

## A Spreadsheet-like Construct for Streamlining and Reusing Mashups

Guiling Wang<sup>1</sup>, Shaohua Yang<sup>1,2</sup>, Yanbo Han<sup>1</sup>

<sup>1</sup> *Research Center for Grid and Service Computing,  
Institute of Computing Technology, Chinese Academy of Sciences,*

<sup>2</sup> *Graduate University of Chinese Academy of Sciences  
Beijing, 100190, P.R. China*

*{wangguiling, yangshaohua}@software.ict.ac.cn, yhan@ict.ac.cn*

### Abstract

*It is challenging to provide end users an easy-to-use problem-solving tool to combine data from different sources and reuse the results. Inspired by spreadsheets, we argue that spreadsheet-like programming paradigm can help to reduce the complexity and to improve user experience in building mashups. In this paper, we propose a spreadsheet-like construct as the basis of this mashup building paradigm. The construct includes a data model, a “nested table” view structure and a set of carefully chosen mashup operators. Data from a variety of sources is structured like a spreadsheet, and end-users are not necessarily aware of the underlining data flow. SpiderCharlotte, a tool to help end users to build situational applications for their daily uses, was developed to demonstrate the characteristics of this construct.*

**Keywords:** *Situational applications, mashups, nested table, end-user programming*

### 1. Introduction

There is a tremendous amount of information available on web sites as HTML pages, in the enterprise back-end databases or in the enterprise information systems as web services. In many cases, it is quite valuable to integrate information from various sources. In general, the information integrated applications are built by IT developers to meet long-lived requirements. However, it is impossible to anticipate and build all of the integrated applications that will be required in the future. Sometimes, the integrated applications should be built on-the-fly to solve transient and ad-hoc business problems (so-called “*Situational Applications*” [1]). Such applications need to combine data from a variety of

sources directly by business users (usually end users) without any involvement of IT developers in a “just-in-time” way. At the same time, we note that there has recently been lots of interest in so called “mashup” and “mashup editor” where a mashup is a web application that combines data from more than one source into a single integrated application, and a mashup editor is a WYSIWYG tool providing a visual interface to build mashups [2]. Situational applications are, in fact, kind of mashup applications except that they might combine data not only from the Web but also from enterprise sources, such as back-end databases and internal information systems.

Today, more and more mashup editors are becoming available. Probably the best-known mashup editor is Yahoo Pipes [3], which helps users to create flows of services and RSS feeds, and generate new feeds for sharing. Besides Yahoo Pipes, applications like Microsoft Popfly [4], IBM Damia [5], Intel MashMaker [6] and CMU Marmite [7] are all such mashup editors.

The new breed of mashup editors demonstrated that simple approaches such as RSS feeds, RESTful services and AJAX technologies can lower the barrier of building situational applications. However, it is still very difficult for average end users to combine and integrate data from a variety of sources, especially when facing some specific but ad-hoc goals. On the one hand, mashup editors need to be powerful enough because combining data from different sources involves both modeling and computation tasks. On the other hand, mashup editors need to be easy-to-use, for the target users include a large number of users who may only have little programming skills. However, most mashup editors have not achieved a good balance between “powerfulness” and “easiness-for-use”. It may demand a lot of time and skill for end users to create service (or data) flows using mashup editors like Yahoo Pipes, IBM Damia and Marmite. Furthermore,

most of the mashup editors focus on the consumption of RSS feeds and RESTful services, whereas HTML pages, the majority of Web-based data sources, can't be imported without the help of external data extraction tools. Although some mashup editors can import web pages as ready-to-use data sources, they require extensive knowledge like HTML and regular expression, which is beyond the ability of average end users.

To balance "powerfulness" and "easiness-for-use", this paper presents a spreadsheet-like construct for streamlining and reusing mashups. The construct includes a data model, a "nested table" view structure and a set of carefully chosen mashup operators. It offers a lightweight information integration approach that is easy enough for end users to combine and integrate data from different sources by themselves.

Spreadsheets have achieved remarkable success in allowing non-programmers to represent complex data and configure certain computations. One of the key advantages of spreadsheets is their "low entry barrier" programming paradigm. It requires little time and skill before end users are rewarded by simple but functioning programs that model their problems of interest. As noted, spreadsheets suggest that "a limited set of carefully chosen, high-level, task-specific operations and a strong visual format for structuring and presenting data are key characteristics for user programming environment" [8]. If mashup editors inherit the key characteristics of spreadsheets such as low entry barrier, mixing values and expressions, carefully chosen operations and strong visual format, it can help end-users greatly in reducing the complexity and improve user experience in building mashups. Users can build and reuse mashups without knowledge of data flow modeling.

The main contribution of this paper is to propose a novel spreadsheet-like construct for building mashups. In order to make spreadsheets programming paradigm more suitable for mashup editors, we introduce several changes to spreadsheets paradigm including the user interface, data model and function operations. We propose a data model that is simpler than XML schema but powerful enough for representing general web-based data and services, and adopt the nested table as the powerful view structure. In addition, we define a set of carefully chosen mashup operators over the data model. Using the mashup operators, users can import a service as a table, drag and drop a column onto another for merging, execute a service on current data through expressing a formula, etc.

Based on the spreadsheet-like construct, we have implemented a prototype called *SpiderCharlotte*, which enables end users to build situational applications themselves. *SpiderCharlotte* is seamlessly integrated with *Grubber* (the detail of *Grubber* will be introduced in another paper), a tool enabling end users to build RESTful services from Web-based data sources. So it is more convenient to combine HTML pages than other related work.

The paper is organized in the following way: Section 2 describes the spreadsheet-like mashup construct from three aspects: user interface, data model and view structure, mashup operators. Section 3 discusses related work and Section 4 presents the demonstration scenario. Finally, Section 5 concludes the paper and introduces our future study.

## 2. A Spreadsheet-like Construct

This section presents in detail the spreadsheet-like construct for streamlining and reusing mashups. We first give a brief introduction of the user interface design of *SpiderCharlotte* for a better understanding of the construct. Then we define the data model and its view structure. Finally, we describe a set of operators defined over the data model.

### 2.1 User Interface

*SpiderCharlotte* consists of three parts: (1) a web server hosting a community of different forms of data sources, such as RSS feeds, RESTful services, SOAP-based services; (2) *Canvas*, an extension of Firefox web browser, which helps end users to combine data sources in a drag-and-drop manner; (3) *Grubber*, another extension of Firefox web browser for generating RESTful services from HTML pages. In distribution, *Canvas* and *Grubber* are packaged into one extension. The user interface of *SpiderCharlotte* contains three areas: a resource list pane, a spreadsheet pane (the major working area of *SpiderCharlotte*), and a data flow pane (see Figure 1). The user can drag data sources from the resource list pane and drop them to the spreadsheet pane for combination. Sometimes, he/she may need to assign parameters for the selected data sources. The combination results (or data) are displayed as a nested table (section 2.2). Users can click on a column's label area to modify/add/delete the column, drag and drop a recommended service onto a blank column or drag and drop a column onto another

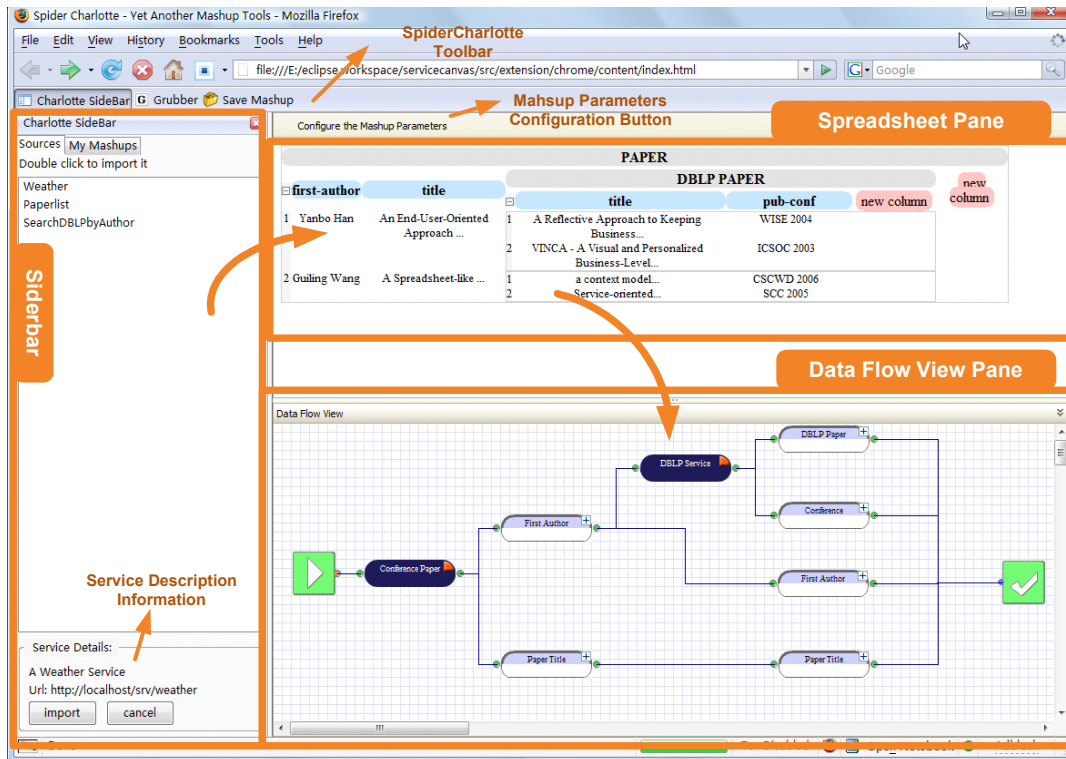


Figure 1. SpiderCharlotte User Interface

column for merging. While users are operating on the spreadsheet pane, the operation sequences are recorded and displayed on the data flow pane. Users can browse the automatically generated data flow. This lets the user remember the history of operations and understand the current mashup results. (Our data flow view is currently read-only, in that end-users cannot click on a node to rollback the operations or configure the parameters of a node to debug the mashup process. These features may be added in our later version.)

## 2.2 Data Model and View Structure

In this section, we define a data model for describing the data structures of the data sources that can be accessed from the resource list pane.

The data on the spreadsheet pane is any set of data values conforming to a common *type*. A data value of a specific type is called an *instance* of the type. In this study, we consider the following *type*:

- *Atomic type*: the basic type that can't be subdivided it into other types represents simple data values like strings and numbers. An instance of an atomic type can only assume atomic values.
- *Tuple type*: If  $T_1, \dots, T_n$  are types, then  $\langle T_1, \dots, T_n \rangle$  is also a type constructed from  $T_1, \dots, T_n$  using a tuple constructor.

- *List type*: If  $T$  is a type, then  $\{T\}$  is also a type constructed from  $T$  using a set constructor. An instance of a list type is an ordered set of instances of the same type.

Suppose a user is browsing the accepted paper list web page on a conference website. The user may be interested in some papers and want to know more about the background information. So it can help a lot to present the accepted paper list together with the publication history from DBLP website. Here is an example of such a composite type combining the accepted paper information and DBLP publications history of the same author:

$$PAPER: \langle firstAuthor, title, \{DBLP: \langle titlename, pubconf \rangle\} \rangle \quad (1)$$

This tuple type, *PAPER*, is composed of two atomic types (*firstAuthor* and *title*) and one list type defined over a tuple type *DBLP*. *PAPER* describes the structure of the accepted papers of an academic conference. *DBLP* describes the structure of the list of publications by the first author from DBLP database. An instance of *PAPER* is a data value as follows:

$$PAPER: \langle Guiling Wang, A spreadsheet-centric programming..., \{ DBLP: \langle Modeling context..., CSCWD 2006 \rangle, DBLP: \langle Service-oriented..., SCC 2005 \rangle \} \rangle$$

There are two kinds of data sources in our construct. One are those sources imported as a new nested table. The other are those sources that can be applied on an existed nested table as computational operators, which will be introduced in section 2.3. Here we define the first kind of source as a tuple  $s = \langle name, desc, uri, params, schema \rangle$ , where *name* is the source service's method name, *params* is the source service's input parameters, *schema* is the data schema of the source service's output represented as a set of *types* defined above. Our previous work on Web-based data extraction and service encapsulation has studies how to generate a source service from HTML pages [9]. For SOAP services, *params* and *schema* are described in the associated WSDL. For REST, RSS/Atom services, the definition can be derived from the services' documentation or example request/response messages by programmers.

Though the type based data model defined above are powerful enough for representing most of Web-based data sources, an adequate visual structure for facilitating mashing up and presenting data by the end-users has yet not been proposed. The tabular grid structure of Spreadsheet enlightens us to adopt the nested table as the core visual structure for making mashups in our work.

A list type defined as  $NT\{\langle T_1, \dots, T_n \rangle\}$  is a nested table schema, where *NT* is the name of the nested table,  $T_i$  is an atomic type, a list type defined over an atomic type or a table schema. Figure 2 shows the nested table representation of the above example. A nested table always has an equivalent tree view. Every instance of a type has a certain level property, a parent instance (except the root instance), sibling instances and child instances (except the leaves). In fact, we implement the above data model based on a multi sub-tree data structure.

However, spreadsheet programming paradigm also has some disadvantages. For example, with the size of operation sequences growing, a spreadsheet becomes more difficult for end users to comprehend and rollback mashing up process, because the order of operations and data dependencies are invisible to the end users. Therefore, the data flow view is adopted as an assistant structure to make up for these disadvantages. As users become more familiar with the editor, they can learn how to configure the data flow and perform more complicated operations. For example, the advanced users can browse the operation sequences and data dependencies, enter new parameter values, or rollback their operations.

### 2.3 Mashup Operators

Before we define the necessary operators in our construct, we have the following conventions to simplify the maintenance of data dependencies:

<i>PAPER</i>			
<i>firstAuthor</i>	<i>title</i>	<i>DBLP</i>	
		<i>title</i>	<i>pubConf</i>
<i>Yanbo Han</i>	<i>An End-User-Oriented Approach...</i>	<i>A Reflective Approach to Keeping Business...</i>	<i>WISE 2004</i>
		<i>VINCA - A Visual and Personalized Business-Level ...</i>	<i>ICSOC 2003</i>
		...	...
<i>Guiling Wang</i>	<i>A spreadsheet-centric programming ...</i>	<i>Modeling context...</i>	<i>CSCWD 2006</i>
		<i>Service-oriented ...</i>	<i>SCC 2005</i>
		...	...

Figure 2. Nested table representing instances of *PAPER* according to (1)

1. *Services that can be imported or used as operators are read-only services.* We only consider read-only services currently.
2. *Type is the atomic variable in our construct.* Users can't perform any operation on a cell independently. This is unlike the typical spreadsheet in which cells are the variables.
3. *For a nested table schema  $NT\{\langle T_1, \dots, T_n \rangle\}$ , an instance of the tuple  $\langle T_1, \dots, T_n \rangle$  or  $T_i$  is the integral and inseparable computation element.* For example, suppose an user want to merge a type  $T_i$  from table  $NT\{\langle T_1, \dots, T_n \rangle\}$  with a type  $T_i'$  from the other table  $NT'\{\langle T_1', \dots, T_n' \rangle\}$ . If we only move the instances of  $T_i$  from *NT* to *NT'* without bringing the instances of other types  $T_j$ , the integrity of instance of tuple  $\langle T_1, \dots, T_n \rangle$  will be destroyed. Therefore, when users perform the merge operation on two types, all of their parent types are merged one by one.
4. *Only types of the same level can be merged.* It is always meaningless to merge two types of different levels.

With these conventions indicated above, we select and define the following necessary operators:

- *import operator:* If a source service is modeled as a function  $f(X)=Y$ , where  $X=\{X_1, \dots, X_m\}$  denotes the input parameters,  $Y=\{Y_1, \dots, Y_n\}$  denotes the output parameters, *import* can be defined as a function  $import(sourceid, X)=Y$ , where *sourceid* is the ID of the source service.

- *service operator*: If an operation service is modeled as a function  $g(X)=Y$ , *service operator* can be defined as a function  $s(serviceid, X, mapping)$ , where *mapping* is responsible to transform the function  $g$  into a new function  $g':g \rightarrow g'(T)=Y$ . As a result,  $X$  is mapped to  $T$ , a set of types in current nested table  $NT$ . When a user enters the related function  $g(X)$  of a service operation in a blank column's label area and construct *mapping* in an interactive way, the service is invoked for each instance of  $NT$ . After the operation service is executed, the result  $Y$  is added into  $NT$ .
- *deleteType operator*: It is defined as a function  $deleteType(T)$ , where  $T$  can be an atomic type, list type defined over an atomic type or a table schema.
- *mergeType operator*: It is defined as a function  $mergeType(T_1, T_2)$ , where  $T_1, T_2$  are atomic types, list types defined over an atomic type or a table schema. If  $T_1, T_2$  are list types or a table schema, a new list type is directly generated. For example,  $T_1 = \{<A, B>\}$ ,  $T_2 = \{<A', B'>\}$ , the merged type will be generated with its schema as  $T = \{<A, B, A', B'>\}$ . If  $A=A'$  and  $B=B'$ , the merged type will be generated with its schema as  $T = \{<A, B>\}$ . If  $T_1, T_2$  are atomic types, the merged type is an atomic type  $T$ .
- *fuseType operator*: It is defined as a function  $fuseType(T_1, T_2)$ , where  $T_1, T_2$  can only be atomic types. This operation not only merges  $T_1, T_2$  as an atomic type  $T$  (the same semantics as *mergeType*), but also merges the instances of  $T$  (the same semantics as *mergeInstance*).
- *mergeInstance operator*: It is defined as a function  $mergeInstance(T)$ , where  $T$  can only be an atomic type. This operation merges the instances with the same value of  $T$  as one instance.
- *filter operator*: It is defined as a function  $filter(T, conditions)$ , where  $T$  can only be an atomic type, *conditions* is a group of condition expressions.
- *sort operator*: It is defined as a function  $sort(T, order)$ , where  $T$  can only be an atomic type, *order* indicates which order the instances are sorted.
- *sink operator*: It is defined as a function  $sink(option)$ , where *option* indicates which format the user want to export the current mashup as.

With the operators defined as above, a data mashup is defined as a tuple  $m = \langle name, desc, params, script \rangle$ , where *params* is the parameters of the mashup and *script* is the operation sequences.

### 3. Related Work

In this section, firstly we discuss the existing mashup editors and their constructs. Secondly, we present the advantages of our spreadsheet-like construct. The mashup editors and their constructs generally fall into one of three categories: 1). **flow-centric mashup editors and data-flow construct** where data and services are processed in a manner similar to Unix pipes or workflow. Yahoo Pipes, Microsoft Popfly, IBM Damia and Marmite fall into this category. The flow-centric mashup editors often adopt a data flow view structure and a set of workflow-like mashup operators. 2). **spreadsheet-centric mashup editors and flat spreadsheet construct** where data and services are processed in a manner similar to spreadsheets. They often adopt a flat spreadsheet construct. C3W [10] falls into this category. 3). **tree-centric mashup editors and tree-like construct** where users interact with the tool to combine data and services based on a tree view structure. Intel MashMaker falls into this category. The tree-centric mashup editors adopt often a tree-based data model and view structure.

As discussed in Section 1, the concept of data flow is always the main barrier to complete the mashup task. We argue that the data-flow construct is not so good at balancing powerfulness and easiness-for-use for average end users. The spreadsheet-centric editors are very easy for end users to use. But the flat spreadsheet construct can't process and present the complex data in a nested structure. Though the tree-centric editors overcome the disadvantages of the spreadsheet-centric editors by supporting more complex data, it is not as convenience as spreadsheets to define computation operations conducted on several nodes. Based on the above analysis, the spreadsheet-like construct goes beyond the other three categories. Besides this, the ability to clip any web form to form cells on a spreadsheet may feature largely in C3W. But unlike C3W, we first extract web pages as services and import services to form blocks of cells on a spreadsheet-like pane, which is also convenient for users.

### 4. Demonstration Scenario

We plan to show the interesting features of this spreadsheet-like construct through a demonstration scenario. In this scenario, the user is planning to attend an international conference, and wants to know the accepted papers and also the related information about the authors. The user also wants to combine the information together as an integrated view.

First, the user opens the "accept papers" web page on the conference's web site. Then the user clicks on

“discover source service” icon on his/her browser toolbar. Now, *SpiderCharlotte* starts up *Grubber* to help the user build a RESTful service. The user can extract this web page as a “source service” by highlighting the authors and paper title. Second, the user imports this source service into the work area. Third, the user clicks on the blank column area in the “paper list” table, selects a DBLP service from the services list, and maps the DBLP service’s “author” parameter to the “first author” column in current “paper list” table. Now the user can get an updated nested table just like Figure 2. At the same time, these imported services and their relations are displayed on the “data flow view pane”. Finally, the user can save this mashup as a mashup service.

As a result, the next time the user wants to access the updated paper lists and the related information about the authors, all the user need do is open this mashup from the “mashup list”. The user need not open the conference web page, the DBLP web page and merge the information manually. He/she also need not repeat the above mashup process.

## 5. Conclusions and Future Directions

The paper presented a spreadsheet-like construct for enabling end users to build situational applications by combining Web-based data and web services. We adopt a data model simpler than XML schema but powerful enough for representing general Web-based data and services. The nested table is adopted as the view structure more powerful than general spreadsheets. A set of carefully chosen mashup operators based on the data model are demonstrated. We also introduce the user interface featured by the central spreadsheet view with the data flow view as assistance.

In our future work, we will not only focus on the implementation and evaluation of *SpiderCharlotte*, but also plan to adapt it to a situational applications environment in a specific domain (e.g., bioinformatics experiment). Currently the state-of-the-art means to construct the information integration systems are services composition and other integration technologies. However, it is difficult for classical services composition solution to solve problem under the situations where business requirements are non-deterministic and can’t be pre-defined. It is the case that users can’t pre-define the whole process. Considering the flexibility of our spreadsheet-like construct, we are inclined to believe that it can also be adapted in our problem solving environment(i.e., VINCA4Science [11]) to enhance the abilities of end users (e.g., the bioinformatics scientist) to build their own situational applications.

## Acknowledgments

The work is supported by the National Natural Science Foundation of China under Grant No.60573117, the National Basic Research Program of China under Grant No.2007CB310804 and Chinese Academy of Sciences Olympic2008 Project (Grand: KACX1-03).

## References

- [1] L. Cherbakov, A. Bravery and A. Pandya. SOA meets situational applications, Part 1: Changing computing in the enterprise. <http://www-128.ibm.com/developerworks/webservices/library/ws-soa-situational1/>. 2007
- [2] Wikipedia, <http://en.wikipedia.org/wiki/>, 2008.
- [3] Yahoo Pipes, Inc. <http://pipes.yahoo.com/>, 2008.
- [4] Microsoft Popfly, <http://www.popfly.com/>, 2008
- [5] QEDWiki, <http://services.alphaworks.ibm.com/qedwiki/>, 2007
- [6] R. Ennals and D. Gay. User-friendly functional programming for web mashups, In Proceedings of the 12th International Conference on Functional Programming (ICFP), Freiburg, Germany, 2007, 223-234.
- [7] J. Wong and J. I. Hong, Making mashups with marmite: towards end-user programming for the web, In Proc. of the 2007 conference on Human factors in computing systems (CHI), San Jose, California, USA, 2007, 1435-1444.
- [8] B. A. Nardi,; J. R. Miller, The Spreadsheet Interface: A Basis for End User Programming, HP Labs Technical Reports, <http://www.hpl.hp.com/techreports/90/HPL-90-08.pdf>, 1990
- [9] S.H. Yang, H.L. Lin and Y.B. Han. Automatic data extraction from template-generated Web pages. *Journal of Software*. 19(2): 209-223, 2008.
- [10] J. Fujima, A. Lunzer, K. Hornbaek, etc. Clip, connect, clone: combining application elements to build custom interfaces for information access, In Proc. of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST), 2004, 175-184.
- [11] Jing Wang, Yanbo Han, Shuying Yan, Wanghu Chen, Guang Ji .: VINCA4Science: A Personal Workflow System for e-Science. The 3rd IEEE International Conference on Internet Computing for Science and Engineering, Jan.28~29, 2008.