Building Mashups By Example

Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock
Information Science Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
pipet@isi.edu, pszekely@isi.edu, and knoblock@isi.edu

ABSTRACT
Creating a Mashup, a web application that integrates data from multiple web sources to provide a unique service, involves solving multiple problems, such as extracting data from multiple web sources, cleaning it, and combining it together. Existing work relies on a widget paradigm where users address those problems during a Mashup building process by selecting, customizing, and connecting widgets together. While these systems claim that their users do not have to write a single line of code, merely abstracting programming methods into widgets has several disadvantages. First, as the number of widgets increases to support more operations, locating the right widget for the task can be confusing and time consuming. Second, customizing and connecting these widgets usually requires users to understand programming concepts. In this paper, we present a Mashup building approach that (a) combines most problem areas in Mashup building into a unified interactive framework that requires no widgets, and (b) allows users with no programming background to easily create Mashups by example.

Author Keywords
Mashups, Information Integration, Programming by Demonstration.

ACM Classification Keywords
H.5.2 User Interfaces: User-centered design

I. INTRODUCTION
Recently, average Internet users have evolved from content consumers to content providers. In the past, creating a simple web application was a complicated process. Today, we can create professional looking blogs or profile pages on a social network site without knowing HTML.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IUI'08, January 13-16, 2008, Maspalomas, Gran Canaria, Spain.
Copyright 2008 ACM 978-1-59593-987-6/ 08/ 0001 $5.00

The latest generation of WWW tools and services enables web users to generate web applications that combine content from multiple sources, and provide them as unique services that suit their situational needs. This type of web applications is referred to as a Mashup. A Mashup can be created as easily as manually typing information into each map marker in GoogleMap. More interesting Mashups, such as Zillow (zillow.com) and SkiBonk (skibonk.com), are much more complex because they need to deal with five basic issues:

Data Retrieval involves extracting data from web pages into a structured data source (i.e., table or XML). In addition to figuring out the rules to extract particular data from HTML pages [8,9], the structure of data on a page or the location of data which can span multiple web pages can make the process more complicated.

Source Modeling is the process of assigning the attribute name for each data column so a relationship between a new data source and existing data sources can be deduced.

Data Cleaning is required to fix misspellings and transform extracted data into an appropriate format. For example, the extracted data “Jones, Norah” might need to be transformed to “Norah Jones” to conform to the format of existing data sources.

Data Integration specifies how to combine two or more data sources together. For example, building a Mashup that lists all the movies ever performed by this year’s Oscar award winners will require us to merge (a) an Oscar winner list and (b) a movie database using a database join operation on the winner’s names.

Data Visualization takes the final data generated by the user and displays it (i.e., a table, a map, or a graph). Customizing the display and specifying the interaction model for the GUI often requires programming.

Our goal is to create a Mashup building framework where an average Internet user with no programming experience can build Mashups easily. Currently, there exist various Mashup building tools, such as Microsoft’s Popfly

(www.popfly.ms), Dapper (www.dapper.net), and Yahoo's Pipes (pipes.yahoo.com) to name a few. These tools aim at allowing users to build Mashups without writing code. However, not having to write code to build a Mashup does not always mean building one is easy. Most existing solutions employ a widget approach, where users select, customize, and connect widgets to perform complex operations. Figure 1 shows some widgets in Yahoo's Pipes that provide support for: fetching a RSS Feed, looping, and replacing a string using a regular expression.

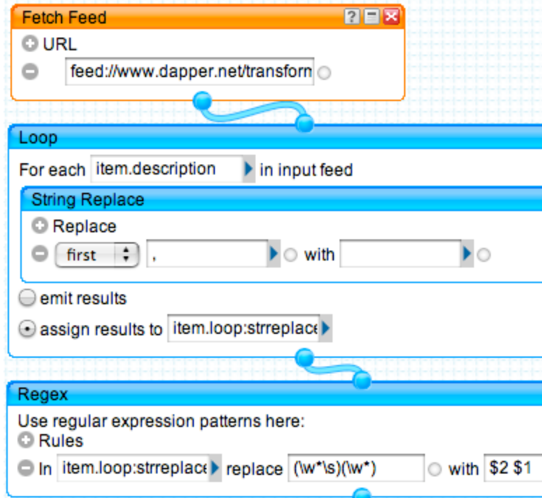

Figure 1. Sample widgets offered in Yahoo's Pipes. The user would customize each widget and connect them together to form complex operations.

There are two problems with the widget approach. First, the numbers of widgets (i.e., 43 for Yahoo's Pipes, 300+ for Microsoft's Popfly) can increase as Mashup tools try to increase their functionality. As a result, locating a widget that will accomplish the task can be difficult and time consuming. Second, while no programming is required, users often need to understand programming concepts to fully utilize them. Furthermore, most systems focus on particular information integration issues while ignoring others. As a result, the process of building Mashups is still quite complicated and the range of Mashups that naïve users can build is still limited.

In this paper, we illustrate how to address the first four Mashup building issues, often solved separately or partially, into one seamless process using the programming by demonstration paradigm. Using our approach, users (a) do not have to program or understand programming concepts to build a Mashup, and (b) indirectly solve each issue during the Mashup building process by only providing examples.

The rest of the paper is organized as follow: we first describe our motivating example, which highlights our approach. Then, we outline our approach and the details of each component. Next, we survey current Mashup solutions from both industry and the research world. Then, we provide the preliminary evaluation of our implementation against the current state-of-the-art offerings. Finally, we discuss our contributions and plans for future work.

II. MOTIVATING EXAMPLE

This section shows how a user would interact with Karma, our Mashup builder that incorporates the concept of programming by demonstration, to build a Mashup that

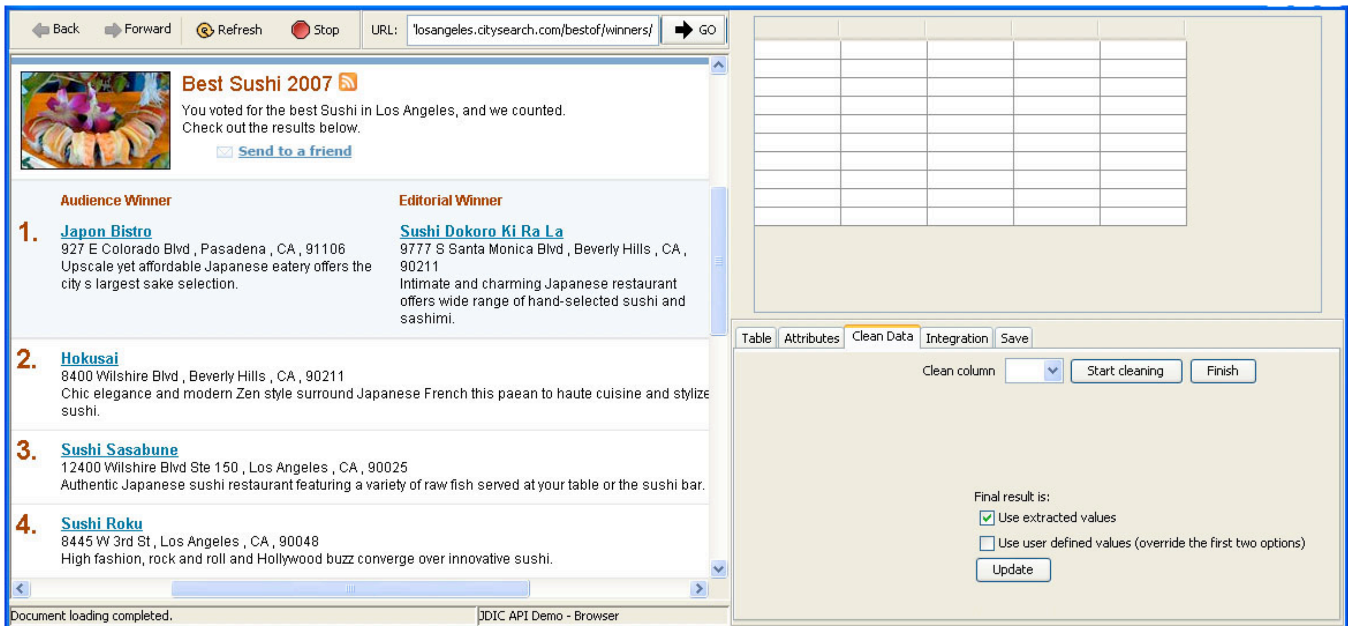

Figure 2. The interface of Karma. The left window is an embedded web browser. The top right window contains a table that a user would interact with. The lower right window shows options that the user can select to get into different modes of operation.

combines the listing of the best sushi restaurants in Los Angeles with information about their health ratings. Karma allows users to solve each information integration issue implicitly by simply providing examples. To build this Mashup, the user needs to combine data from two web sources: LA City Guide (http://losangeles.citysearch.com), and the LA department of public health (http://ph.locountry.gov/rating), and display the final result on a map.

We will break the Mashup building process into four consecutive modes: data retrieval, source modeling, data cleaning, and data integration. In practice, however, the user may switch freely back and forth between each mode. Also, the user can preview the Mashup display (i.e., map) at any time. Details of the inner workings will be elaborated in the next section.

Figure 2 shows the interface for Karma. The left area is an embedded web browser, where the user can navigate through web pages. The upper right area is a blank table where the data is populated based on the user's interaction with Karma. The lower right area shows multiple modes and their options from which the user can select.

Data retrieval

First the user will extract the data from the sushi page on the left into a table on the right side as shown in Figure 2. The end result table should look like the table in Figure 4, which contains restaurant names, addresses, descriptions, and number of reviews. Karma's goal is to let the user do this by providing only a small set of examples.

select one			
Japon Bistro			
Sushi Dokor…			
Hokusai			
Sushi Sasab…			
Sushi Roku			
Hide Sushi			
Fat Fish			
Sushi Katsu-ya			
Gindi Thai / …			
Katana			
Echigo			

Figure 3: By dragging "Japon Bistro" into the first row, Karma automatically fills the rest of the column

Once the user navigates to the best sushi restaurants page, he extracts the data by highlighting a segment of the text ("Japon Bistro") on the page, then dragging and dropping the highlighted text into a cell in the table on the right. Recognizing that the data element is a list on the web page, Karma proceeds to extract the rest of the restaurants from the page and fills the first column of the table in Figure 3.

The user can proceed to extract the address and the restaurant description of Japon Bistro, and Karma will automatically fill in the rest of the table. Note that the user can also click the link of Japon Bistro to go to its separate detail page and extract the number of reviews. Recognizing that the detail page belongs to Japon Bistro, which in turn is a part of the list in the original page, Karma then iterates through each restaurant in the list and extracts the corresponding data from detail pages. Figure 4 shows the result table where the user has extracted the restaurant name, addresses, description, and the number of reviews. Note that the user only has to drag in the four values in the first row to populate the entire table.

select one	address	select one	select one
Japon Bistro	927 E Color…	Upscale yet …	28 Reviews
Sushi Dokor…	9777 S Sant…	Intimate an…	3 Reviews
Hokusai	8400 Wilshir…	Chic eleganc…	30 Reviews
Sushi Sasab…	12400 Wilshi…	Authentic Ja…	66 Reviews
Sushi Roku	8445 W 3rd …	High fashion…	62 Reviews
Hide Sushi	2040 Sawtel…	No fuss, jus…	25 Reviews
Fat Fish	616 N Rober…	Inventive ro…	38 Reviews
Sushi Katsu-ya	11680 Vent…	The MOCA o…	49 Reviews
Gindi Thai / …	4017 W Riv…	Burbank res…	29 Reviews
Katana	8439 W Sun…	Rustic Japa…	96 Reviews
Echigo	12217 Sant…	Stellar sushi …	49 Reviews

Figure 4: The user extracts the whole list by dragging only four values into the first row of the table.

Source modeling

In the source modeling mode, Karma will help the user assign the right attribute name to each data column. While the user is busy extracting data from the page, Karma compares extracted data with existing data in its repository to identify possible attribute names.

For a column where Karma is confident, it fills in the attribute name automatically (i.e., address in Figure 4). For a column that Karma cannot identify or for which it is not confident, the attribute name is entered as "select one," as shown in Figure 4. The user can select the source modeling mode by clicking the "attribute" tab in Figure 2 and specifying the correct attribute by entering his own or searching from the list of existing attributes in the data repository. We will assume that the following attributes are assigned (by Karma and the user) to the table: *restaurant name*, *address*, *description*, and *number of reviews*.

Data Cleaning

Frequently, the extracted data needs to be cleaned because of misspellings and/or formatting. Karma lets the user clean the data by specifying the end result of what the clean data should look like. In this case, the user wants to get rid of the string "Reviews" in the fourth column of Figure 4.

To enter the cleaning mode, the user selects the "Clean data" tab in Figure 2. The user can then select which column is to be cleaned from the menu under the tab. Let us assume that the user selects the column "Number of reviews". The table will be in the cleaning mode as shown in Figure 5.

In the cleaning mode, two extra columns (user-defined and final) will be populated next to the column that the user

wants to clean. The user-defined column allows the user to enter the end result, and Karma will try to deduce the cleaning transformation from the user's examples. For example, if the user enters "28" in the first row, Karma will deduce the transformation between "28 reviews" and "28", and apply the same transformation to the rest of the data under the same column.

s	description	number o...	user defi...	final
lora...	Upscale yet ...	28 Reviews	28	
ant...	Intimate and...	3 Reviews		
shir...	Chic eleganc...	30 Reviews		
ilshi...	Authentic Ja...	66 Reviews		
3rd ...	High fashion...	62 Reviews		
vtell...	No fuss, just...	25 Reviews		
ber...	Inventive rol...	38 Reviews		
entu...	The MOCA o...	49 Reviews		
Rive...	Burbank rest...	29 Reviews		
Sun...	Rustic Japan...	96 Reviews		
anta...	Stellar sushi ...	49 Reviews		

Figure 5: Karma in the cleaning mode. The user can specify the clean result and Karma will try to induce the cleaning transformation.

Data Integration

In the data integration mode, Karma will analyze attributes and data in the table to determine possible join conditions between the data in the table and the data in the repository. Based on the analysis, Karma can suggest existing data sources in the repository that can be linked to the new data in the table. For example, let us assume that the LA Health Rating source has been extracted and stored in the repository through a similar process, perhaps by a different user. Based on the restaurant data in the user's table, Karma might suggest "Health Rating" as a new attribute that can be added to expand the table. If the user chooses "Health Rating" as the attribute for the new column, Karma will generate a query to retrieve the health rating data from the repository and fill the new "Health Rating" column.

The final result is the data table that contains restaurant data integrated with health rating information. Note that while Karma does not focus on the data visualization problem, Karma still provides a basic GoogleMap display if the table contains address information. The user can display the final restaurant Mashup on a GoogleMap by selecting a map option from the save tab in Figure 2.

While this example is about restaurants, the structure of the problem (i.e., extracting a list from a page, cleaning and integrating with other sources) is the same in general Mashup building tasks.

III. APPROACH

The approach that we use in Karma is based on two main ideas.

1. Instead of providing a myriad of widgets, we capture and model most Mashup building operations from examples that users can easily supply. In our case, users simply provide examples that they understand well – data elements from the website (i.e., Japon Bistro) or the resulting data that they want to see as the finished product (i.e., '28' from '28 reviews'). Providing examples should be easy, since building a particular Mashup implies that users know a little bit about the data from the web sources they want to extract and manipulate. By letting users work on data instead of programming widgets (i.e. stringtokenizer, loop, and regex), users do not have to spend time locating widgets and figuring out how to use them.

2. The reason that building Mashups can be difficult lies in those information integration issues stated earlier. Those issues are often solved separately, since each problem is already difficult on its own. As a result, most Mashup tools focus on some issues but ignore others, because subjecting users to the whole process is tedious and complicated. Karma overcomes this barrier by combining them together under a single interaction platform – a table. In the computer science research field, "divide and conquer" is one of the golden rules. However, we believe that our approach is logical and novel, because these issues are all interrelated. By treating them as a single process, results generated from solving one issue often help solve other issues.

The rest of this section is devoted to the technical details of how we implement our ideas in each of the problem areas and how information from one area is used to help solve problems in other areas.

Data Retrieval

In Karma, we use a Document Object Model (DOM) tree as a basic structure for the extraction process. The DOM tree is constructed based on the organization of HTML tags in the web page. Figure 6 shows the simplified DOM tree of the "best sushi restaurant" page from our motivating example.

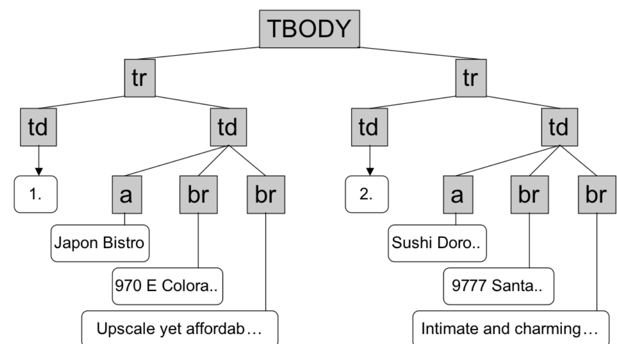

Figure 6. A simplified DOM tree that represents the best restaurant page in the motivating example. The gray nodes represent the HTML tags, while the white nodes represent the data embedded within those tags.

Extracting data from the same page

Using a DOM tree is an effective way to identify a list. For example, when the user drags the value "Japon Bistro" into

the table, we can (a) identify an XPath (www.w3.org/TR/xpath) from the root to that value, and (b) compute parallel Xpaths in different branches to extract other nodes that store restaurant names. An XPath is an expression language that is used to manipulate information in XML documents. For example, an XPath for "Japon Bistro" (i.e., /tbody/tr[1]/td[2]/a) means traverse the following path: *tbody*, the first *tr* tag, the second *td* tag, and retrieve all the *a* tag nodes. To find parallel paths, we can generalize the path by discarding the ordering number of nodes. For example /tbody/tr/td/a will return two nodes: /tbody/tr[1]/td[2]/a and /tbody/tr[2]/td[2]/a.

After extracting the first column of data, Karma handles extraction in other columns based on the position of the nodes in the first column. The set of nodes from the first column are used as markers to compute extraction rules based on the relationship between a marker and the newly extracted node.

For example, when the user starts dragging the restaurant's address (i.e. 970 E Colora…) into the same row as "Japon Bistro," Karma creates a mapping rule R: XPath_*marker* → XPath_*neighbor*, such that given a marker's Xpath (i.e., an Xpath to Japon Bistro), the rule can compute an Xpath for the "970 E Colora.." node. This mapping rule is then applied to other markers to extract their respective address nodes. The mapping rule is computed by first finding the common path between the marker and its neighbor. Then, the path not in common with the neighbor is added to the end.

XPath_*marker*: /tbody/tr[1]/td[2]/a
XPath_*neighbor*: /tbody/tr[1]/td[2]/br
Common Path: /tbody/tr[*]/td[*]/
Rule: *common_path* + br

So given an XPath for "Sushi Doro.." (/tbody/tr[2]/td[2]/a), we can apply the rule by extracting the common path and add *br* at the end, which will result in the XPath (/tbody/tr[2]/td[2]/br) that can extract Sushi Doroko's address node.

This mapping rule is used to disambiguate the case when there is a list within a list. In our example, we have a list of restaurants. And under each restaurant *td* node, we also have a list of two *br* nodes. If we did not use a mapping rule, then locating all the parallel XPaths to find address nodes with similar path structure to "970 E Colora.." will result in getting all four *br* nodes in Figure 6. Among these four *br* nodes, two of them contain the restaurant description, which we do not want in the address column.

Extracting data from detail pages

In our example, each restaurant has a link to its detail page, which contains more information about the restaurant. We want to extract this information as well. Under the hood, the following steps need to be performed to extract data from detail pages: (a) specify that the data on the first page is a list, (b) specify the link between each element in the list of the first page to its detail page, (c) extract the data on the detail page separately, and (d) specify how to combine the data from the first page with the data from the detail pages. Because of its complexity, most data Mashup tools do not support detail page extraction. Karma abstracts these tasks, so users can extract detail pages without explicitly doing all the above steps.

In Karma, we leverage the structure of the table to allow users to extract data from detail pages by example. While a table is a simple structure, there are multiple implicit constraints associated with it; the data in the same column is a list belonging to the same attribute. Also, the data on the same row is a combination of related content that forms a tuple.

When the user extracts "Japon Bistro," Karma can already induce that the first column is a list. Next, when the user navigates to the "Japon Bistro" detail page and drags the number of reviews into the first row of the table, the user indirectly specifies: (a) that a particular detail page is linked to "Japon Bistro," (b) the extraction rule for this new data element, and (c) where the new data element from a new page should be in the table with respect to the data from the first page.

By computing the mapping rule between the node that stores the URL of the detail page and its respective marker, Karma can locate other URLs from other restaurants, extract data from their detail pages, and fill the table automatically. This approach allows users to navigate deep into multiple levels of detail pages (not uncommon in many complex websites) and extract data while retaining the whole view of the overall extracted data in one table.

Source Modeling

In Karma, we keep a repository of data that can be used for source modeling, data cleaning, and data integration. This data is obtained from users previously extracting data and building Mashups. When the user adds a new column to the table, we use the repository to compute a set of candidate attribute names for the new column. Let:

V: a set of values from the new column.
S: a set of all available data sources in the repository
att(s): a procedure that returns the set of attributes from the source s where $s \in S$
val(a,s): a procedure that returns the set of values associated with the attribute *a* in the source s.
R: ranked candidate set:

$$\{a \mid \forall a,s: a \in att(s) \wedge (val(a,s) \subset V)\}$$

Figure 7 shows the mapping according to the constraint formulated for the first data column that contains restaurant names in Figure 4. After the user extracts the first value and Karma fills the rest of the column, Karma then uses all the values in that column as a starting set to find out possible attribute mappings. For each value in the starting set, Karma queries the repository to determine whether that

value exists in any table. If it exists, Karma extracts the attribute to which a value corresponds. For example, in Figure 7, there exist “Sushi Sasabune” and “Japon Bistro” under the attribute “restaurant name.” However, “Hokusai” can be associated with multiple attributes {restaurant name, artist name}.

In the case where all new values can be associated to only one attribute, Karma sets the attribute name of that particular column in the user table automatically. When there is an ambiguity, Karma sets the attribute name for that column to “select one.” Then, the user can select the attribute from a ranked candidate list. The ranking is computed by simply counting how many values can be associated with a particular attribute. For example, the attribute “restaurant name” will have a score of 3, while the attribute “artist name” will have a score of 1.

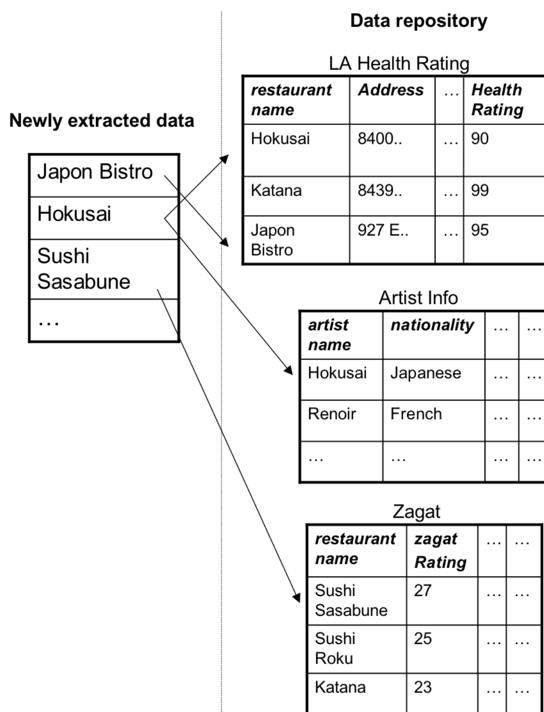

Figure 7. A view of the overlapping between newly extracted data and existing data in the repository.

Our method assumes that there is an overlap between newly extracted data and existing data in the repository. If there is no overlap, then Karma will also output “select one” as the attribute name for that column, and let the user select from the list of existing attribute names from the repository, or allow him to specify the attribute name himself. In the future, we plan to integrate the work on semantic modeling [11] to generate a better ranked candidate set.

Data Cleaning

Data cleaning is considered tedious and time consuming, because it involves detecting discrepancies in the data, figuring out the transformation to fix them, and finally applying the transformation to the dataset [13].

Usually, a Mashup is not considered an enterprise application. As such, some forms of error can be tolerated. However, it is still necessary to clean the data, especially when integrating multiple data sources using a ‘join’ operation. For example, if we want to combine two sources where the first one contain “jones, norah” and the second one contains “Norah Jones” under the same attribute “artist,” then the join condition will not produce a match.

In Karma, we use a cleaning by example approach that lets users specify how the cleaned data should look like. Karma then will try to induce the cleaning transformation rule. We adapt our cleaning by example approach from Potter’s Wheel [13]. Given a string of data, we first break the string into different tokens based on the following data types: <word>, <number>, <blankspace>, and <symbol>. For example, “jones, norah” would correspond to {*<word1>*, *<symbol>*, *<blankspace>, <word2>*}. Once the user specifies the cleaned result, for example “Norah Jones”, the user-defined data will also be broken into different tokens {*<word1>*, *<blankspace>*, *<word2>*}. Karma then tries to determine the transformation as follows:

First locate tokens with the same value between the *O* (original) and *D* (user-defined) set, and determine if the ordering has been swapped or not. If yes, add the swap instruction for that token into the set *T,* which stores all transformation sub-rules.

For each token in *O* that cannot be matched to *D*, apply a set of pre-defined transformations *S* and see if the result of the transformation can be matched to any value in D.

If no, then discard that token from O. If yes, add the pre-defined transformation and the swap instruction, if any, to *T*.

S is a set of pre-defined transformations that can be expanded to support more transformations. For example, one of the transformations is the method *capitalFirst*, which will transform the input word into the new word with the first character capitalized. Applying our procedure to the Norah Jones example above, the instructions in T would be: {delete <symbol>, set <blankspace> to position 2, apply capitalFirst to <word1>, set <word1> to position 3, apply capitalFirst to <word2>, set <word2> to position 1}. Applying *T* to “jones, norah” will result in “Norah Jones.” This *T* is then used to apply to other data under the same attribute.

In our example, when the user selects the cleaning mode, he can type in a new value (i.e., “28”) under the user-defined column. Then, Karma will try to compute a *T* that captures the transformation between “28 Reviews” and “28” and apply it to other values to fill the user-defined column. Note that Karma also lets the user define multiple cleaning rules (*T*) under the same column, and it will apply the first rule that matches the data in the cell. Finally, the user can decide how to combine the original, and user-defined data by checking the appropriate boxes shown in Figure 2.

Data Integration

Karma’s approach to the data integration problem is based on our previous work [14]. In this paper, we will provide

the intuition of how Karma solves the data integration problem. The theoretical constraint formulations that enable our approach to work are described in [14].

Our goal in data integration is to find an easy way to combine a new data source (that we extract, model, and clean) with existing data sources. The general problems are (a) locating the related sources from the repository that can be combined with a new source, and (b) figuring out the query to combine the new source and existing valid sources.

Karma solves these problems by utilizing table constraints with programming by demonstration. The user fills an empty cell in the table by picking values or attributes from a suggestion list, provided by Karma. Once the user picks a value, Karma calculates the constraint that narrows down the number of sources and data that can be filled in other cells.

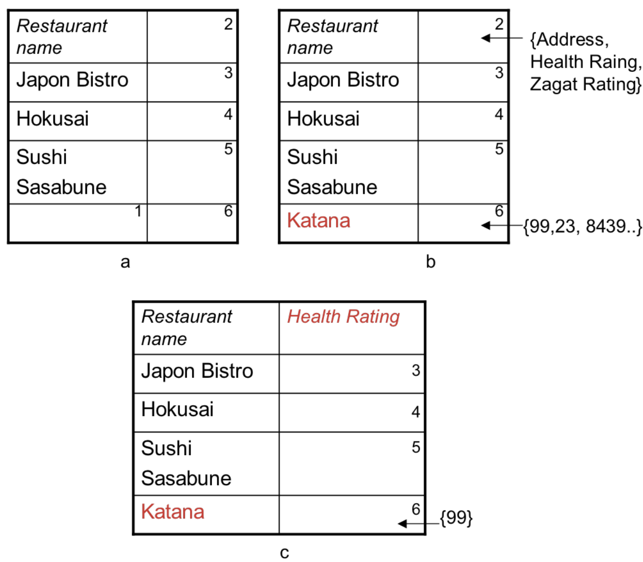

Figure 8 shows how the user can integrate new data with existing data through examples. When the user selects more examples, the table becomes more constrained. The value 1-6 designated empty cells.

To demonstrate how Karma handles the data integration, let us assume, for the sake of simplicity, that the user first extracts the list of restaurant names, and invokes the data integration mode. We will assume that our data repository only contains the three data sources from Figure 7.

Figure 8a shows a table with the newly extracted data, where the empty cells that can be expanded are labeled with numbers (1-6). Based on the existing data repository, there is a limited set of values that can fill each cell. For example, the value set that Karma will suggest to the user for cell 1 would be {Katana, Sushi Roku}. The reason is that to preserve the integrity of this column, each suggestion for cell 1 must be associated with the attribute "Restaurant name." We call this a vertical constraint where values under the same column must be associated with the same attribute name. Currently, there are only two sources with column "Restaurant name," so Karma formulates the query based on the vertical constraint to generate the suggestion list.

In Figure 8b, we assume that the user picks "Katana" to fill cell 1. To fill cell 6 (next to Katana), we need to ensure that the values Karma suggests come from a row in the source that has the value "Katana" associated with "Restaurant Name." We call this a horizontal constraint. These values are shaded in Figure 9.

From the horizontal constraint, the possible values that can be suggested in cell 6 would be {99, 23, 8439…}. The reason is that since Katana is a restaurant, there are only two valid rows that have Katana as a restaurant in the repository (row 2 from the LA Health Rating source and row 3 from the Zagat source).

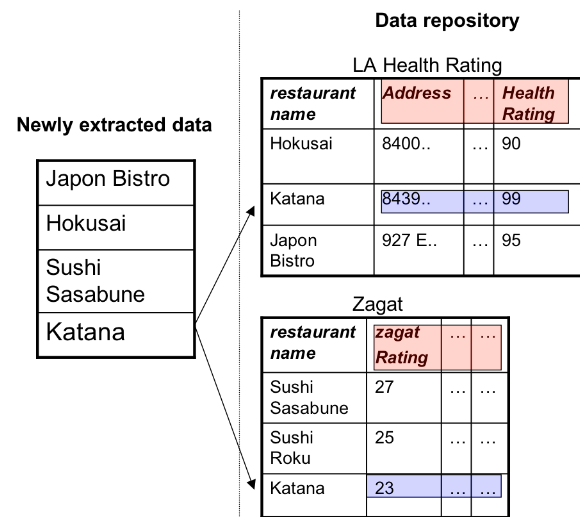

Figure 9. Selecting Katana in cell 1 limits the choices in other cells, such as cell 6 and cell 2, through the horizontal constraint.

On the other hand, cell 2 is only limited to three attributes (shaded in the attribute rows in Figure 9) since these attributes come from sources that have "Restaurant name" as one of the attributes. If the user picks cell 2 to be "Health rating" in Figure 8c, Karma can narrow down the choices through constraints and automatically fill the rest of the column (cell 3,4,5,6) with the health rating value with respect to each restaurant.

By choosing to fill an empty cell from values suggested by Karma, the user (a) does not need to search for data sources to integrate, (b) picks the value that is guaranteed to exist in the repository, yielding the query that will return results, (c) indirectly formulates a query through Karma, so the user does not need to know complicated database operations, and (d) narrows possible choices in other empty cells, as the user provides more examples.

IV. RELATED WORK

First we survey existing Mashup tools. Then, we review related fields of research.

Existing Mashup Tools

There exist a wide range of Mashup building tools from both industry and academia. We list the tools that aim to support average users in Table 1.

Simile [8], the earliest system among all, focuses mainly on retrieving the data from web pages using a DOM tree. Users can also tag sources with keywords that can be searched later. Dapper improves over Simile by providing an end-to-end system to build a Mashup. However, users still have to do most of the work manually to define attributes and integrate data sources together. Dapper provides only one cleaning operation that enables users to extract a segment of text (i.e., similar to Java's substring). Compared to Simile and Dapper, Karma extends the DOM tree approach to support more data structures and extraction from detail pages.

	Data Retrieval	Source Modeling	Data Cleaning	Data Integration
MIT's Simile	DOM	Manual	N/A	N/A
Dapper	DOM	Manual	Manual	Manual
Yahoo's Pipes	Widgets	Manual	Widget	Widget
MS's Popfly	Widgets	Manual	Widget	Widget
CMU's Marmite	Widgets	Manual	Widget	Widget
Intel's Mashmaker	Dapper	Manual	Widget	Expert

Table 1. Approach comparison between different Mashup tools segmented by problem areas.

Yahoo's Pipes, MS's Popfly, and CMU's Marmite [15] are similar structurally in terms of their approach. They rely on the widget paradigm where users select a widget, drop a widget onto a canvas, customize the widget, and specify how to connect widgets together. The difference between each system is the number of widgets (i.e., 43 for Pipes and around 300 for Popfly), the type of widgets supported, and the ease of use. For example, Marmite and Popfly will suggest possible widgets that can be connected to existing ones on the canvas, while Pipes will rely on users to select the right widgets. Compared to these systems, Karma uses a unified paradigm that does not require users to locate widgets or understand how each widget works.

Intel's MashMaker [5] took a different approach where its platform supports multiple levels of users. In MashMaker, expert users would do all the work in each area. For a normal user, she would use the system by browsing a page (i.e., Craigslist's apartment), and MashMaker will suggest data from other sources that can be retrieved and combined (i.e., movie theaters nearby) with data on the user's current page. Note that MashMaker supports only web pages that are already extracted through Dapper. Compared to Karma, MashMaker limits choices for its normal users to pages that exist in Dapper and data integration plans that have already been specified by experts.

In terms of data visualization, all Mashups building tools, including Karma, provide a set of display options for Mashups (i.e., Map), but none provides any framework that supports complex customization for the Mashup display.

Bungee Labs (www.bungeelabs.com), IBM's QED wiki (www.ibm.com), and Proto Software (www.protosw.com) are example Mashup tools for enterprise applications. These tools also use widgets to support most Mashup building functionality, but experts are required to use them because of their complexity. Google MyMaps allow users to create and import map points from limited sources. Aside from Google MyMaps, Google also has its own Mashup Editor (editor.googleMashups.com). However, it is aimed at programmers since programming is required.

D.Mix[6] and OpenKapow (openkapow.com) allow users to 'sample' or 'cut' data from web pages to be used later. However, both systems assume some level of expertise in programming in HTML and Javascript.

Related Research Fields

In the data retrieval domain, earlier work, such as Stalker [9], uses machine learning techniques to capture the extraction rules from users' labeled examples. Simile [8] employs the DOM approach, which requires less labeling. While this approach makes data retrieval easier, the DOM alone does not provide a mechanism to handle web pages with multiple embedded lists or detail page extraction. Karma fills these gaps by extending the DOM approach with the use of marker and table constraints.

Source modeling [7] outlined in this paper is closely related to the problem of schema matching. A good survey of source modeling and schema matching techniques can be found in [12]. While these techniques automatically generate possible mappings, the accuracy of these approaches is limited to 50-86% [4]. Karma solves the source modeling problem by using existing schema matching techniques [2] to generate possible candidate mappings and relies on users to determine the correct mapping. Since our users extract data from web pages themselves, we believe they can select sensible mappings.

A good survey of commercial solutions for data cleaning can be found in [1]. The data cleaning process in these solutions usually lacks interactivity and needs significant user effort to customize [13]. Karma's cleaning by example approach is based on an interactive data cleaning system called Potter's wheel [13], where users can specify the end result instead of writing a complicated transformation.

Our data integration approach is based on our past work in [14], which also contains survey of existing data integration approaches and systems. In [14], we assume that the problems of data retrieval, source modeling, and data

cleaning have already been addressed. Our work in this paper addresses that assumption and integrates four data integration techniques into a unified framework.

By combining these research problems, often solved separately, Karma can simplify and interleave each process, allowing greater flexibility. Karma pipelines data from one problem area to the next as soon as it is available. For example, as soon as the extracted data is available, it is sent to solve the source modeling problem automatically. Users can also switch seamlessly back and forth between each problem area during the Mashup building. For example, they can choose to extract and clean a particular column before moving on to extract more data in the next column.

Our framework is based on the concept called programming by demonstration [3,10], where methods and procedures are induced from users' examples and interaction. Clio [16] is a system for schema matching and data integration that also employs the programming by demonstration approach. However, it is intended for semi-expert users as understanding of source schemas and database operations are required.

V. EVALUATION

In this section, we perform an evaluation comparing Karma with Dapper and Yahoo's Pipes (we will refer to it as Pipes). The reasons for choosing these two systems are: (a) Dapper is an improvement, over Simile, (b) Pipes represents the widget approach and is readily available and more popular than Microsoft's Popfly, and (c) Intel's Mashmaker relies on experts to do most of the work, while our focus is on do-it-yourself Mashup building.

Claim and hypothesis

For the Mashup tasks that the combination of Dapper and Pipes (DP) can do, Karma lets users do it easier and faster.

Experimental setup

Designing the experiments that include qualitative and quantitative measurements between these systems is a challenge. First, Dapper and Pipes do not cover all the problem areas; Dapper's main focus is on data extraction from web sources and it outputs the result as an RSS feed. On the other hand, Pipes has widgets for cleaning and combining sources, but it cannot extract data from web sources that do not provide RSS feeds. Second, these systems have a high learning curve; users must read tutorials, try out examples, and understand programming concepts.

For our evaluation, we solve the first problem by combining Dapper and Pipes to finish our designed tasks; we use Dapper for data extraction and Pipes for the other data processing tasks. Note that the approach of combining tools to build a Mashup is not uncommon and is widely practiced by developers at MashupCamp (www.Mashupcamp.com), a biannual conference on cutting edge Mashup technology. For the second problem, we use an expert that knows every system used in the evaluation to do all the tasks. Then, the measurement is done as a unit of "*steps*." Each of the following actions constitutes one unit step: (a) typing values in a textbox, (b) clicking a button, (c) selecting options from a list, (d) dragging and dropping widgets from one area to another area, and (e) connecting one widget to another widget.

In our experiment, the expert will carry out three Mashup building tasks. Each task is designed to address some specific problem areas in the Mashup building process. Performance will be measured in the number of "*steps*" segmented by each problem area.

Tasks

1. Extracting a list of female adult contemporary artists (i.e., album name, artist name, description) created by an Amazon.com user at http://www.tiny.cc/0ctOx Notice that cleaning is needed to correct some artist names ("Jones, Norah" to "Norah Jones"). This is a simple task of extracting a list of data that requires simple cleaning.

2. Extract and combine cheapest gas prices from Los Angeles (www.losangelesgasprices.com), and Orange County (http://www.orangecountygasprices.com). These two data sources have identical structure and will require a database "union" to combine the two sources. There is no cleaning in this task.

3. Extract and combine the best sushi restaurant data with LA health ratings. This task is the same as the motivating example and we will assume that LA health rating data has already been extracted. This task requires using a database "join" to combine the two sources.

Result

	Data Retrieval	Source Modeling	Data Cleaning	Data Integration
Task1 K	3	7	6	0
Task1 DP	8	10	21	9
Task2 K	9	10	0	0
Task2 DP	18	30	0	28
Task3 K	5	10	4	5
Task3 DP	8	11	16	12

Table 2. Evaluation results for the tree tasks. The number of steps is broken down according to each problem area. K represents Karma, while DP represents a combination of Dapper/Pipes.

Table 2 shows the number of steps for each, task segmented by problem areas. *K* represents Karma, while *DP* represents Dapper/Pipes combination. Overall, Karma takes fewer steps in each area to complete the three tasks.

Task 1 involves extracting and cleaning data from one source. Karma allows the user to clean by example, resulting in fewer steps compared to DP. Figure 1 shows an actual snapshot of how the data cleaning is done in Pipes

for task 1. In addition, DP incurs a fixed cost of 9 steps to send the extracted data from Dapper to be cleaned in Pipes.

In task 2, DP needs to extract and define the output for each source separately, while Karma allows the expert to extract two sources into the same table. Also, the structure of the Karma table allows the union to be done implicitly; the expert can stack the data from the second source as new rows in the table under the first source. DP, however, needs 3 widgets to union the two sources together.

In task 3, the number of steps for each system is fewer compared to that of task 2 because we assume that the Health Rating source is already extracted. Note that DP is unable to extract detail pages as specified, so the result shown is actually (a) the steps DP takes to finish the task without extracting detail pages, and (b) the steps Karma takes to fish the task including detail page extraction.

Each scenario requires Dapper to be linked to Pipes causing additional steps in Data Integration. However, even if we ignore the cost of linking, Karma still performs better in each problem area.

VI. CONCLUSION AND FUTURE WORK

Our contribution in this paper is an approach to build Mashups by combining four common information integration techniques, often solved separately, into a unified framework. In this framework, users can build Mashups, without writing code or understanding programming concepts, by providing examples of what the end result for each intended operation should look like.

While existing work shares the same vision of building Mashups without programming, the widget approach still requires users to understand basic programming concepts. Furthermore, other tools lack a unified framework to make tasks simple for users and address only some of the Mashup building issues.

In terms of the future work, we plan to do an extensive user evaluation comparing our system to current state-of-the-art systems. We also plan to apply the same programming by demonstration principle to the problem of visualization to allow users to customize Mashup displays.

VII. ACKOWLEDGMENTS

This research is based upon work supported in part by the NSF under Award No. IIS-0324955, in part by the Air Force Office of Scientific Research under grant number FA9550-07-1-0416, and in part by DARPA, through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

REFERENCES

1. S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. In *SIGMOD Record*, 1997
2. W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. *In Proc. of the IJCAI*, 2003.
3. A. Cypher, Watch what I do: Programming by demonstration, MIT Press, 1993.
4. R. Dhamanka, Y. Lee, A. Doan, A. Halevy, and P. Domingos. Imap: Discovering complex semantic matches between database schemas. *In Proc. of SIGMOD*, 2004.
5. R. Ennals and D. Gay. User Friendly Functional Programming for Web Mashups. In *ACM ICFP*, 2007.
6. B. Hartmann, L. Wu, K. Collins, and S. Klemmer. Programming by a Sample: Rapidly Prototyping Web Applications with d.mix, UIST, 2007.
7. A. Heß and N. Kushmerick, Learning to attach semantic metadata to web services. *In Proc. of ISWC*, 2003.
8. D. Huynh, S. Mazzocchi, and D. Karger. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. *In Proc. of ISWC,* 2005.
9. C.A. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and reliably extracting data from the web: A machine learning approach. *Intelligent Exploration of the Web,* Springer-Verlag, Berkeley, CA, 2003.
10. T. Lau, Programming by Demonstration: a Machine Learning Approach, PhD Thesis, University of Washington, 2001.
11. K. Lerman, A. Plangrasopchok, and C. A. Knoblock, Semantic Labeling of Online Information Sources, *In Pavel Shaiko (Eds.) IJSWIS*, Special Issue on Ontology Matching, 3(3), 2007.
12. E. Rahm and P. Bernstein. On matching schemas automatically. *VLDB Journal, 10(4),* 2001.
13. V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. *In Proc. of VLDB*, 2001.
14. R. Tuchinda, P. Szekely, and C.A. Knoblock Building Data Integration Queries by Demonstration, *In Proc. of IUI*, 2007.
15. J. Wong and J.I. Hong. Making Mashups with Marmite: Re-purposing Web Content through End-User Programming. *In Proc of ACM Conference on Human Factors in Computing Systems*, *CHI Letters,* 9(1), 2007.
16. L. Yan, R. Miller, L. Haas, and R. Fagin. Data driven understanding and refinement of schema mappings. *In Proc. of SIGMOD*, 2001.