

MU: an hybrid language for Web Mashups*

Michele Mostarda
Asemantics s.r.l.
Circonvallazione Trionfale, 27 00135
Rome, Italy
michele@asemantics.com

Davide Palmisano
Asemantics s.r.l.
Circonvallazione Trionfale, 27 00135
Rome, Italy
davide@asemantics.com

ABSTRACT

Web Mashup, Web 2.0, recombinant or remixable Web are all terms with the same meaning: informally, they express the possibility of building applications that are able to manage data contained in different repositories exposed as Web services. The availability of a broad range of Web services for different purposes and domains, from online bids to the weather forecasts, motivates the development of several different mashup applications throwing up a wide class of problems related, mainly, to the data format interoperability. The high number of the data formats, together with a huge technological heterogeneity creates the need for a framework that easily allows the development of such mashup applications. The main objective of this paper, therefore, is to present a novel approach based on a hybrid functional-logic high-level language called *MU* that allows the description of data source aggregation and the manipulation and presentation phases over multiple external sources with an extremely compact and flexible syntax. We also present an effective use case implemented using "em-up", our reference implementation of the *MU* language, where an example of mashup application is shown in detail.

Categories and Subject Descriptors

H.3.5 [Information Systems]: Online Information Services
; D.1.6 [Software]: Programming Techniques—*Logic Programming, Miscellaneous*
; D.3.2 [Programming Languages]: Language Classifications—*Multiparadigm languages*

General Terms

Web Mashups, Unification, UI Induction, JSONModel, JSON-Path, Java, Google Web Toolkit

Keywords

web mashups development framework, functional-logic hybrid language, UI induction

*Copyright is held by the International World Wide Web Conferences Steering Committee (IW3C2)

Copyright is held by the author/owner(s).
WWW2009, April 20-24, 2009, Madrid, Spain.

1. INTRODUCTION

MU is a hybrid functional and logic programming language meant to perform Web mashups. The main goal of *MU* is to provide a novel approach in writing web mashup applications both in industrial and scientific fields. This paper, mainly intended as an introduction to the language, is organized as follows: The remaining part of this introduction is aimed at describing in detail, the chief design principles behind our solution. In section 3 we will provide an overview of the language, while section 4 will provide a meaningful use case with the aim of demonstrating the potential of our solution. An informal comparison with related works will close the paper.

1.1 Key Principles

MU is designed around the following key principles:

1. It is a scripting language. This means that any mashup application written with *MU* can be elaborated, stored and exchanged as text. Neither special editors nor browser plugins are needed to deal with *MU*.
2. It gives support for *unification*. The unification is based on the ability of defining *JSONModels*, that allow to *MU* to unify over *JSON* data structures. The unification is the base mechanism of the *predicate overloading* (see 3.6.2) and the *UI Induction* (see 3.7). The concepts behind unification are explained in detail in section 3.6.1.
3. It is based on the *Type Morphing* paradigm. The type morphing paradigm is the ability of the language to cast any primitive type to another where needed. Every *MU* type is internally represented as *JSON*. The transformation across different types is done by following a set of predefined rules. The transformation ability is a key aspect when operating with semi-structured and live data. Let's suppose we have written a mashup script using *MU* to extract a number from each section of a certain data source. After a change in the original data, instead of a number the script will find a list of elements. *MU* deals with this minor issue casting the list to a number, returning the size of the list, without raising exceptions. This automatic behavior can be avoided when needed.
4. It provides *JSON Path*¹ native support. Being the *JSON* format the internal data representation used

¹<http://goessner.net/articles/JsonPath/>

for any *MU* type, it is needed to have operators to address sub parts of such type instances. *JSONPath* is the candidate format for this purpose, since it is compact, powerful and easy to understand.

5. It provides the *User Interface Induction*, that is the ability of auto generate a User Interface from a *JSON* data. The user interface induction is a concept that defines a way to auto generate entire (or parts of) user interfaces. The auto generation is needed for the semi structured and time changing nature of the data processed by mashup applications. The *UI Induction* is explained in section 3.7.
6. It defines both *JRE* (Java Runtime Environment) and Javascript profiles. *MUP* is written in *Java* (JRE 1.5.0) but the language core can be translated in *Javascript* with *GWT* (Google Web Toolkit), allowing it to run entirely in a web browser. The possibility to run *MUP* as a *GWT* and a *JRE* application is called multi profiling. The multi profiling of the language is not discussed in this paper, for further information see section 8.

The concepts behind *MU* largely take their inspiration from the cited works. In particular the concept of *editing and manipulating semi-structured live data* has been taken from *Mash Maker* [3]. The *Untyped Tree Data Model* is the origin of the *UI Induction* logic. *MU* also takes its inspiration from *Prolog* [6] for the *logic programming* approach and in particular for the *Unification* concept and is inspired by *Erlang* [8] for what concerns parallelization and distribution. These features are still under analysis (see section 7), but in general the adoption of the functional paradigm is furthermore motivated by the implicit capacity of this class of languages to write applications that take advantage of the distributed model[5].

2. BACKGROUND

The recent trend, often referred to as *Web 2.0*, where large Web sites provide access to their data with APIs, can be considered as the formalization of an old idea where the information contained on the Web, mainly intended to be used by people and often stored in repositories with heterogeneous schemas, can be handled in a programmatic way in order to build cross-domain applications.

According to a classical and informal definition that describes a mashup as:

"an application that combines data, either through APIs or other sources, into a single integrated user experience"[9]

it is not difficult to recognize the enormous amount of Web services currently available to be used as data sources for new mashups and the implicit potential hidden in this 'data space'[4]. Application fields vary from simple data aggregation to novel techniques such as the one represented by the news recommendation.

Imagine, for example, a simple mashup that retrieves the most cited word among the overall public timeline of one *microblogging* application, and uses it to retrieve the last 10 items from an *RSS* news feed. Apart from its simplicity

it shows how the data on the Web can be composed to satisfy information needs otherwise unsatisfiable without using some sort of combination of the original data sources.

Moreover, the social networking offers the users the possibility to access and manipulate their own social graphs and, since social networks seldom offer native APIs, this makes the development of mashup applications more enjoyable. In fact, users can aggregate around their personal data information taken from different applications.

Another set of opportunities is provided by the current proliferation of the so-called semantic web services[2], where it is possible to obtain *RDF* descriptions of some concepts, so enforcing the possibility of building applications where the semantic descriptions of web contents can be used to enrich the data creation, discovery and aggregation.

Although this degree of heterogeneity heavily impacts on the mashup development process[10] making it complex and time-consuming, these kinds of applications have been recognized to usually follow a specific pattern made up of source identification, data selection and manipulation, and the rendering of the resulting composition.

This consideration, jointly with the hypothesis that all the formats adopted by the services for the data delivery (*RSS*, *ATOM*, *HTML*, a specific *XML*) share a common tree data structure [3], led us to develop a hybrid functional logic programming language that, as widely described below, allows mashups to be written and compiled that can be deployed as client-side applications (using the Java to JavaScript provided by *GWT*²) or as a server-side service. The main idea behind its hybrid nature is motivated by the functional paradigm itself: since the data to be handled are modeled as trees, the functional approach is the most suitable one[1].

3. THE MU LANGUAGE: AN OVERVIEW

As introduced in the section above, mashup applications share a common set of peculiarities that motivate the choice of *MU* to adopt both functional and logical paradigms. Let's begin by introducing how *MU* represents data (the data model) and how it perform data manipulation.

- Data Model. Data subjected to the mashing operation are semi-structured, typically, with tree-based data structures. Both such DOM-like data structures (such as *HTML* static pages or *XML REST* responses) and these semi-structured representations of semantic data (such as *RDF/XML* encoding) can all be accessed using methods based on the concept of the path above a tree. *MU* chose *JSON* as its internal data model. Every *MU* type has a *JSON* counter part, even the *Graph* type is represented as a *JSON* object.
- Data Manipulation. Once the data sources to be mashed up are chosen, mashup applications are usually processed with a flow like the following:

data retrieval where raw data is addressed and extracted from the Web.

²Google Web Toolkit

data decomposition where subsets of raw data are extracted.

data aggregation where the data model is built.

data rendering where the renderization logic is provided in order to produce a presentation of the data model in a way that can be shown to the end user (that can be a human or a system).

This flow can be easily described with a functional language since every step can be modeled as a function. *MU* groups its predicates in *functional sets*, described in section 3.4. The predicates belonging to a functional set are called *operations*.

The data retrieval is related to the *Source Operations Functional Set*. The data extraction is related to the *Inspect Operations Functional Set*. The data aggregation is related to the *Model Operations Functional Set*. Finally the data rendering is related to the *Renderize Operations Functional Set*.

3.1 MU expressions, types and functional sets

The following section is not an exhaustive language technical report, but a rapid demonstration of its main capabilities. The *MU* language main construct is the *Predicate*, the terms *predicate* and *function* are interchangeable.

3.2 Language expressions

MU main expressions are based on **predicate declarations** and **predicate invocations**. Predicate and variable names always start with upper and lower case, respectively. A predicate declaration has the following form:

$$\begin{aligned} & \textit{PredicateName}(\textit{param1}, \textit{param2}) : \\ & \textit{PredicateCall1}(\textit{param1}), \textit{PredicateCall2}(\textit{param2}); \end{aligned}$$

where the part on the left of the `:` is called *predicate head* and the part on the right *predicate body*. In this sample code we're declaring a predicate called *PredicateName* taking a list of formal parameters *param1*, *param2*. The predicate body contains the invocation of a list of other predicates *PredicateCall1*, *PredicateCall2*, that take as actual parameters respectively *param1* and *param2*.

A predicate invocation follows the below form:

$$\begin{aligned} & \textit{PredicateCall1}(\textit{p11}, \dots), \textit{PredicateCall2}(\textit{p21}, \dots) \dots \\ & \textit{PredicateCallN}(\textit{pN1}, \dots); \end{aligned}$$

This invocation will invoke first the predicate with name *PredicateCall1* then the predicate with name *PredicateCall2* then the subsequent predicates until *PredicateCallN* with the given parameters, and will return the value returned by the last invoked predicate (*PredicateCallN*).

MU predicates are like functions: they return the last value processed in their predicate body. The examples below aim to show, from the language syntax point-of-view, its hybrid nature. In fact the following expressions are procedural, functional and logic respectively:

$$\begin{aligned} & \textit{Procedure}(a, b) : r1 = \textit{PredA}(a), r2 = \\ & \textit{PredB}(b), \textit{if}(\textit{Gt}(r1, r2), \textit{PredC}(r1), \textit{PredD}(r2)); \end{aligned}$$

that first computes *r1* as result of *PredA(a)*, then *r2* as result of *PredB(b)*, then if *r1* (cast as *Numeric*) is GreaterThan

r2 (also cast as *Numeric*) returns the result of the invocation *PredC(r1)*, otherwise will return the result of the invocation *PredD(r2)*.

$$\textit{Func}(a, b) : \textit{Sum}(\textit{P1}(a), \textit{P2}(b));$$

This statement declares a predicate called *Func* that takes two arguments *a* and *b*, then invokes predicates *P1(a)*, *P2(b)*, casts their result to *Numeric*, sums them and returns the result.

$$\textit{Pred}(a, b) : \textit{P1}(a) \& \textit{P2}(b) | \textit{P3}(a, b);$$

This statement declares a predicate called *Pred* accepting parameters *a* and *b*, that invokes the predicate *P1(a)*, if its result cast to *Boolean* is *true*, then also *P2(b)* is evaluated, otherwise not and the result of *Pred* is *false*. If the result of *P1(a)* is *true* and the result of *P2(b)* is *true* the result of *Pred* is *true*, otherwise the result of *Pred* is equals to the result of *P3(a, b)*.

3.3 Type Morphing

MU provides a set of primitive types accompanied by a complete mechanism aimed at achieving a complete polymorphism between them. In fact, according to a predefined set or rules, every native type always can be converted to another native type.

Furthermore, every *MU* type has a *JSON* native counterpart, as shown in table 4.1.3. This choice was taken in order to ensure the highest degree of robustness.

The table 3.3 shows an example of type morphing among *Numeric* type and *List* type. The table can be read as:

- a *Numeric* value is cast to a *List* value by creating a list with one element containing that numeric value.
- a *List* value is cast to a *Numeric* value by evaluating the size of the list.

3.4 Functional sets

All the main constructs provided by *MU* can be grouped in 11 different functional sets. The Figure 1 depicts such sets, inside every set are reported the most relevant operations.

A detailed enumeration of all the language constructs is not provided within this paper. For further details it is possible to access the official documentation as described in section 8.

In the following list will be used a notation like:

$$\langle \textit{ReturnType} \rangle \textit{PredicateName}(\langle \textit{CastType1} \rangle \textit{arg1}, \dots, \langle \textit{CastTypeN} \rangle \textit{argN})$$

that means: the predicate *PredicateName* takes *N* arguments where the *i-th* argument with name *arg-i* will be cast internally as a *CastType-i* type; finally the predicate will return a value of type *ReturnType*.

The entities bounded as `<< entity >>` are meta types, and are used to define predicate references and variables.

Source Operations This class comprises all the operations needed to access web resources through the HTTP pro-

Table 1: Type Morphing Example

Origin Type / Destination Type	Numeric	List
Numeric	-	List(<i>n</i>)
List	size(<i>list</i>)	-

Table 2: MU Primitive Types

Primitive Type	Description	Example
Boolean	represents a boolean value	true/false
Numeric	represents any numeric value	1, -2.5, 10 ¹²
String	represents a string	"a", "thisisastring"
List	represents an ordered list of values of any type	[1, "a", true, a : 5]
Map	a map key value in which keys and values can be any type	{a : 1, 2 : 3, "3" : a}
Graph	a graph on nodes and arcs connecting the first ones	'{sub1' : {'pred1' : 'obj1', 'pred2' : 'obj2'}}
JSON	represents a JSON value of any type	"k1" : [1, true, "a"]

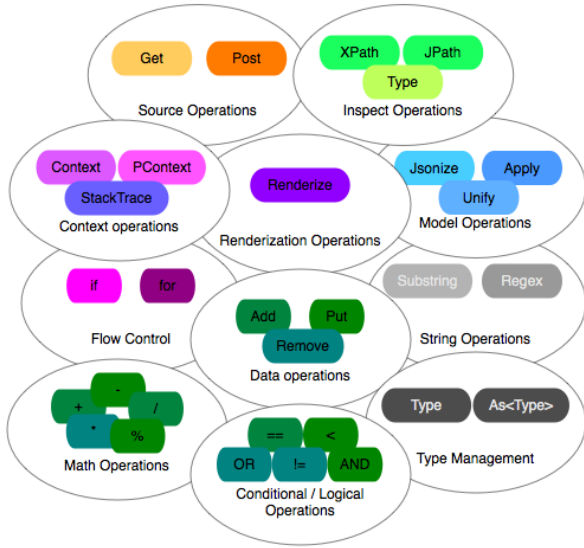


Figure 1: MU Functional sets

to col. Facilities to avoid SOP³ restrictions is also provided.

Inspect Operations Functions that allow the decomposition of the retrieved data are members of this set. For example, the following function performs an *XPath* like operation over an XML string:

$\langle \text{String} \rangle \text{XPath}(\langle \text{String} \rangle \text{path}, \langle \text{String} \rangle \text{xmlContent})$

Note: at the moment of writing the *XPath* support is restricted the the forward operators.

While this one allows to perform a *JSON Path* query:

$\langle \text{JSON} \rangle \text{JPath}(\langle \text{String} \rangle \text{path}, \langle \text{JSON} \rangle \text{data})$

³Same Origin Policy, basically these policies prevents access to most methods and properties across pages on different sites from scripts running on pages originating from the same site

Context Operations Operation provided by this class are intended to handle and manage the programming contextual aspects. Basically, it provides mechanisms for accessing user-defined predicates at run-time.

Render Operations These functions provide the renderization logic of the language.

$\langle \text{JSON} \rangle \text{Renderize}(\langle \text{JSON} \rangle \text{data})$
Transforms a JSON data in a *GUI Native Model*.
 $\langle \text{JSON} \rangle \text{Modelize}(\langle \text{JSON} \rangle \text{data})$
Transforms a JSON data in a *JSON UI Model*.
 $\langle \text{JSON} \rangle \text{Concretize}(\langle \text{JSON} \rangle \text{model})$
Transforms a *JSON UI Model* in a *Native UI model*.

Model Operations Model operations are intended to be used for producing and manipulating JSON models from raw data. For example, the following function:

$\langle \text{JSON} \rangle \text{Jsonize}(\langle \text{String} \rangle \text{template}, \langle \text{List} \rangle \text{parameters})$

This predicate provides the expansion of a JSON template expressed as string with a list of parameters. For example the expression

$\text{Jsonize}(\{'a1' : \%s, 'a2' : \%s, 'a3' : [\%s]'\}, \text{List}('v1', 'v2', 'v3'));$
will return the object
 $\{'a1' : 'v1', 'a2' : 'v2', 'a3' : ['v3']\}$

$\langle \text{Map} \rangle \text{Unify}(\langle \text{JSON} \rangle, \langle \text{JSONModel} \rangle)$

This predicate provides unification between a JSON data structure and a JSON model, the result is a map which keys are the variables defined in the *JSONModel* and the values are the unified *JSON* data.

$\langle \text{List} \rangle \text{Apply}(\langle \text{String} \rangle, \langle \text{JSONModel} \rangle)$

This predicate visits recursively the JSON argument and invokes the predicate with name for each element found.

Flow Operations A collection of procedural flow control operators.

```
< Value > if(< Condition >, << Predicate >>
ifPredicate, [<< Predicate >> elsePredicate])
```

If the value returned from the *condition* predicate (cast to *Boolean*) evaluates to *true* then the predicate *IfPredicate* is evaluated, otherwise the *elsePredicate* is evaluated. The *if* predicate returns the result of the evaluated branch predicate.

```
for(< List > items, << Variable >> var,
<< Predicate >> target)
```

The *for* flow control iterates over all the elements of list *items*, valorizing the variable *var* with the current element, and invoking the *target* predicate that in its own actual parameters should contain *var*. All the results of the invocation of *target* for the different elements are collected in order of execution and returned at the end of the iteration. For example the statement

```
for(List('a', 'b', 'c'), i, Print('current : ', i, ' '));
```

will return the list

```
['current : 1', 'current : 2', 'current : 3'].
```

Data Operations As long as *MU* provides complex data structures as Lists and Maps, which contributes to its high-level nature, operations included in this set are intended to be used for accessing such kind of data structures in a native way. Below the declaration of a function that adds a *triple* into a *graph* is provided:

```
< Graph > GAddArc(< Graph > graph, < Value >
from, < Value > label, < Value > to)
```

String Operations A minimal set of functions for string manipulation.

Math Operations A set of common algebraic operators. Provide basic math operations.

Conditional and Logical Operations A set of logic operators.

Type Operations A collection of operators to handle casting among data types.

3.5 JSON Model

To understand the unification and the predicate overloading, it is needed to learn first the concepts behind *JSON Model*. The *JSON model* extends the concept of *JSON object* by introducing variables. The *JSON Model* grammar is defined as an extension of the *JSON specification*⁴. Basically the *JSONModel* grammar adds the possibility of using variables and rest operators inside a *JSON object*.

The following example is a well formed *JSONModel*, where we have two variable values *a* and *b* and a rest variable *o*.

```
{ k1 : a, k2 : b | o }
```

In the following sections *JSON Model* syntax will be further explained with regards to its application in the language.

⁴<http://www.json.org>

3.6 Unification and Overloading

3.6.1 Unification

According to the first-order logic *unification* definition[7], *MU* provides inner mechanisms for its resolution in order to match a data model with a data structure. Since *MU* basic data models are represented using the *JSON Model* syntax and the data structures as *JSON objects* the unification can be resolved as a possible assignment of the data structure to the model.

Given the previous example, "*k1*" and "*k2*" are constants and matches with constants with the same name. The *a* and *b* are variables and matches with the values associated the related keys is the unified *JSON data*. The *o* variable is a rest variable that unifies with the rest of the object not unifying with the first part, if any.

Assuming that the symbol *::=* means *unifies with* and the symbol *→* expresses the result of the unification the following example shows a unification with its result:

$$P1(\{k1 : a, k2 : b|o\}) ::= (\{k1 : 1, k2 : 2, k3 : 3\}) \rightarrow P1(a := 1, b := 2, c := \{k3 : 3\})$$

where the *JSON Model* represented as $\{k1 : a, k2 : b|o\}$ is unified with the *JSON object* $\{k1 : 1, k2 : 2, k3 : 3\}$ obtaining the assignment at the right of the arrow in the expression above.

3.6.2 Overloading

Another key-aspect of our solution is provided by the support for predicate overloading. In fact with *MU* it is possible to use the same predicate or function name with different arguments, obtaining different behavior. The overloading is based on the *unification* concept, this means that it is possible to use *JSON models* as predicate arguments.

An example of predicate overloading is listed in the code example below:

```
OL([a, b, c], v) : Print('overload1');
OL([a, b, c, d], v) : Print('overload2');
OL("k1" : v1, "k2" : v2, v) : Print('overload3');
OL("k1" : v1, "k2" : v2, "k3" : v3, v) : Print('overload4');
```

where we define four overloads of the same predicate *OL*. The predicate *Print(value)* prints the given value on the system out of the *MU* console. Now we can discriminate among the different overloads by invoking *OL* with different argument structures.

For example, the invocation of *OL* as:

- *OL(List(1, 2, 3), 4)*; will print *overload1*
- *OL(List(1, 2, 3, 4), 5)*; will print *overload2*
- *OL(Map('k1', 1, 'k2', 2), 4)*; will print *overload3*
- *OL(Map('k1', 1, 'k2', 2, 'k3', 3), 5)*; will print *overload4*

3.7 UI Induction

The *GUI* module is responsible of providing the logic to transform a generic *JSON data* in a *Native GUI Model*. The key concepts to understand the UI Induction are:

- *JSON data*. Any *JSON data structure*.

- JSON UI model .A particular JSON format representing a User Interface.
- Native UI model. A User Interface model ready to be shown by the native GUI system.

3.7.1 Renderization flow

The usage of the *Renderize()* operation can be internally subdivided in three distinct steps. In the first step, a generic *JSON* data is converted by the *Modelize()* operation to a *JSON UI Model*. This model is a tree model representing a default User Interface built on top of the data model, by applying a set of predefined transformations. In the second step the *JSON UI Model* generated at step two can be customized using the *Apply()* operation. In the last step the *JSON UI Model* can be converted in a *UI Native Model* by using the *Concretize()* operation.

The user can choice to use the *Renderize()* operation as is, or by replacing it with the *Modelize()* / [*Apply()*] *Concretize()* operations. The renderization flow is shown in picture 2.

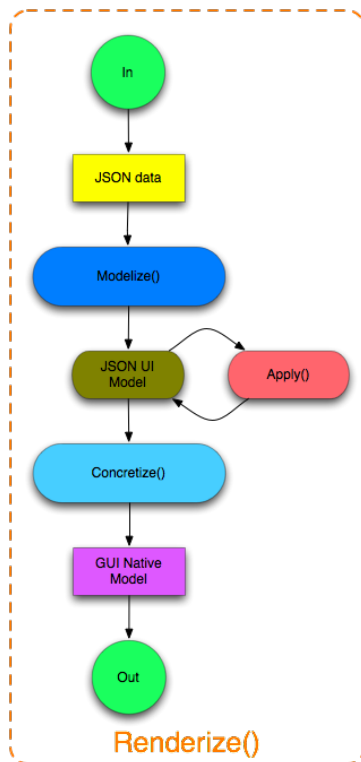


Figure 2: Renderization Flow

3.7.2 Renderization flow and Predicate Overloading

The predicate overloading is important not only for the role it plays in a functional-like language such as *MU*, but also for its suitability for the data presentation. The overloading can be used in combination with the *Apply()* operation to customize a *UI Model*.

The following example shows how predicate overloading is used in order to control the presentation of a data structure:

```

ChangeBG({background : bg{o}}) :
  Add(o, Map(background, red));
  
```

The defined *ChangeBG* (change background) predicate is able to recognize any *JSON* object containing a *background* key. When the matching happens, the predicate simply changes the matched object key by setting its value to 'red'.

Given a *UI Model* instance *uiModel*, it is possible to apply recursively the *ChangeBG* predicate on every model element using the construct:

```

Apply('ChangeBG', uiModel);
  
```

3.7.3 UI Interactivity

The current implementation of *MUP* supports the generation of non interactive interfaces. This means that the current version of the language can be used to generate only static content.

This limitation will be overcome with a solution, still under analysis, called *Live Model*, that will be implemented with subsequent improvements of the framework, as explained below.

As seen previously, every *JSON UI Model* can be generated by a *JSON data*, that is the data model.

The first improvement concerns the introduction of action handlers in the *JSON UI Model*. This will allow to execute *MU* code in response of user actions on the interface. In this phase it is still not possible to change dynamically the UI.

The second improvement concerns the possibility of creating a *1 to 1* relation between the *JSON data* elements and the *JSON UI Model* elements.

The third improvement concerns the ability of every *JSON UI Model* element to listen for changes of the corresponding *JSON data* element. With this last improvement, will be possible to modify the original data model in response of user interactions with the UI (for example by pressing a button), and obtain an automatic update of the UI content.

4. AN EFFECTIVE USECASE

In the following section two simple, but meaningful scenarios are described. The main aim of this description is to provide a readable demonstration of the main peculiarities of the *MU* language.

4.1 A simple processing flow

4.1.1 Description

This scenario follows a canonical mashup process, starting from the data source and arriving at the rendered widget. The goal of this example application is to produce a widget that lists the titles of the item contained in a RSS feed accessed through the HTTP protocol. Reading the flow diagram in figure 3 from top to bottom, it is possible to see how the different functional sets are involved in the different phases of mashup processing. Inside the rectangular box on the right of the functional set circles it has been added a coherent code snapshot example.

- The first step is the raw data retrieval, it can be done with an operation belonging to the *Source Operations* functional set. The code sample performs an *HTTP*

GET request on the specified URL and returns the retrieved XML content of an RSS feed.

- The second step performs the decomposition of the raw data (the retrieved XML) in sub parts. This is done with the help of the *Inpsct Operations* functional set. The code sample performs an *XPath* like operation to extract all the *RSS* item descriptions.
- The third step creates a complex *JSON* data using the operations of the *Model Operations* functional set. The code sample creates an object which keys are the string "entry" and the values are the description strings retrieved from the *RSS* feed.
- The last step creates a *UI* for the given *JSON* data, this is achieved by using the *Renderize Operations* functional set. The code sample shows the usage of the *Renderize()* operation, which provides a default induction of the *UI*.

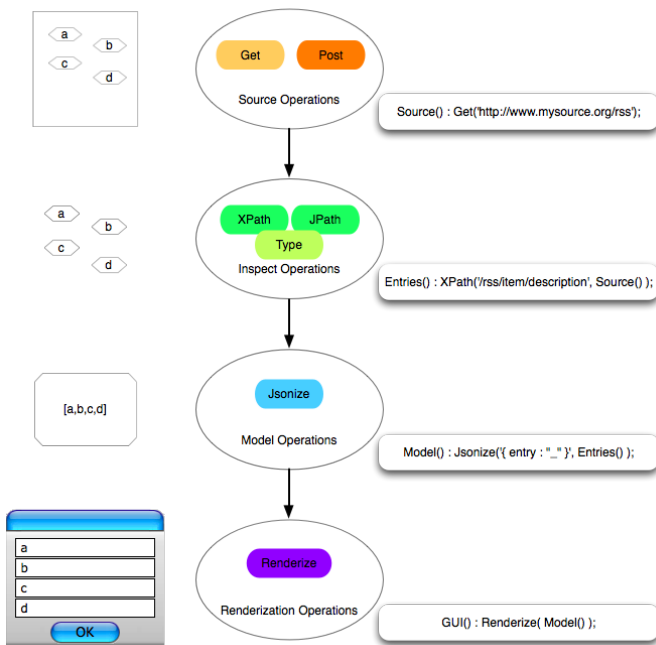


Figure 3: Simple processing flow

4.1.2 Formalization

Follows the *MU* program, where the numbers at the start of each line are not part of the code listing:

```
1 Feed() : PGet("http://www.repubblica.it/rss/rss2.0.xml", List());
2 Titles() : XPath( 'title', Feed() );
3 JsonTitle(t) : Jsonize('{ "title" : %s }', List(t));
4 JsonTitles() : for( Titles(), title, JsonTitle(title) );
5 Renderize( JsonTitles() );
```

where:

- Line 1** The *Feed()* predicate is defined as a list. Every element of the list contains a *JSON* object representation for every item in the *RSS* feed. The *PGet* function is used to define the access method to the feed.

- Line 2** The *Titles()* predicate is defined. It extracts all the 'title' nodes found in the result of *Feed()*.

- Line 3** A *JSON* template is iterated over a string. The result will be something like: "title" : t where t is the given parameter.

- Line 4** This iteration over all available titles generates a list of *JSON* objects, like: ["title" : t1, "title" : t2... "title" : tn] where [t1, ...,tn] is the result of *Titles()*.

- Line 5** The final *JSON* model is rendered. Since the call to the *Renderize()* function is done without parameters the resulting widget is rendered using the *MU* default rules.

4.1.3 Result

In Figure 4 is shown the widget resulting from the mashup.

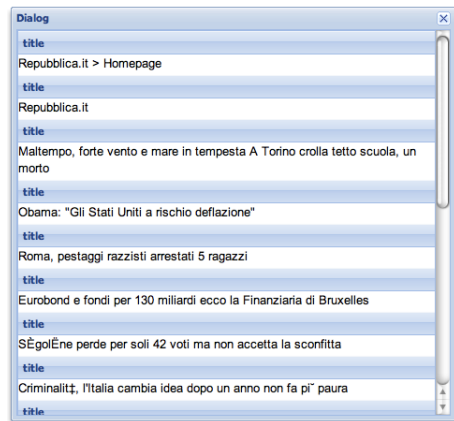


Figure 4: A simple resulting widget

4.2 A complex processing flow

Due the complexity of this scenario, it has been implemented on synthetic data sources, so the data retrieved by the data sources are auto generated in function of the input parameters.

4.2.1 Description

Main aim of this scenario is to demonstrate a mashup of two different data sources. The first data source is a Web service S1 returning a list of restaurants in a specified city, providing for each restaurant name the street in which it is located.

The second data source is a Web service S2 returning a list of hotels near a given street.

The goal of this scenario is to combine these sources to obtain a list of restaurants in a given city with a sublist of hotels near each restaurant.

The expected data format for the first source (S1) can be represented with the following table, where < city > is expanded with the input parameter specifying the name of the city.

restaurant name	restaurant street
RN1<city>	RS1<city>
RN2<city>	RS2<city>

If we invoke the service *S1* for city : *Rome* we will obtain an HTML table with two rows representing two results. Every row is a couple Restaurant Name (RN) - Restaurant Street (RS). The first result will be *RN1Rome*, *RS1Rome* that means: we've found the restaurant *RN1Rome* at street *RS1Rome* and the second result will be *RN2Rome*, *RS2Rome* that means: we've found the restaurant *RN2Rome* at street *RS2Rome*.

The service *S2* takes two parameters, *city* and *street*. The expected data format for this data source can be represented with the following table, where *X* is expanded with the concatenation of the input parameters.

Hotel name	Stars
HN1<X>	stars1<X>
HN2<X>	stars2<X>
HN3<X>	stars3<X>

If we invoke the service *S2* for street: *S1* we will obtain an HTML table of two columns representing the hotel name found in the result record and the number of hotel stars. The table will contain two results. Every result is a different hotel.

The data source *S1* represents the query done at row 01 while the data source *S2* represents the query done at row 06.

4.2.2 Formalization

The previous problem can be formalized with the following statements.

```
#01# S1(city) : Get('http://fake.findrestaurants.dom/find?city=_' ,
                  city);
#02# RawRomeRestaurants() : S1('Rome');
#03# ListOfRomeRestaurants() :
      Path( 'html/body/table/tr' , RawRomeRestaurants() );
#04# RestaurantName(row) : XPath( 'td[0]' , row );
#05# RestaurantStreet(row) : XPath( 'td[1]' , row );
#06# S2(city,street) :
      Get('http://fake.findhotels.dom/find?city=_street=_' ,
          List(city, street) );
#07# HotelsInRomeAtStreet(street) : S2('Rome' , street);
#08# ListOfStreetHotels(street) :
      XPath( 'html/body/table/tr' ,
             HotelsInRomeAtStreet(street) );
#09# HotelName(row) : XPath( 'td[0]' , row );
#10# HotelStars(row) : XPath( 'td[1]' , row );
#11# JsonHotel(row) :
      Jsonize( '{ hotel-name : %s, hotel-stars : %s }' ,
              List( HotelName(row) , HotelStars(row) ) );
#12# HotelsInStreet(street) :
      for( ListOfStreetHotels(street) , hotel , JsonHotel(hotel) );
#13# Restaurant(row) :
      Jsonize(
        '{ restaurant-name : %s, restaurant-street : %s,
          hotels : %a }' ,
        List(RestaurantName(row) , street=RestaurantStreet(row) ,
             HotelsInStreet(street))
      );
#14# RomeRestaurants() :
      for( ListOfRomeRestaurants() , row , Restaurant(row) );
```

Follows the description of every single row.

The row 01 defines the predicate *S1* that, taken the *city* parameter queries the web page <http://fake.findrestaurants.dom/> and returns an HTML page. This page contains a table with the list of restaurants in the specified city.

The row 02 simply defines a predicate that wraps the one defined at 01 and returns an HTML page with the list of restaurants in *Rome*. The expected result will be:

```
<html>
...
<body>
...
<table>
  <tr><td>RN1Rome</td><td>RS1Rome</td></tr>
  <tr><td>RN2Rome</td><td>RS2Rome</td></tr>
</table>
...
</body>
...
</html>
```

The row 03 extracts every single table row from the previous data, returning a list like: [*" <tr ><td > RN1Rome </td ><td > RS1Rome </td ></tr >"*, *" <tr ><td > RN2Rome </td ><td > RS2Rome </td ></tr >"*]

The row 04 is able to extract the first column of a given table row, while the row 05 is able to extract the second column.

The row 06 defines the predicate *S2* that, taken the *city* and *street* parameters queries the Web page <http://fake.findhotels.dom/find> and returns a result like:

```
<html>
...
<body>
...
<table>
  <tr><td>HN1RomeRS1Rome</td><td> SR1RomeRS1Rome </td></tr>
  <tr><td> HN2RomeRS1Rome </td><td> SR2RomeRS1Rome </td></tr>
  <tr><td> HN3RomeRS1Rome </td><td> HN3RomeRS1Rome </td></tr>
</table>
...
</body>
...
</html>
```

The row 07 defines a predicate wrapping the predicate *S1* that finds hotels in *Rome* at a given street.

The row 08 defines a predicate returning in this specific case the list

```
[
  ' <tr><td>HN1RomeRS1Rome</td><td> SR1RomeRS1Rome </td></tr>' ,
  ' <tr><td> HN2RomeRS1Rome </td><td> SR2RomeRS1Rome </td></tr>' ,
  ' <tr><td> HN3RomeRS1Rome </td><td> HN3RomeRS1Rome </td></tr>'
]
```

The row 09 is equals to the row 04.

The row 10 is equals to the row 05.

The row 11 creates a JSON object representing an Hotel from a table row.

The row 12 creates a JSON object representing a list of Hotels in a given street.

The row 13 defines a predicate that takes a table row representing a restaurant and returns a JSON array where every element is a JSON object (see the result) reporting the restaurant name, the restaurant street and a list of hotels in that street with respective number of stars.

The row 14 defines a predicate that invokes the predicate defined at row 13 for every restaurant in Rome, returning the final result.

4.2.3 Result

The result of the last statement is the JSON model listed below:

```
[
  {
    "restaurant-name" : "RN1Rome",
    "restaurant-street" : "RS1Rome",
    "hotels" : [
      { "hotel-name" : "HN1RomeRS1Rome",
        "hotel-stars" : "SR1RomeRS1Rome" },
      { "hotel-name" : "HN2RomeRS1Rome",
        "hotel-stars" : "SR2RomeRS1Rome" },
      { "hotel-name" : "HN3RomeRS1Rome",
        "hotel-stars" : "SR3RomeRS1Rome" }
    ]
  },
  {
    "restaurant-name" : "RN2Rome",
    "restaurant-street" : "RS2Rome",
    "hotels" : [
      { "hotel-name" : "HN1RomeRS2Rome",
        "hotel-stars" : "SR1RomeRS2Rome" },
      { "hotel-name" : "HN2RomeRS2Rome",
        "hotel-stars" : "SR2RomeRS2Rome" },
      { "hotel-name" : "HN3RomeRS2Rome",
        "hotel-stars" : "SR3RomeRS2Rome" }
    ]
  },
  {
    "restaurant-name" : "RN3Rome",
    "restaurant-street" : "RS3Rome",
    "hotels" : [
      { "hotel-name" : "HN1RomeRS3Rome",
        "hotel-stars" : "SR1RomeRS3Rome" },
      { "hotel-name" : "HN2RomeRS3Rome",
        "hotel-stars" : "SR2RomeRS3Rome" },
      { "hotel-name" : "HN3RomeRS3Rome",
        "hotel-stars" : "SR3RomeRS3Rome" }
    ]
  }
]
```

5. RELATED WORK

This section discusses the figure 5, comparing the *MU* language with similar solutions. The columns define the relevant features of a mashup language, chosen in function of the key concepts at the base of *MU*.

1. Javascript Based if the solution is written in Javascript.
2. Java Based if the solution is written in Java.
3. Extensible if the language can be enriched with new constructs.
4. Client Side if the solution runs in a web browser.
5. Server Side if the solution runs in a remote server.
6. Open Source if the framework sources are available as Open Source code.
7. Cross Browser if the framework can be used across the main web browsers (Mozilla, IE, Safari, Opera).
8. Logical if the language supports logical paradigm.
9. Unification Support if the language supports unification.
10. JSON Model Support if the language supports JSON model syntax.
11. Graph Support if the language supports Graph types and operators.
12. Type Morphing if the language supports the type morphing paradigm.

13. UI Induction if the language supports operators for UI Induction (see section 3.7).

The rows enlist products and solutions compared with *MUP*.

1. Afrous⁵ is an online mashup development platform that can be used from your web browser, without anything else. You can mashup not only the Internet web services, but also intranet contents.
2. Yahoo Pipes⁶ is a powerful composition tool to aggregate, manipulate, and mashup content from around the web.
3. Mash Maker⁷ is a browser extension that learns what information you are interested in and automatically creates personalized mashups with content and visualizations from other sources on the web, and brings it all together in one place.
4. OpenKapow⁸ is an open service platform, this means that you can build your own services (called robots) and run them from the openkapow.com server.
5. DSL Sharable Sharable Code⁹ is a platform for building, sharing, and managing Web 2.0 API mashups. The platform allows creation of a common structure for Web 2.0 mashups.

6. CONCLUSIONS

Em'up is a maturing technology, where some aspects are consolidated, such as the core aspects of the language, while others need to find a crystallization (like the UI Model induction logic). At the time of writing we are considering to move more effort into the development of the *JRE* profile while waiting for GWT to become more reliable. The *MUP* team is following a tight development roadmap, reported in the next section. An early preview of the framework code is under release.

7. ROADMAP

These are the features scheduled for the next development stage, regarding both the language and the *em'up* framework.

- Add native support for *JSON* storing and querying. The *MU* internal data model is based on *JSON*. The target of this requirement is to realize an interface to a *JSON* storage providing basic *CRUD* (Create Read Update and Delete), able to persist any type of data.
- Empower the *JSON Model* syntax with *regexp* operators. The *JSON Model* will accept regular expressions where now it is possible to specify string constants.
- Implement a command line console for the *JRE* profile. An *MU* interactive command line will be implemented.

⁵<http://afrous.com/>

⁶<http://pipes.yahoo.com/>

⁷<http://mashmaker.intel.com/>

⁸<http://openkapow.com/>

⁹<http://services.alphaworks.ibm.com/isc/>

Solution	Javascript Based	Java Based	Extensible	Client Side	Server Side	Open Source	Cross Browser	Logical	Unification supp.	JSON Model supp.	Graph supp.	Type Morphing	UI Induction
MU	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Afrous	Yes	No	No	Yes	No	Yes	Yes	No	No	No	No	No	No
Yahoo Pipes	No	No	No	No	Yes	No	Yes	No	No	No	No	No	No
Mash Maker	Yes	No	No	Yes	No	No	No	Yes	No	No	No	No	No
OpenKapow	No	Yes	Yes	No	Yes	No	Yes	No	No	No	No	No	No
DSL Sharable Sharable Code	No	No	No	No	Yes	No	Yes	No	No	No	No	No	No

Figure 5: Comparison Table

- Complete the XPath support (some operators are missing).
 - Completion the JSONPath support (some operators are missing).
 - Empower of the MU syntax with the integration of the JSONPath and the XPath syntaxes. It will be possible to use the XPath and JSONPath constructs as sub syntaxes of the MU syntax. For example it will be possible to write something like: $PredA().p1.p2.p3[i]$ to access the result of predicate $PredA$ as JSON and extract the addressed sub part, and $PredB/p1/p2/p3$ to access the result of predicate $PredB$ as string (expected to be XML) and extract the the addressed sub part.
 - Add support for interactive UI Models.
 - Add support for parallelization and distribution. It is under analysis the infrastructure for parallelizing a MU application and distributing it across different processor nodes.
- [8] C. Wikstrom. Distributed programming in erlang, 1994.
- [9] N. Zang and M. B. Rosson. What’s in a mashup? and why? studying the perceptions of web-active end users. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, pages 31–38, 2008.
- [10] N. Zang, M. B. Rosson, and V. Nasser. Mashups: who? what? why? In *CHI ’08: CHI ’08 extended abstracts on Human factors in computing systems*, pages 3171–3176, New York, NY, USA, 2008. ACM.

8. RESOURCES

The project code and documentation can be found at location <http://code.google.com/p/em-up/>.

9. REFERENCES

- [1] S. Antoy and M. Hanus. Functional logic design patterns. pages 67–87. 2002.
- [2] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web (ldow2008). In *WWW ’08: Proceeding of the 17th international conference on World Wide Web*, pages 1265–1266, New York, NY, USA, 2008. ACM.
- [3] R. J. Ennals and M. N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1116–1118, New York, NY, USA, 2007. ACM.
- [4] M. Franklin, A. Halevy, and D. Maier. From databases to dataspace: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.
- [5] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.
- [6] K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, March 1989.
- [7] S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.