Spreadsheet as a Generic Purpose Mashup Development Environment

Dat Dac Hoang[1], Hye-Young Paik[1] and Anne H. H. Ngu[2]

[1]School of Computer Science & Engineering, University of New South Wales, Sydney
[2]Department of Computer Science, Texas State University, San Marcos
{ddhoang, hpaik}@cse.unsw.edu.au, angu@txstate.edu

Abstract. Mashup development is done using purposely created tools. Because each tool offers a different paradigm and syntax for wiring mashup components, users need to learn different tools for different tasks. We believe that there is a need for a generic purpose mashup environment catering for a wider range of mashup applications. In this paper we introduce MashSheet - a spreadsheet-based, generic purpose mashup tool. Using MashSheet, mashups can be built using spreadsheet paradigms that many users are already familiar with. We use a generic data model (XML-based) to represent mashup components and data produced by them, which enables the reuse of intermediate mashup results. We support three classes of mashup operations: data, process and visualization.

1 Introduction

Mashup is a new application development method enabling users to create applications by reusing existing contents and functionalities. Recently, we have witnessed rapidly growing interests in mashup tools and applications in the web community. In programmableweb.com, for example, there are over 4815 mashup applications and 1995 Web APIs registered[1]. However, many mashup programmings today are done in "special-purpose" and "hand-crafted" manners using a purposely created development environment [8]. For example, MapCruncher [1] is designed for users to create mashups involving maps. Swivel.com makes it easy for users to create graph mashups from multiple data tables. Yahoo Pipes [5] is best suited to generate feeds. Due to the ad-hoc nature of the popular mashups tools, users are required to learn different tools, paradigms and syntaxes to write mashup applications. We believe that there is a need for a generic purpose mashup environment that can cater for a wider range of mashup applications with a uniform mashup paradigm.

In this paper, we propose a spreadsheet-based mashup programming framework called MashSheet. Spreadsheets are frequently used to analyze data imported from different sources (e.g, database, file, data service), particularly in the context of decision support. They also have been tagged as the most successful end-user development environment with millions of users [11, 16].

We believe that spreadsheets can provide a programming paradigm that many users are already familiar with, hence would be a good environment for designing a generic

[1] The figures are as of 18 May 2010, from [2]

purpose mashup tool. In fact, there have been some efforts in spreadsheet-based mashup tools (e.g., [4, 18, 14, 12]) with varying degree of success [10]. There is still much work left to do in order to create a spreadsheet-based mashup framework which can be generically applied to multiple situations. We use these existing systems to benchmark our prototype to show that our tool is more widely applicable for mashup programming.

1.1 Reference scenario

Let us consider the following scenario as our running example. Tom wants to create a mashup application that will help him to find Points of Interest (POI) (e.g., restaurant, cinema) when he travels to a city. Tom considers the following five services:

- `RL`: with `read`() operation, it returns a list of restaurants with addresses,
- `CL`: with `read`() operation, it returns a list of cinemas with addresses,
- `DG`: with `getDR`() operation, it returns car driving direction,
- `WF`: with `getWeather`() operation, it returns weather forecast information,
- `BU`: with `getBU`() operation, it returns bus itinerary.

Some services require SOAP-based interactions and some are RSS feeds (Figure 1(b)). Figure 1(a) shows the scenario of the application. Tom inputs the address of a hotel, calls `RL` and `CL`, merges and sorts the results. He would use `DG` only when the `WF` service reports rain, otherwise he prefers to use public transport, so information from `BU` will suffice. Finally, Tom visualizes the direction results in a grid of cells.

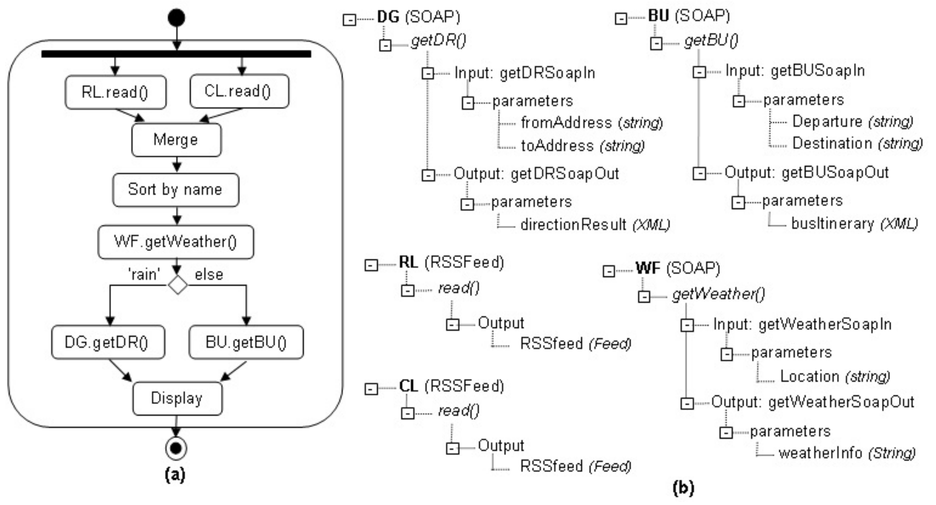


Fig. 1. Tom's scenario

1.2 Contributions

To make the tool generic, we start with a generic data model that will support a wide-range of data types we need to deal with in the framework. We name the data type 'MashSheet Object Type' (MOT) and it is an XML-based data model. MOT is based on [12], where the conventional spreadsheet data model is extended to include complex data types to represent RSS feeds. Using MOT, we introduce two components to represent a web service (`service`) and web service output (`service-output`). At present, MashSheet can work with SOAP-based services and RSS data feeds. However, it is not difficult to extend the range in the future, e.g., REST-based services.

MashSheet offers the following advantages over existing approaches:

- MashSheet defines all operations as spreadsheet formula: data, process and visualization. This means that the mashup operations would be like another spreadsheet formula to the users - the concept many users are already familiar with.
- MashSheet provides different classes of mashup operations to make the framework applicable to many mashup scenarios.
  - *Data operators:* In spreadsheet, the data is represented as simple data types (e.g., string, number). Web service invocation, on the other hand, often returns complex data type such as XML document. In other spreadsheet-based mashup tools, users have to "flatten" the complex data into two-dimensional grid of spreadsheet before data operations (e.g., sort, filter, join) can be applied. This creates unnecessary step in mashup creation process. In MashSheet, users can manipulate the complex data "as-is" by applying data mashup operation directly on MOT. In addition, any intermediary result created by data operations can be reused by other data and visualization mashup operators at any stage. This increases the reusability of intermediate data in the application.
  - *Process operators:* Evaluation of spreadsheet's formula is driven by data dependencies between cells. This data-flow model allows users to define some process mashup operations using natural spreadsheet programming metaphor (e.g., sequence by using cell referencing mechanism).
    However, the semantics of some control flow patterns are not inherently supported in spreadsheets and none of the existing spreadsheet-based mashup tools address this issue (e.g., exclusive choice, synchronization). Supporting a basic set of control flow patterns are important for mashup component composing scenarios. We introduce an extension to the spreadsheet formula language so that the basic control flow patterns [17] are supported.
  - *Visualization operators.* Data visualization needs in mashup cover a wide range of options (e.g., grid, map, chart, timeline). In current spreadsheet-based mashup tools we investigated, data can only represented in a grid of cells by using either simple grid (i.e., column, row) [14, 4] or nested grid (i.e., hierarchical, index, repeater) [12, 18]. Also, the data view needs to reflect any change in the data source and make it immediately visible to users.
    We define visualization operators as components to present service output data using different visualization types (e.g., grid, map). The benefits of having these operators are: (i) *flexibility*: the layout of data is automatically updated when there is a change in the data source); (ii) *generic applicability*: we can

apply different visualization operators to produce different layouts of data; (iii) *easiness to use*: can be called like spreadsheet functions.

The remainder of this paper is organized as follows. The related work is discussed in section 2. We introduce the architecture of MashSheet in section 3. Sections 4 and 5 explain the application model and operators of MashSheet. Implementation, mashup formula evaluation mechanism and a brief evaluation are discussed in sections 6 and 7. Section 8 presents conclusion and future work.

2 Overview of spreadsheet-based mashup tools

According to Fischer et al. [7], spreadsheet is one of the six programming paradigms for mashup development, along with integrated development environment, scripting languages, wiring paradigm, programming by demonstration and automatic creation of mashup. In this section, we present mashup tools that use spreadsheet paradigm[2].

StrikeIron [4] is a Microsoft Excel plugin allowing users to pull data from SOAP web services into Excel worksheet. Using StrikeIron, users can create mashup applications by 'wiring' output data of one service with the input parameter of another service (via cell referencing). However the main limitation of this approach is that it does not support all of the basic control flow patterns. In addition, data visualization in StrikeIron is limited to conventional spreadsheet data visualization (i.e., data is visualized by a grid of cells and a cell can only accommodate simple types such as number, string).

AMICO:CALC [14] is an OpenOffice Calc extension that lets users to create mashups within Calc spreadsheet by using AMICO-Read and Amico-Write functions. The execution of mashup application is based on a middleware named Adaptable Multi-Interface COmmunicator. By using a combination of Read and Write functions, users can model basic control flow patterns in their mashup application. However, AMICO:CALC does not provide data manipulation and visualization operators.

Mashroom [18] is a mashup tool that relies on nested table algebra and spreadsheet programming metaphor to simplify the mashup creation process. However, Mashroom is not a generic mashup framework since it focuses only on data mashup and its data visualization is limited to nested table.

In SpreadMash [12], web data is modeled using entity relations, then browsed and visualized using widgets. SpreadMash, however, is limited to data importation and visualization. It does not mention control flow and is not a generic mashup tool. Spreadator [15], as SpreadMash's earlier prototype, mainly provides means for accessing/manipulating web data from spreadsheet. MashSheet builds further on the early ideas presented in SpreadMash and Spreadator, focusing on generic applicability and mashup component composition. We will discuss these tools further in section 7.

3 MashSheet Overview

This section sketches the overall design of MashSheet and explains how it draws inspiration from spreadsheet. As depicted in Figure 2, the architecture of MashSheet comprises of four elements:

[2] Please refer to [10] for more detailed analysis.

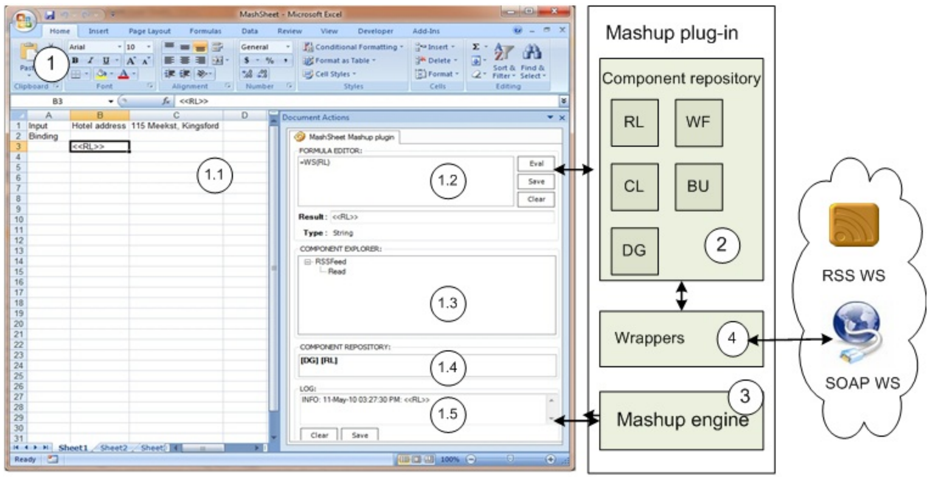


Fig. 2. The architecture of MashSheet.

– **MashSheet's graphical user interface (GUI) (1).** This element is a conventional spreadsheet interface which includes a grid of cells (1.1) and a mashup plugin implemented as a Task Pane. Cells are capable of storing values and formulas, while mashup plugin allows users to build mashup applications by specifying composition logic and layout information. The GUI plays a role as a development environment and users mainly interact with this element. The mashup plugin includes:
  - *Formula editor (1.2)*. This element allows users to enter mashup formulas. Note that our formula input area is different with the conventional formula input in spreadsheets. The reason for this design is that most of the spreadsheets do not allow developer to modify their evaluation engine with conventional input.
  - *Component explorer (1.3)*. This element lets users to visualize structure of web service (e.g., WSDL) and service output (e.g., XML) in a tree view.
  - *Component repository (1.4)*. This element shows to the users a list of components (i.e., representations of web services in the tool). It is a front-end of "Component repository" element which will be defined below.
  - *Log (1.5)*. This element displays the output message of each operation.
– **Component Repository (2).** This element is a repository of all mashup components available in the tool. Users can choose an external web service and create a component that allows them to interact with the service within the tool. This service component can be assigned a friendly name (i.e., alias) and can be called in the MashSheet formula editor. For example, in the Tom's scenario, `DG` is a component representing a SOAP-based service that provides driving direction information.
– **Mashup Engine (3).** This key element is responsible for evaluating the mashup formula (i.e., composition logic) and "wiring" components together. Since most of spreadsheet tools do not allow users to modify their formula evaluation mechanisms, mashup engine could be developed as an extension to spreadsheet evaluation engine. Mashup engine is also responsible for maintaining the formula eval-

uation context and facilitate the reaction to cell modification by triggering the re-evaluation of dependent formula (i.e., upon a service invocation returns, the corresponding references need to be updated with the returning value). We will explain the working of the engine with examples in the later sections.

– **Wrappers (4).** Wrappers facilitate interoperability among services which have different data formats (e.g., XML, RSS) or use different access protocols (e.g., SOAP, RSS). For example, we need different wrappers to create components to correctly serve the Tom's scenario, such as SOAP wrapper for DG, RSS wrapper for RL. We rely on existing works (e.g., RSS.NET library [3]) to provide wrappers for services.

4 MashSheet application model

MashSheet extends conventional spreadsheet to provide its application model. We use the formal definitions of spreadsheet application in [6] to describe our model as follow:

A MashSheet application ($\mathtt{S}$) is a spreadsheet contains a collection of cells organized in a tabular grid. Each cell is identified by its address ($\mathtt{a}$), value ($\mathtt{v}$) and formula ($\mathtt{f}$). An address uses absolute coordinates of the cell in the grid (e.g., $\mathtt{A1}$). A formula can contain a value ($\mathtt{v}$) (e.g., a number), a reference to another cell address ($\mathtt{a}$) (e.g., $\mathtt{A1}$) or mashup operators (ω) (e.g., $\mathtt{invoke}()$). A cell's value is obtained by evaluating the cell's formula. In MashSheet, a value ($\mathtt{v}$) can be either a simple data type (α) (e.g., number, string) or complex data type. We model the complex type as XML-based data type and name it MashSheet Object Type (MOT). Using MOT, we build two components: $\mathtt{service}$ ($\mathtt{SC}$) and $\mathtt{service} - \mathtt{output}$ ($\mathtt{OC}$) to represent web service and its output data, respectively. Figure 3 illustrates the MashSheet's model. We use the notion $\|(\mathtt{a},\mathtt{f})\|_{\mathtt{S}}$ to

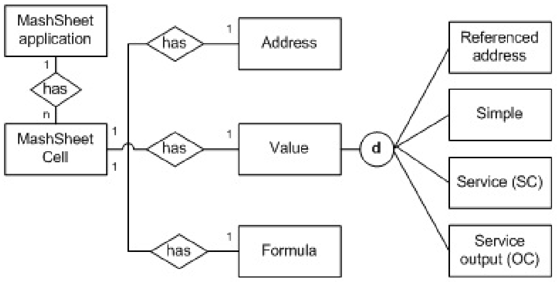


Fig. 3. MashSheet application model

denote the evaluation of a MashSheet cell with address $\mathtt{a}$ and formula $\mathtt{f}$. Cell reference can use single cell address (e.g., $\mathtt{A1}$) or a range of cells (e.g., $\mathtt{A1:C3}$).

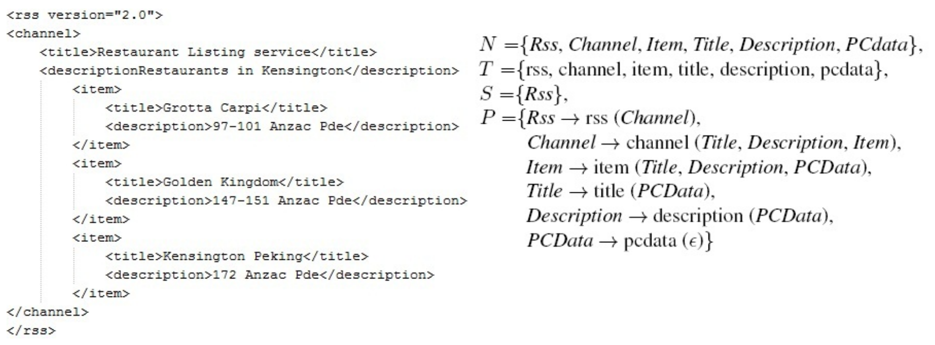
We use regular tree languages and tree automata from [13] to define MOT:

MashSheet Object Type (MOT) is a quadruples $(\mathtt{N},\mathtt{T},\mathtt{B},\mathtt{P})$, where:

– $\mathtt{N}$ is a finite set of nonterminals which are regular expressions represented by capitalized $Italic$ characters (ϵ is a null sequence of nonterminals),

– $\mathtt{T}$ is a set of terminals which are symbols represented by lowercase characters ($pcdata$ is a special terminal character matched with any node in the tree),
– $\mathtt{B}$ is a set of start symbols, where $\mathtt{B} \in \mathtt{N}$,
– $\mathtt{P}$ is a set of rules in the form of $\mathtt{X} \rightarrow \mathbf{a}r$, where $\mathtt{X} \in \mathtt{N}$, $\mathbf{a} \in \mathtt{T}$, and r is a regular expression over $\mathtt{N}$.

Figure 4 shows an example of data produced by RL's `read()` operation and its MOT representation (for visibility issue, we remove some fields such as Link, PubDate).



```
<rss version="2.0">
<channel>
    <title>Restaurant Listing service</title>
    <descriptionRestaurants in Kensington</description>
        <item>
            <title>Grotta Carpi</title>
            <description>97-101 Anzac Pde</description>
        </item>
        <item>
            <title>Golden Kingdom</title>
            <description>147-151 Anzac Pde</description>
        </item>
        <item>
            <title>Kensington Peking</title>
            <description>172 Anzac Pde</description>
        </item>
</channel>
</rss>
```

$N = \{Rss, Channel, Item, Title, Description, PCdata\}$,
$T = \{\text{rss, channel, item, title, description, pcdata}\}$,
$S = \{Rss\}$,
$P = \{Rss \rightarrow \text{rss}\ (Channel)$,
$Channel \rightarrow \text{channel}\ (Title, Description, Item)$,
$Item \rightarrow \text{item}\ (Title, Description, PCData)$,
$Title \rightarrow \text{title}\ (PCData)$,
$Description \rightarrow \text{description}\ (PCData)$,
$PCData \rightarrow \text{pcdata}\ (\epsilon)\}$

Fig. 4. Example of RL data and its MOT representation.

Service component ($\mathtt{SC}$*)* is a triplet $(\mathtt{Name}, \mathtt{Type}, \mathtt{URL})$,where:

– $\mathtt{Name}$ is an unique identifier of a web service ($\mathtt{WS}$) to be used in MashSheet,
– $\mathtt{Type}$ is the type of the $\mathtt{WS}$,
– $\mathtt{URL}$ is unified resource locater of the $\mathtt{WS}$,

When a $\mathtt{SC}$ is bound to a MashSheet cell, MashSheet creates an instance of the $\mathtt{SC}$ in the application. We call the instance of the $\mathtt{SC}$ as '$\mathtt{bound\ SC}$' and denote as $\mathtt{SC}^*$. $\mathtt{SC}^*$ inherits all the attributes of the $\mathtt{SC}$ and has a new attribute: $\mathtt{Location}$ which is the address ($\mathtt{a}$) of the $\mathtt{SC}$ in MashSheet.

Service-output component ($\mathtt{OC}$*)* is a triplet $(\mathtt{Name}, \mathtt{OprCell}, \mathtt{OutputCell})$, where:

– $\mathtt{Name}$ is an unique identifier of a $\mathtt{service - output}$ component,
– $\mathtt{OprCell}$ is the address ($\mathtt{a}$) of the $\mathtt{SC}$,
– $\mathtt{OutputCell}$ is the address ($\mathtt{a}$) of the $\mathtt{OC}$.

For example, consider a simple MashSheet application as follow:

$\mathtt{create(`http://www.ecubicle.net/driving.asmx?WSDL', `DG')}$
$\mathtt{(A1, = bind(`DG'));}$
$\mathtt{(B2, = invoke(A1.getDR(C1, C2)))}$

The first formula creates $\mathtt{SC_{DG}} = \{\mathtt{DG}, \mathtt{SOAP}, \mathtt{http://www.ecubicle.net/driving.asmx?WSDL}\}$ in the component repository. Evaluation of the formula in cell $\mathtt{A1}$ creates: $\mathtt{SC^*_{DG}} = \{\mathtt{DG}$, $\mathtt{SOAP}$, $\mathtt{http://www.ecubicle.net/driving.asmx?WSDL}$, $\mathtt{A1}\}$ and evaluation of the formula in cell $\mathtt{B2}$ creates: $\mathtt{OC_{DG}} = \{\mathtt{DG}, \mathtt{A1}, \mathtt{B2}\}$.

5 MashSheet's mashup operators

MashSheet offers operators that support data, process (orchestration of services) and visualization operations in a mashup application. They are used in MashSheet formula and can be classified into four main categories: *life cycle*, *process*, *data* and *visualization*. In this section, we briefly introduce these operators.

5.1 Life cycle operators

MashSheet has three life cycle operators for the purpose of managing $\mathtt{SC}$ and $\mathtt{SC}^*$ in the application: component creation, component deletion, service binding.

Component creation: $\mathtt{create}()$ registers a new $\mathtt{SC}$ to the component repository by importing the service interface referenced. This operator does not produce any data value in the MashSheet grid. For example, the following formula creates $\mathtt{SC}_{\mathtt{DG}}$:

$$= \mathtt{create}(\text{‘}\mathtt{http://www.ecubicle.net/driving.asmx?WSDL}\text{’}, \text{‘}\mathtt{DG}\text{’})$$

Component deletion: $\mathtt{delete}()$ removes the referenced $\mathtt{SC}$ from the component repository. Similar to $\mathtt{create}()$, $\mathtt{delete}()$ does not affect the MashSheet's cell values since it is repository management operator. For example, the following formula removes $\mathtt{SC}_{\mathtt{DG}}$ from the repository:

$$= \mathtt{delete}(\text{‘}\mathtt{DG}\text{’})$$

Component binding: $\mathtt{bind}()$ binds a $\mathtt{SC}$ to a specific cell and creates $\mathtt{SC}^*$. When the $\mathtt{SC}^*$ is created, cell address ($\mathtt{a}$) is used to identify the component and its operations are accessible by using dot notation. The incentive of using binding operator is that users can interact with a specific web service invocation within cells. For example, evaluation of the following cell binds $\mathtt{SC}_{\mathtt{DG}}$ to cell $\mathtt{B3}$ and creates ($\mathtt{SC}^*_{\mathtt{DG}}$):

$$(\mathtt{B3}, = \mathtt{bind}(\text{‘}\mathtt{DG}\text{’}))$$

Thereafter, Tom can refer to the operation of $\mathtt{DG}$ as $\mathtt{B3.getDR(String, String)}$. $\mathtt{bind}()$ has one input parameter which is $\mathtt{SC}$ and returns $\mathtt{SC}^*$. That is, it can be expressed as $\|(\mathtt{a}, \mathtt{bind}(\mathtt{SC}))\|_{\mathtt{S}} \rightarrow \mathtt{SC}^*$.

5.2 Process operators

MashSheet has three process operators for the purpose of supporting orchestration of web services: simple invocation, extended IF and synchronization. They are used to support modeling basic control flow patterns [17] in the application.

Simple invocation: $\mathtt{invoke}()$ is defined much of the same way as other spreadsheet-based mashup tools, however, in MashSheet, the output is held in a *OC* for further access and manipulation. For example, evaluation of the following cell invokes $\mathtt{getDR}()$ operation provided by $\mathtt{SC}^*_{\mathtt{DG}}$ in cell $\mathtt{B3}$ and holds the output data ($\mathtt{OC}_{\mathtt{DG}}$) in cell $\mathtt{B5}$:

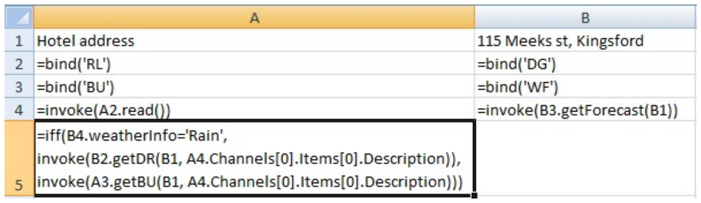
$$(\mathtt{B5}, = \mathtt{invoke}(\mathtt{B3.getDR}(\mathtt{B1}, \mathtt{B13})))$$

$\mathtt{invoke}()$ has one parameter which is $\mathtt{SC}^*$ and returns $\mathtt{OC}$. That is, it would be expressed as $\|(\mathtt{a}, \mathtt{invoke}(\mathtt{SC}^*))\|_{\mathtt{S}} \rightarrow \mathtt{OC}$.

Extended if: `iff()` is provided to model an exclusive choice pattern in conventional spreadsheet. Consider the exclusive choice case in our reference scenario. The pseudo code is represented as: `If WF` =‘`rain`’ `Then DG Else BU`. The semantically correct interpretation of this situation is that there is only one execution between `DG` and `BU`. But in conventional spreadsheets (and any other spreadsheet mashup tools we investigated), both `DG` and `BU` will run in order for the code/formula can be evaluated.
`iff()` has a condition and two $\mathtt{SC}^*$ parameters and returns `OC`. It would be expressed as $\|(\mathtt{a}, \mathtt{iff}(\mathtt{condition}, \mathtt{SC}^*, \mathtt{SC}^*))\|_{\mathtt{S}} \rightarrow \mathtt{OC}$. The semantic of `iff()` is defined as follow:

$$\|(a, iff(condition, opr_1, opr_2)\|_S = \begin{cases} \|(a, opr_1)\|_S & \text{if } condition = True \\ \|(a, opr_2)\|_S & \text{if } condition = False \end{cases}$$

For example, Figure 5 illustrates an exclusive choice scenario.



	A	B
1	Hotel address	115 Meeks st, Kingsford
2	=bind('RL')	=bind('DG')
3	=bind('BU')	=bind('WF')
4	=invoke(A2.read())	=invoke(B3.getForecast(B1))
5	=iff(B4.weatherInfo='Rain', invoke(B2.getDR(B1, A4.Channels[0].Items[0].Description)), invoke(A3.getBU(B1, A4.Channels[0].Items[0].Description)))	

Fig. 5. `iff()` example: User input is in cell `B1`; Service bindings are in cells `A2`:`B3`; Simple invocations are in cells `A4`:`B4`; Exclusive choice is in cell `A5`.

Synchronization: `sync()` is provided to correctly support the semantic of AND-join (synchronization) and XOR-join (simple merge) in spreadsheet [17]. It has parameters which are $\mathtt{SC}^*$ and a condition indicating the semantic of `sync()` (i.e., AND/XOR) and returns a Boolean value (α) indicating the joining status of participating service invocations: $\|(\mathtt{a}, \mathtt{sync}(\mathtt{SC}^*, .., \mathtt{SC}^*, \mathtt{condition}))\|_{\mathtt{S}} \rightarrow \alpha$. The semantic of AND-join is defined as follow:

$$\|(a, sync(opr_1, .., opr_n, AND)\|_S = \begin{cases} False & \text{if any } \|(a, opr_k)\|_S \,|k \in (1, n) \text{ has not evaluated} \\ True & \text{if all } \|(a, opr_k)\|_S \,|k \in (1, n) \text{ has evaluated} \end{cases}$$

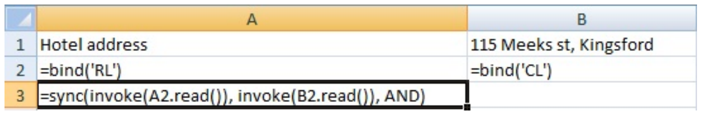
The semantic of XOR-join is defined as follow:

$$\|(a, sync(opr_1, .., opr_n, XOR)\|_S = \begin{cases} False & \text{if no } \|(a, opr_k)\|_S \,|k \in (1, n) \text{ has evaluated} \\ True & \text{if any } \|(a, opr_k)\|_S \,|k \in (1, n) \text{ has evaluated} \end{cases}$$

For example, Figure 6 illustrates a synchronization scenario.

5.3 Data operators

MashSheet provides eight data operators: `union()`, `join()`, `merge()`, `merge_field()`, `rename_field()`, `filter()`, `filter_field()` and `sort()` for manipulating with `OC` data. The main differences of these operators in comparison with data mashup operators in other spreadsheet-based mashup tools are:



	A	B
1	Hotel address	115 Meeks st, Kingsford
2	=bind('RL')	=bind('CL')
3	=sync(invoke(A2.read()), invoke(B2.read()), AND)	

Fig. 6. `sync()` example: User input is in cell `B1`; Service bindings are in cells `A2:B2`; `sync()` is in cell `A3`. Users check the value of `A3` to see whether two invocations are synchronized or not.

- Input parameters of MashSheet data operators are XML data. Therefore, instead of mapping data into the grid before can run the operator (like in other tools), MashSheet data operators can run directly on the XML data. The operators can perform operations either on the structure of data (e.g., `merge()`, `filter_field()`) or on the data itself (e.g., `filter()`, `sort()`).
- Instead of immediately visualized in cells, the output of a data mashup operator are held in a cell for further processing by other data or visualization operators. These operators are motivated by the fact that intermediate data in a mashup application should be reused by others operators and visualized only when needed.

In this section, due to the space limitation, we only introduce the `merge()` operator. Full description of all operators is given in [9].

Merge: `merge()` combines two `OC`s in a uniform structure to an `OC` containing all the data from participating `OC`s. Evaluation of the cell containing `merge()` is defined as: $\|(\mathtt{a}, \mathtt{merge}(\mathtt{OC}, \mathtt{OC})\|_{\mathtt{S}} \rightarrow \mathtt{OC}$. We denote the first `OC` as $\mathtt{OS}_1(\mathtt{N}_1, \mathtt{T}_1, \mathtt{B}_1, \mathtt{P}_1)$ and the second `OC` as $\mathtt{OC}_2(\mathtt{N}_2, \mathtt{T}_2, \mathtt{B}_2, \mathtt{P}_2)$. The `merge()` operator runs on $\mathtt{OC}_1, \mathtt{OC}_2$ will produce a new $\mathtt{OC}(\mathtt{N}, \mathtt{T}, \mathtt{B}, \mathtt{P})$ with MOT representation as follow:

$$\mathtt{N} = \mathtt{N}_1 = \mathtt{N}_2, \mathtt{T} = \mathtt{T}_1 = \mathtt{T}_2, \mathtt{B} = \mathtt{B}_1 = \mathtt{B}_2, \mathtt{P} = \mathtt{P}_1 = \mathtt{P}_2$$

For example, consider the following cells. The formula in cell `C3` merges the output data produced by `RL`'s `read()` and `CL`'s `read()` operations in cell `A2` and `B2`.

$(\mathtt{A1}, = \mathtt{bind}(\text{‘}\mathtt{RL}\text{’})); (\mathtt{B1}, = \mathtt{bind}(\text{‘}\mathtt{CL}\text{’}));$
$(\mathtt{A2}, = \mathtt{invoke}(\mathtt{A1.read}())); (\mathtt{B2}, = \mathtt{invoke}(\mathtt{B1.read}()));$
$(\mathtt{C3}, = \mathtt{merge}(\mathtt{A2}, \mathtt{B2}))$

5.4 Visualization operators

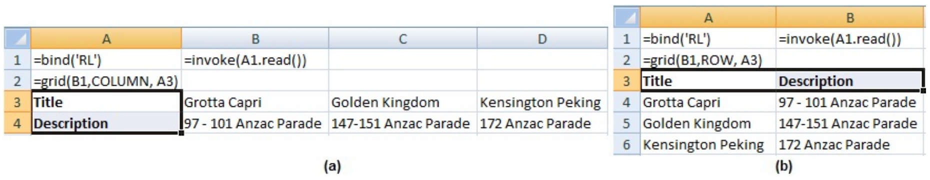
The goal of these operators is to separate the presentation layer from data layer so that users can visualize `OC` data using different layouts (e.g., grid, map). The separation is an important issue, especially when users want flexibility and reusability in creating their own display of `OC` data. Basically, visualization operators map a data space into a visual space. In our context, data space is `OC` and visual space is a set of cell addresses (`A`).

MashSheet visualization operator is a triplet $\mathtt{O} = (\mathtt{D}, \mathtt{V}, \mathtt{R})$, where:

- `D` is an `OC`, $\mathtt{OC} = (\mathtt{N}, \mathtt{T}, \mathtt{B}, \mathtt{P})$
- `V` is a visual space including a set of cells addresses `A`
- `R` is a set of visualization rules, a rule $\mathtt{r} \in \mathtt{R}$ is a mapping $\mathtt{r} : \mathtt{N} \rightarrow \mathtt{V}$

MashSheet contains two visualization operators: `grid()` and `map()`:

Grid: $\texttt{grid}()$ allows users to visualize $\texttt{OC}$ data in a grid of cells. Evaluation of the cell containing $\texttt{grid}()$ is defined as: $\|(\texttt{a}, \texttt{grid}(\texttt{OC}, \alpha, \texttt{a})\|_\texttt{S} \rightarrow (\texttt{a}, .., \texttt{a})$. There are two types of $\texttt{grid}()$: $\texttt{column}$ and $\texttt{row}$ identified by parameter α. Column (row) grid visualizes data as a horizontal (vertical) sequence of columns (rows) where a column (row) is constructed as a vertical (horizontal) sequence of attribute in $\texttt{N}^*$ ($\texttt{N}^*$ is a subset of $\texttt{N}$ constructed by selecting all "leaf" nodes in the $\texttt{OC}$'s tree). For example, Figure 7(a) and 7(b) present two scenarios when Tom visualizes the data produced by $\texttt{RL}$'s $\texttt{read}()$ operator in cell $\texttt{B1}$ by a range of cells which the top-left cell is cell $\texttt{A3}$.



(a)

	A	B	C	D
1	=bind('RL')	=invoke(A1.read())		
2	=grid(B1,COLUMN, A3)			
3	Title	Grotta Capri	Golden Kingdom	Kensington Peking
4	Description	97 - 101 Anzac Parade	147-151 Anzac Parade	172 Anzac Parade

(b)

	A	B
1	=bind('RL')	=invoke(A1.read())
2	=grid(B1,ROW, A3)	
3	Title	Description
4	Grotta Capri	97 - 101 Anzac Parade
5	Golden Kingdom	147-151 Anzac Parade
6	Kensington Peking	172 Anzac Parade

Fig. 7. Visualize RL data by (a)column, (b)row

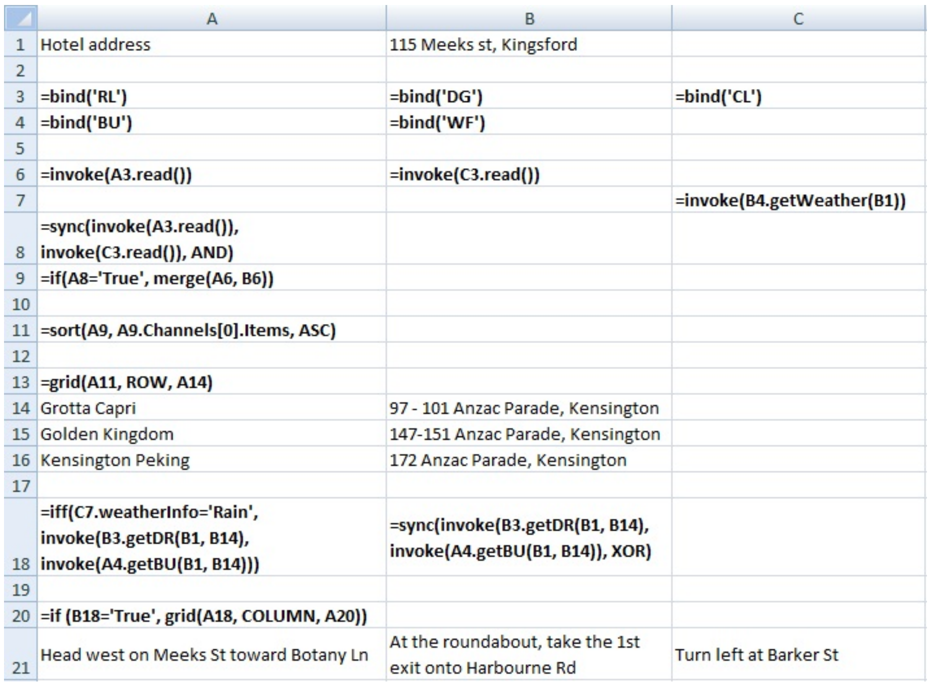
Map: $\texttt{map}()$ lets users to visualize static data in an $\texttt{OC}$ using a map interface. This operator creates a map object with points indicating the addresses and detail information about each point. For implementation purpose, we choose Google API as foundation of map provider. $\texttt{map}()$ does not produce any data value in the MashSheet grid. Instead, it shows a map object as a gadget in the grid interface. We define $\texttt{a}_0$ is a special address representing a floating gadget. Evaluation of the cell containing $\texttt{map}()$ is defined as: $\|(\texttt{a}, \texttt{map}(\texttt{OC})\|_\texttt{S} \rightarrow (\texttt{a}_0)$. $\texttt{map}()$ has two parameters which are addresses of the POIs and information adding to each POI. Users have to manually extract the data for these two parameters from an $\texttt{OC}$. Consider the following cells, evaluation of formula in cell $\texttt{B2}$ will get data produced by $\texttt{RL}$'s $\texttt{read}()$ operation in cell $\texttt{B1}$ and display it in a map object.

$$(\texttt{A1}, = \texttt{bind}(\text{‘}\texttt{RL}\text{’})); (\texttt{B1}, = \texttt{invoke}(\texttt{A1.read}()));$$
$$(\texttt{B2}, = \texttt{map}(\texttt{B1.Channels.Items.Description}, \texttt{B1.Channels.Items.Title}));$$

6 Building and executing MashSheet application

Figure 8 shows how Tom's scenario can be implemented using MashSheet. First, Tom defines cell $\texttt{B1}$ as the input area for entering hotel's address. He uses $\texttt{create}()$ to register five components to the repository (not shown in the figure). He, then, binds five components to cells $\texttt{A3}$:$\texttt{B4}$, $\texttt{C3}$. Tom is now ready to invoke services and build the scenario:

– He invokes the $\texttt{read}()$ operations provided by $\texttt{RL}$ and $\texttt{CL}$ in cells $\texttt{A6}$ and $\texttt{B6}$ respectively, He also invokes $\texttt{getWeather}()$ by $\texttt{WF}$ in $\texttt{C7}$,
– Tom uses a $\texttt{sync}()$ formula in cell $\texttt{A8}$ to check the synchronization status of two $\texttt{read}()$ operations provided by $\texttt{RL}$ and $\texttt{CL}$,



	A	B	C
1	Hotel address	115 Meeks st, Kingsford	
2			
3	=bind('RL')	=bind('DG')	=bind('CL')
4	=bind('BU')	=bind('WF')	
5			
6	=invoke(A3.read())	=invoke(C3.read())	
7			=invoke(B4.getWeather(B1))
8	=sync(invoke(A3.read()), invoke(C3.read()), AND)		
9	=if(A8='True', merge(A6, B6))		
10			
11	=sort(A9, A9.Channels[0].Items, ASC)		
12			
13	=grid(A11, ROW, A14)		
14	Grotta Capri	97 - 101 Anzac Parade, Kensington	
15	Golden Kingdom	147-151 Anzac Parade, Kensington	
16	Kensington Peking	172 Anzac Parade, Kensington	
17			
18	=iff(C7.weatherInfo='Rain', invoke(B3.getDR(B1, B14), invoke(A4.getBU(B1, B14)))	=sync(invoke(B3.getDR(B1, B14), invoke(A4.getBU(B1, B14)), XOR)	
19			
20	=if (B18='True', grid(A18, COLUMN, A20))		
21	Head west on Meeks St toward Botany Ln	At the roundabout, take the 1st exit onto Harbourne Rd	Turn left at Barker St

Fig. 8. A running example implementing Tom's scenario. Note that in this figure we combine formula view and evaluation view in one worksheet due to the space limitation. In fact, the formulas will be evaluated to concrete data in the corresponding cells.

– After both `RL` and `CL` are completed, he merges the data in cell `A6` and `B6` and puts the result in cell `A9`[3],
– He then, sorts the content of `A9` and put the result into `A11`. The sorting is done by the name column (i.e., `A9.Channels[0].Items`) in ascending order.
– In cell `A13`, Tom visualizes the data in cell `A11` by a grid of cells which the top-left cell of the grid is cell `A14`.
– Tom enters the formula in cell `A18` to get the direction from his hotel to the first POI in the list (in case `WF` reports `rain`, he uses `DG` otherwise he uses `BU`).
– Finally, he visualizes the direction produced in cell `A18` in a grid of cells which the top-left cell is cell `A21` by entering a formula in cell `A20`.

To build the MashSheet application, Tom needs to consider two aspects: (i) control flow: defines the order in which the cells' formulas are evaluated and the condition under which a certain formula may or may not be evaluated; (ii) data flow: defines how data is passed between the evaluations of the formulas. We observe that spreadsheet evaluation engine only depends on data flow through cell referencing mechanism. In this framework, we extend the MashSheet language to support both control and data

[3] It is not shown in the figure for space reason, but the evaluation result of each operation is displayed on the component explorer (Figure 2 part 1.3) when associated cells are clicked.

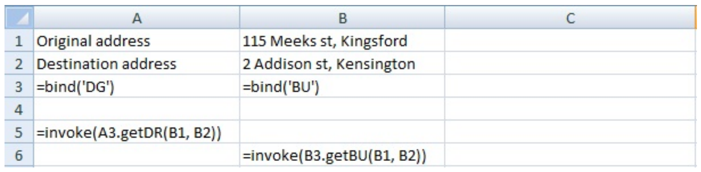
flows in a MashSheet application but still using spreadsheet-like language. We consider the basic control flow patterns as defined by [17], and explain how to model each pattern in the following:

Sequence can be modeled in MashSheet either by spatial arrangement of cells' formulas in a MashSheet application or by using cell referencing (the spatial arrangement of cells is considered first).

– Spatial arrangement: We define the spatial arrangement of the cell formulas as the evaluation order. Two formulas are sequentially evaluated if their cells are located in two *adjacent cells*. The evaluation order progresses from left to right and from top to bottom of the cell locations. For example, the group of cells `A3`: `B4`, `C3` in Figure 8 are sequentially executed in the order:
`A3` $\rightarrow$ `B3` $\rightarrow$ `C3` $\rightarrow$ `A4` $\rightarrow$ `B4`.
– Cell referencing: Two formulas are also considered sequentially evaluated if they have *input-output dependency* to each other. For example, the formula in cell `A6` is evaluated after the evaluation of formula in cell `A3` in Figure 8.

Parallel split can also be modeled either by spatial arrangement of cell's formulas or using cell referencing.

– Spatial arrangement: Two formulas are evaluated in parallel if their cells are located in two *non-adjacent cells without data dependency*. For example, cells `A6` and `C7` in Figure 8 are evaluated in parallel.
– Cell referencing: Two formulas are considered executed in parallel if they have *data dependency with the same data and have no data dependency to each other*. For example, cells `A5` and `B6` in Figure 9 are evaluated in parallel.



	A	B	C
1	Original address	115 Meeks st, Kingsford	
2	Destination address	2 Addison st, Kensington	
3	=bind('DG')	=bind('BU')	
4			
5	=invoke(A3.getDR(B1, B2))		
6		=invoke(B3.getBU(B1, B2))	

Fig. 9. Parallel example with input-output dependency

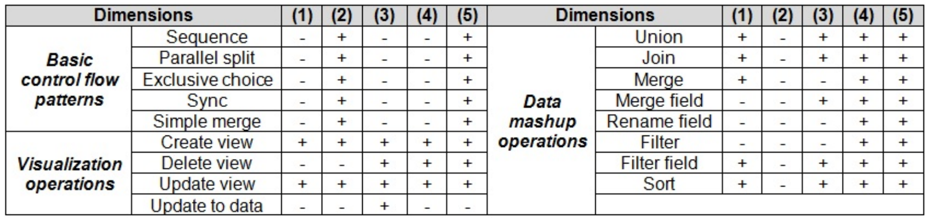
Exclusive choice can be modeled using `iff()` operator. For example, in Figure 8 the formula in cell `A18` models an exclusive choice pattern.

Synchronization and Simple Merge can be modeled using `sync()` operator with `AND` and `XOR` parameters, respectively. For example, the `sync()` formula in `A8` performs *synchronization* for the formulas in `A6` and `B6` are executed in parallel, whereas the `sync()` in `B18` performs *simple merge* because the service invocations in `iff()` is done as *exclusive choice*.

Implementation. MashSheet is implemented as a plug-in of Microsoft Excel application. It extends the work presented in [15], which is mainly designed for presenting data services in spreadsheet. The language of choice for implementation is C# using Visual Studio for Office (VSTO) toolkit and the language for implementing evaluation engine is JScript. MashSheet GUI is implemented as a Task Pane in the Excel. RSS service is accessible by using open source library RSS.NET. We use applications provided in Microsoft .NET SDK to access SOAP services (e.g., wsdl.exe, csc.exe).

7 MashSheet benchmark

In this section we evaluate the generic property of MashSheet against other spreadsheet-based mashup tools. The first group of dimensions examines the support for basic control flow patterns. The second group of dimensions looks at data mashup operation aspects and the last group includes visualization operators. Figure 10 shows the result. StrikeIron, SpreadMash and Mashroom do not support the dimensions in the first



Dimensions		(1)	(2)	(3)	(4)	(5)	Dimensions		(1)	(2)	(3)	(4)	(5)
Basic control flow patterns	Sequence	-	+	-	-	+	***Data mashup operations***	Union	+	-	+	+	+
	Parallel split	-	+	-	-	+		Join	+	-	+	+	+
	Exclusive choice	-	+	-	-	+		Merge	+	-	-	+	+
	Sync	-	+	-	-	+		Merge field	-	-	+	+	+
	Simple merge	-	+	-	-	+		Rename field	-	-	-	+	+
Visualization operations	Create view	+	+	+	+	+		Filter	-	-	-	+	+
	Delete view	-	-	+	+	+		Filter field	+	-	+	+	+
	Update view	+	+	+	+	+		Sort	+	-	+	+	+
	Update to data	-	-	+	-	-							

Fig. 10. Benchmarking MashSheet and reviewed spreadsheet-based mashup tools: (1)StrikeIron, (2)AMICO:CALC, (3)SpreadMash, (4)Mashroom, (5)MashSheet

group since they are data mashup frameworks. SpreadMash supports "update to data" dimension since it allows users to update the change in spreadsheet to the data source. Mashroom supports all data dimensions. AMICO:CALC supports the basic control flow patterns, but it does not provide data mashup operators.

8 Conclusions and future works

In this paper, we introduced MashSheet - a generic purpose mashup framework that allows users to write mashup applications using spreadsheet formulas. In MashSheet, mashup applications can be built incrementally, producing intermediary results during the process. MashSheet has the following benefits: (i) the mashup language is based on a programming paradigm that many users are already familiar with, (ii) it considers not only data flow but also control flow in mashup development, and (iii) it allows flexible data visualization using visualization operators.

One might argue that the user interface for mashup development should be as simple as drag and drop, so that users do not have to write the formulas. However, drag-and-drop manipulations are only suitable for simple scenarios. We believe supporting formula-based scripting is more generic approach, making it applicable to more complex scenarios.

Currently, we are working on several areas to improve out framework. First, we are conducting an evaluation study to attain both qualitative and quantitative feedback on the feasibility of MashSheet as a generic mashup tool. We target the scientific workflow communities for obtaining concrete mashup scenarios for evaluating the tool. This will also lead to extending our range of supported service interface types. An important feature we would like to integrate into MashSheet is reusability. By supporting reuse, a mashup application in MashSheet will be considered as a service component which can be invoked from other applications.

References

1. Mapcruncher. http://research.microsoft.com/en-us/um/redmond/projects/mapcruncher.
2. Programmableweb.com. http://www.programmableweb.com/mashups/directory.
3. Rss.net library. http://www.rssdotnet.com.
4. Strikeiron web services for excel. http://www.strikeiron.com.
5. Yahoo pipes. http://pipes.yahoo.com.
6. Robin Abraham and Martin Erwig. Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN (PPDP '06)*, 2006.
7. Thomas Fischer, Fedor Bakalov, and Andreas Nauerz. An overview of current approaches to mashup generation. In *Wissensmanagement*, pages 254–259, 2009.
8. Dion Hinchcliffe. The 10 top challenges facing enterprise mashups. http://www.zdnet.com/blog/hinchcliffe/the-10-top-challenges-facing-enterprise-mashups.
9. Dat Dac Hoang and Hye young Paik. Spreadsheet as a generic purpose mashup development environment. Technical report, The University of New South Wales, Sydney, Australia, 2010.
10. Dat Dac Hoang, Hye young Paik, and Boualem Benatallah. An analysis of spreadsheet-based services mashup. In *Proceeding of the ADC '10, Brisbane, Australia*, 2010.
11. Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in excel. In *Proceedings of the ICFP '03, Uppsala, Sweden*, 2003.
12. Woralak Kongdenfha, Boualem Benatallah, Régis Saint-Paul, and Fabio Casati. Spreadmash: A spreadsheet-based interactive browsing and analysis tool for data services. In *Proceedings of the CAiSE '08, Montpellier, France*, 2008.
13. Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM T. Internet Tech.*, 5(4):660–704, 2005.
14. Zeljko Obrenovic and Dragan Gasevicc. End-user service computing: Spreadsheets as a service composition tool. *IEEE Transactions on Services Computing*, 2008.
15. Régis Saint-Paul, Boualem Benatallah, and Julien Vayssière. Data services in your spreadsheet! In *Proceedings of EDBT '08 (Demo), Nantes, France*, pages 690–694, 2008.
16. Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the VLHCC '05, Dallas, USA*, 2005.
17. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
18. Guiling Wang, Shaohua Yang, and Yanbo Han. Mashroom: end-user mashup programming using nested tables. In *Proceedings of the WWW '09, Madrid, Spain*, 2009.