": [
         {"name": "", "value": ""}
       ]
    }],

    "mapper": [
      {"source" : ["ref"],
       "destination" : [{"name":"ref",
          "correspond":"ref" |
          "template":"" |
          "parser":{ "language":"",
                     "query":""}}]
      }]
}}
```

Figure 3: JSON schema for MCDL

supposed to be assigned to the destination parameter, as well as a set of placeholders distributed within the string value. These placeholders are represented as pairs of opening and closing brackets, each of which contains the name of a single source parameter or attribute which will be replaced with its value at runtime (i.e., during the execution of the component). One frequent use of the template rule is for passing URL parameters (when the corresponding WADL document is not present). Consider invoking the Twitter search API with the HTTP wrapper. Using the template rule we can pass a source parameter to the URL attribute of the HTTP wrapper as follows.
`http://search.twitter.com/search.atom?q={source}`
Where `source` refers to the name of the source parameter.

- **Parse.** The *parse* rule entails splitting a source to multiple values and further associating these values to destinations. This rule supports different query languages. A query written in a specified language, is used for extracting the value of the destination from its source. The language can be one of XPath, exclusively for XML-based formats (e.g., SOAP, RSS, ATOM, XML-RPC, etc.), or a regular expression for scraping data out of any kind of text-based formats such as HTML or JSON. This rule is specially useful for extracting data from MIME types such as SOAP messages, RSS/Atom, and XML.

3. JSON SCHEMA FOR MCDL

Having mentioned the meta-model in Section 2, we can define the JSON schema for MCDL that corresponds to the meta-model as depicted in Figure 3. We choose to rely on the JSON syntax not only to take advantage of many related tools and libraries but also because component libraries specified using the MCDL language will be mainly managed from within mashup composition tools running inside a Web browser, where JSON is one of the most efficient data representation and interchange formats [6].

The schema contains the meta-model classes nested as specified in the meta-model through the use of the composition relationships. The abstract interface class is represented as `interface` object containing an array for defining its properties (`property`), as well as two arrays: one for the tasks (`task`) and another for the events (`event`). Parameters are also defined as items of an array which is named as `output` and can be included in every task or event item. The task items can also contain another array of parameters called `input`, since tasks in the meta-model, as opposed to events, can also take input parameters. The parameter items in MCDL take `name`, `value`, and `type`, similar to the abstract parameter class in the meta-model.

The JSON arrays in the schema named as `service_wrapper`, `datasource_wrapper` and `widget_wrapper` respectively express the service, data source, and widget wrappers in the meta-model. Each array can have multiple items which refer to different wrapper types defined under their meta-model class, and are identified by the `type` variable which takes the wrapper type name as a string such as `"HTTP"`,`"DIV"` and `"SOAP"`. The input attributes of a wrapper type are defined in the `input` array, only when they should be initialized by a constant value.

Finally, the mapper class is expressed as an array whose items identify mapper objects. In each mapper object `source` and `destination` make a reference to either a parameter name or wrapper attribute. To ensure their uniqueness, references can be prefixed with the names of the object they belong to. This way they can also be referenced from the mapper objects. To exemplify this, `user` which is an input attribute of the `database` wrapper can be referred to by `dbwrapper.user`, assuming `dbwrapper` is the name of the wrapper object. Likewise, references to parameters can be prefixed by the name of the task/event they belong to.

4. DISCUSSION AND RELATED WORK

The simple but powerful nature of mashup development provides a suitable platform for empowering end-user programming and composition on the Web. This is observed by the emergence of the mashup tools aiming at reducing the programming efforts required for mashup development down to drag-and-drop-and-connect activities. Examples are Yahoo! pipes [22], Intel Mash Maker [9], IBM Mashup Center [10], MashArt [7], and ServFace [13].

However, the main shortcoming of these tools is that they cannot be used to create any type of mashups. From the software composition perspective [3], this inability is caused by the lack of a unified approach to model all types of mashup components. To address this challenge, in this work, we introduced MCDL that not only supports existing types of mashup components (widgets, data sources, and services) and their corresponding implementation technologies (e.g.,

IFrame, SOAP, etc.), but also models both the active and passive behavior of a component (i.e., events and tasks).

There are a number of related work that should be discussed here. The Enterprise Mashup Markup Language (EMML) [8], for instance, is a XML-based standard component meta-model for mashup components providing services and data sources. Though it supports variety of service and data source types, it does not include Web widgets. JOpera, which is a rapid visual service composition tool, also abstracts away variety of service types and Web widgets behind a uniform interface [14], but fails to model stateful event-based behavior of mashup components such as user interaction with a widget. Concerning widgets, Yu et. al. [24] presented a new model for UI components (i.e., widgets) representing the component UI-related specifications (e.g., width, heights, etc.) as well as its corresponding tasks and events. In the same line, Wilson et. al [20] extended the widget specification language proposed by W3C to add support for event-based behavior.

However, none of the work mentioned above offered a component modeling approach, in which all types of mashup components, including widgets, data sources, and services, are supported. The work most closely related to ours is [16], where the authors introduced a meta-model that is able to describe not only different types of services (HTTP/REST and SOAP) but also widgets and UI components. The main advantage of our work lies in the complete separation of technology specific concerns (wrapper) from technology independent concerns (interface). This in turn maximizes flexibility of our meta-model which ensures expanding in future by adding more wrapper types, as new technologies and standards emerge on the Web (e.g., WebSockets [12]).

5. CONCLUSION

The heterogeneity of mashup components is an important obstacle towards developing abstract mashup composition languages. To address this challenge, in this paper we proposed a novel meta-model for describing mashup components which not only provides an abstract, unified interface for mashup components, making them connectable to each other despite their heterogeneity, but also hides their underlying technology-specific invocation and execution mechanism from their abstract interfaces. The meta-model is then represented by MCDL which is a domain specific language inheriting the JSON syntax and parser.

However, our proposed meta-model does not yet support mashups as components (composite components) as well as components utilizing multiple wrapper types that are to be orchestrated according to a specific workflow. The latter case can be exemplified by Zoho API [25] in which `HTTP` is used to retrieve the URL pointing to the newly created editor that should be wrapped in an `IFrame`. Supporting the above mentioned types of components is the objective of our current work.

Acknowledgments

This work is partially funded by the Hasler Foundation under the project LISA (Grant No. 11019).

6. REFERENCES

[1] S. Aghaee and C. Pautasso. End-user programming for web mashups: Open research challenges. In *Proc. of ICWE 2011 (Doctoral Symposium)*, 2011.

[2] S. Aghaee and C. Pautasso. An evaluation of mashup tools based on support for heterogeneous mashup components. In *Proc. of ComposableWeb 2011*, 2011.

[3] U. Assmann. *Invasive Software Composition*. Springer, Secaucus, NJ, USA, 2003.

[4] U. Assmann, S. Zschaler, and G. Wagner. Ontologies, Meta-models, and the Model-Driven Paradigm. *Ontologies for Software Engineering and Software Technology*, pages 249–273, 2006.

[5] J. Bézivin. On the Unification Power of Models. In *Software and System Modeling*, May 2005.

[6] D. Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML 2006*, 2006.

[7] F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. Hosted universal composition: Models, languages and infrastructure in mashart. In *Proc. of ICCM 2009*, 2009.

[8] EMML. `http://www.openmashup.org/`.

[9] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi. Intel mash maker: join the web. *SIGMOD Rec.*, 36:27–33, 2007.

[10] IBM Mashup Center. `http://www.ibm.com/software/info/mashup-center`.

[11] JSON-RPC. `http://json-rpc.org/`.

[12] P. Lubbers and B. Albers. Harnessing the power of HTML5 web sockets to create scalable real-time applications presentation. Web2.0 Expo SF, May 2010.

[13] T. Nestler, M. Feldmann, G. Hubsch, A. Preussner, and U. Jugel. The servface builder - a WYSIWYG approach for building service-based applications. In *Proc. of ICWE'10*, 2010.

[14] C. Pautasso and G. Alonso. From web service composition to megaprogramming. In *Technologies for E-Services*, volume 3324 of *LNCS*, pages 39–53. Springer, 2005.

[15] M. Pidd. *Tools for Thinking: Modelling in Management Science*. Wiley, 2 edition, Mar. 2003.

[16] S. Pietschmann, V. Tietz, J. Reimann, C. Liebing, M. Pohle, and K. Meissner. A metamodel for context-aware component-based mashup applications. In *Proc. of iiWAS 2010*, 2010.

[17] L. Richardson and S. Ruby. *Restful web services*. O'Reilly, first edition, 2007.

[18] M. Schrenk. *Webbots, Spiders, and Screen Scrapers*. No Starch Press, 2007.

[19] I. Smith. Doing web clippings in under ten minutes. Technical report, Intranet Journal, March 2001.

[20] S. Wilson, F. Daniel, U. Jugel, and S. Soi. Orchestrated user interface mashups using w3c widgets. In *Proc. of ComposableWeb 2011*, 2011.

[21] XML-RPC. `http://www.xmlrpc.com/`.

[22] Yahoo Pipes. `http://pipes.yahoo.com/pipes/`.

[23] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *IEEE Internet Computing*, 12:44–52, September 2008.

[24] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *Proc. of WWW 2007*, 2007.

[25] Zoho API. `https://apihelp.wiki.zoho.com/`.