Journal of Visual Languages and Computing ∎ (∎∎∎∎) ∎∎∎–∎∎∎

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Gneiss: spreadsheet programming using structured web service data

Kerry Shih-Ping Chang *, Brad A. Myers

Human-Computer Interaction Institute Carnegie Mellon University Pittsburgh, PA, USA

ARTICLE INFO

Article history:
Received 4 September 2015
Received in revised form
3 February 2016
Accepted 6 July 2016

Keywords:
Spreadsheet
Mashup
End-user programming

ABSTRACT

Web services offer a more reliable and efficient way to access online data than scraping web pages. However, interacting with web services to retrieve data often requires people to write a lot of code. Moreover, many web services return data in complex hierarchical structures that make it difficult for people to perform any further data manipulation. We developed Gneiss, a tool that extends the familiar spreadsheet metaphor to support using structured web service data. Gneiss lets users retrieve or stream arbitrary JSON data returned from web services to a spreadsheet using interaction techniques without writing any code. It introduces a novel visualization that represents hierarchies in data using nested spreadsheet cells and allows users to easily reshape and regroup the extracted structured data. Data flow is two-way between the spreadsheet and the web services, enabling people to easily make a new web service call and retrieve new data by modifying spreadsheet cells. We report results form a user study that showed that Gneiss helped spreadsheet users use and analyze structured data more efficiently than Excel and even outperform professional programmers writing code. We further use a set of examples to demonstrate our tool's ability to create reusable data extraction and manipulation programs that work with complex web service data.
© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

The Internet is full of data. While many data are presented in the form of web pages, more and more data sources now provide *web services* that allow data to be retrieved using web APIs. Compared with scraping web pages, web services offer a more efficient and reliable way to access online data, as they do not rely on parsing web page layouts that might change frequently. Many data sources also support more sophisticated searches, and return more detailed data in their web services than on their websites. As RESTful web services have now become the mainstream,[1] web APIs are often accessed through URLs that are designed to be human-readable [1] and can be directly used in a web browser. The availability and variety of web services make them good resources for people who want to collect online data and perform custom data manipulations such as combining or comparing data.

However, working with web service data remains difficult for many people. The majority of modern web services return structured data, such as JSON and XML documents. The documents often contain large and complex hierarchies and are difficult for people to use without writing conventional code. Moreover, while it is common for people to collect data from multiple sources in daily tasks, working with data from multiple web services requires stitching together multiple web APIs, extracting desired parts from the returned data, and then performing further data operations. This usually requires people to write a significant amount of surprisingly intricate code that deals with asynchronous network calls which may fail to return, often requiring complex and sometimes nested call-backs [2]. Research has found that the majority of web API users are tech-savvy developers, and even they encounter many programing difficulties when trying to integrate multiple web services [3]. Data flow language tools such as Yahoo Pipes [4] and Microsoft PopFly [5] can let people wire up web services without writing conventional code, but studies have found that the dataflow concept is often difficult for people to comprehend [6,7], and these tools have not been particularly popular or successful with end-users.

2. Gneiss

We present Gneiss,[2] a new tool that extends the spreadsheet

* Corresponding author.
E-mail addresses: kerrychang@cs.cmu.edu (K.-P. Chang), bam@cs.cmu.edu (B.A. Myers).

[1] According to ProgramambleWeb (http://programmableweb.com/), there are over thirteen thousand web APIs on the Internet, and 70% of them are REST APIs.

[2] Gneiss (pronounced the same as "nice") is a kind of rock. Here it stands for Gathering Novel End-user Internet Services using Spreadsheets.

http://dx.doi.org/10.1016/j.jvlc.2016.07.004
1045-926X/© 2016 Elsevier Ltd. All rights reserved.

Please cite this article as: K.S.-P. Chang, B.A. Myers, Gneiss: spreadsheet programming using structured web service data, Journal of Visual Languages and Computing (2016), http://dx.doi.org/10.1016/j.jvlc.2016.07.004

ARTICLE IN PRESS

model to work with structured web service data. We chose to leverage the spreadsheet metaphor because it is popular with end-user programmers and familiar to many people [8]. Spreadsheet tools provide many functions for data manipulation and are commonly used to work with data. What current spreadsheet tools do not support well are interacting with web services and handling structured data. While some spreadsheet tools support getting web service data using functions [9,10] or built-in dialog boxes [11], they require people to write non-spreadsheet code to program the function or extract desired fields from the return data. They also do not support structured data and can only store a flattened string in a spreadsheet cell.

This paper is expanded from a previous publication [12], updated with a description of the new user interface for Gneiss and more details about the implementation. We have previously published four papers on Gneiss [12–15]. In summary, the previous papers have described:

- Interaction techniques and spreadsheet languages to construct two-way data flow between arbitrary REST web services and a spreadsheet without writing code [12].
- Interaction techniques to stream data in spreadsheets and manipulate the data using temporal information [13].
- A new method to visualize structured data in spreadsheets using the relative hierarchical relationships among adjacent spreadsheet columns. This method allows users to use interaction techniques to dynamically visualize a hierarchical JSON object as nested spreadsheet cells, or reshape and regroup the object by any fields [14].
- Results from a lab study that evaluated using hierarchical data in Gneiss and showed that Gneiss helped spreadsheet users complete data analysis tasks on average almost twice as fast as using Microsoft Excel, and even outperformed professional programmers writing JavaScript or Python code in most tasks [14].
- Using the spreadsheet language to program interactive, data-driven web applications [15]. The work does not necessarily involve using web service data, and focuses on the ways to program user interface controls.

This paper presents a new contribution that is the architecture and implementation of Gneiss, which has not been included in our previous publications. Gneiss is implemented as a web application with a client-server architecture. The server handles all the web service calls and stores data streamed from web data sources. On the client side, Gneiss uses an internal data model to support sorting, filtering and visualizing hierarchical data in spreadsheets. The interactions between the client UI and the server are implemented using multiple constraint objects from the ConstraintJS library [2]. We describe this in detail in the System Architecture section below. Another contribution of the current paper is to summarize and make clear how all the parts of Gneiss fit together.

The rest of the paper is structured as follows: Section 3 is related work. After related work, we describe a usage scenario to give an overview of Gneiss in Section 4, and present the Gneiss's user interface features in Section 5. Section 6 presents the system architecture. Section 7 describes some limitations of the current Gneiss system. We describe a lab study that evaluates using hierarchical data in Gneiss in Section 8, and a series of examples in Section 9 to demonstrate its ability to create fast and reusable data extraction and manipulation programs. We conclude this paper in Section 10.

3. Related work

Prior systems have tried different approaches to help end-users use web service data. d.mix [16] lets users copy web service calls from pre-annotated web pages to use in personal mashups. Marmite [17] uses a data flow approach to let users create mashups and has several built-in web services with which users can interact using form widgets. DataPalette [18] also uses built-in web services and focuses on helping users combine data from multiple sources. Commercial spreadsheet tools such as Microsoft Excel and Google Spreadsheet support functions or macros that access web services.

There are two unsolved issues in prior systems. First, many of these systems require a web service to be pre-programmed into the tool by a developer before it can be used by end-users through spreadsheet functions or widgets. This makes adding new web services to be used in the system almost impossible for end-users. End-users also cannot select which data to extract from the return documents because that is preselected by the developers when hardwiring the web service into the tool. In contrast, Gneiss introduces a flexible design that allows users to easily extract arbitrary fields returned by arbitrary REST JSON web services.

Second, prior end-user tools are very limited in terms of using structured hierarchical data such as JSON. For example, Excel's `WEBSERVICE` function lets people fetch data given an URL but can only return data as a flattened plaintext string, which is almost unreadable and unusable. The user needs to write additional query statements in languages such as XPath [19] to parse the document and extract desired data. Bakke et al.'s relational worksheet [20] shows the one-to-many relationships in relational databases using nested spreadsheet cells, but does not support further manipulating the nested data. There are also systems that try to extract hierarchical information from an exiting spreadsheet using its formatting and content layout (e.g., [21,22]). Different from those systems, Gneiss targets hierarchical documents instead of tabular documents (like spreadsheets and relational tables). Gneiss not only introduces a new way to show hierarchical data in spreadsheets but also extends spreadsheet languages, sorting and filtering to support manipulating and calculating data using its hierarchies.

Some prior tools focused on algorithms to extract data from regular web pages. Most of these tools use the structure of the web page and heuristics generated from the characteristics of the page. For example, Sifter [23] extracts search items on a web page using the HTML structure and scrapes subsequent web pages by examining hyperlinks (such as "Next page") and URL parameters. Vispedia [24] extracts Wikipedia infoboxes using the table structure and uses the hyperlinks in an infobox to retrieve related topics. There are also commercial web scrapers, such as Scraper [25], a Chrome plugin for scraping similar items in web pages, and ScraperWiki [26], a commercial product that specifically targets scraping Twitter and tabular data. Gneiss is different from these systems as it focuses on using data returned from web services. Currently, Gneiss also supports working with pre-downloaded structured (JSON) data. Therefore, Gneiss could be useful for manipulating structured data after it was scraped from web pages by these tools.

Some other tools focus on interaction techniques for extracting web page data. They also use the page structure to identify related items, and determine the data to be extracted based on the user's demonstration. Karma [27] allows users to extract data from a web page by dragging the first item to a table and then the system populates the rest of the rows. It also allows the user to simultaneously edit multiple similar cells by example, such as to reformat all phone number cells at once. Gneiss makes different contributions as it handles structured web service data and supports two-way data flow between the source and the spreadsheet table. Vegemite [28] lets users extract data by copying and pasting data from web pages to a table. It further records the user's activities in the browser such as entering, copying and pasting text, and pressing buttons, to generate step-by-step scripts to reuse in the future. Different from Vegemite,

Please cite this article as: K.S.-P. Chang, B.A. Myers, Gneiss: spreadsheet programming using structured web service data, Journal of Visual Languages and Computing (2016), http://dx.doi.org/10.1016/j.jvlc.2016.07.004

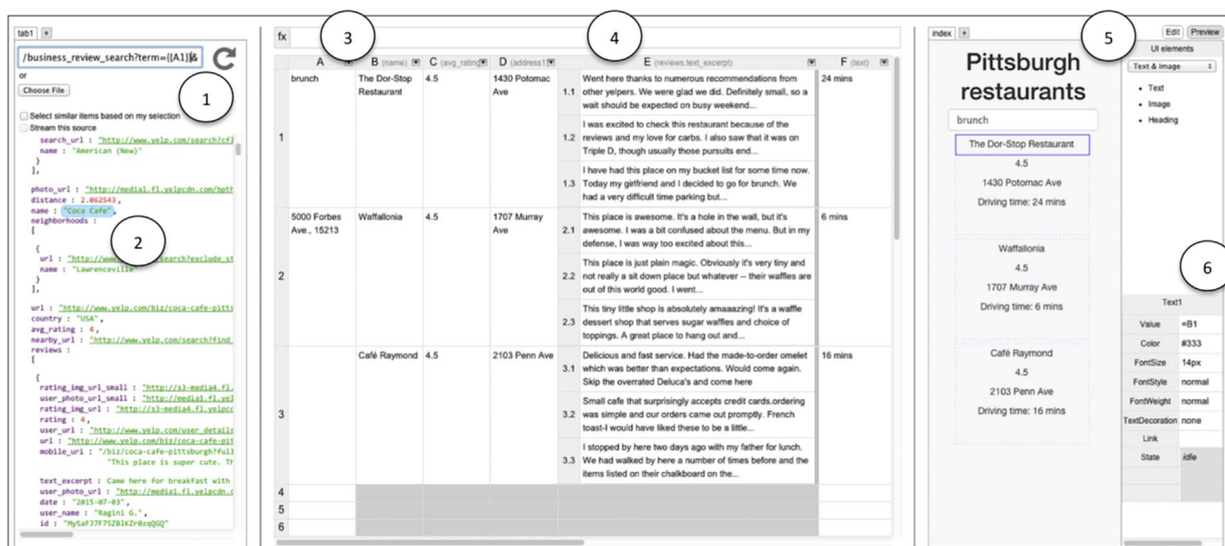

Fig. 1. A screenshot of Gneiss showing the spreadsheet program created in the usage scenario (Section 3). At the left is a source panel that shows raw data returned from a web service. (1) is the URL textbox where the user enters a web API. Note that the value of the cell A1 has been used as the search term. (2) is the returned data. (3) is the spreadsheet interface where the user can store desired fields extracted from the raw web service data and do manipulation. (4) If the extracted data have structure, they are shown in nested tables. The final results, the driving time from Alice's school to the restaurant, are shown in column F. At the right (5) is a web interface builder where the user can create a web application by dragging-and-dropping UI elements from the sidebar. The user can click on any UI element in the web page and edit its properties through a table (6). The value of an attribute can come from data in the spreadsheet, such as at (6) where the value of the text label comes from cell B1.

Gneiss uses the spreadsheet metaphor instead of recording user scripts to support two-way data flow, and is able to generate parallel-running data extraction programs instead of sequential scripts.

Some prior tools also use the spreadsheet metaphor to assist end-users in creating mashups, but they use very different approaches than Gneiss. For example, SpreadATOR [29] allows users to annotate a web data source by constructing a data model that specifies possible fields to use later. From the spreadsheet the user can select a field of a data model to view using functions. Gneiss takes a different approach of letting users extract web service data by demonstrating desired fields directly from the returned data through drag-and-drop without the need for any predefined data models. Gneiss also supports two-way data flow between the spreadsheet and the web data source, whereas in SpreadATOR, data flow is only one-way from the data source to the spreadsheet. Baglietto et al. [30] describe a framework that allows users to create mashups by linking multiple spreadsheets. Gneiss has a different focus, which is extending the spreadsheet metaphor to enable using and manipulating data from multiple web sources.

Another category of related work is research tools that extend the spreadsheet metaphor to support other kinds of end-user programming. For example, Forms/3 [8] is a spreadsheet programming language that allows the user to define objects using custom forms that contain cells that store constants and formulas. Forms/3 supports structured cells that hold other cells and matrixes. Also using structured cells to assist programming, Jones et al. [31] extends regular spreadsheet cells to store vectors. These tools do not deal with web data sources, which is the main focus of our work.

4. Usage scenario

Here we describe a scenario where Alice, a college student, is using Gneiss to create a spreadsheet program that uses a restaurant web service to look for the highest rated restaurants and then uses a direction web service to calculate the driving duration from her school to the restaurant. Fig. 1 is a screenshot of Gneiss showing the final resulting spreadsheet of this example. Gneiss has three panes. At the left is a source pane that displays raw data returned from a web service. In the center is a spreadsheet editor where the user puts the extracted web service data and manipulates them. At the right is a web interface builder where the user can create a web application that uses the data in the spreadsheet (this right pane is more thoroughly described elsewhere [15]).

To start finding restaurants, Alice enters Yelp's restaurant search API into the URL textbox (Fig. 1 at 1). The return data are shown below the textbox (Fig. 1 at 2). Alice wants the search term to be the value of cell A1 so she can initiate a new search by editing A1. To do so, she changes the value of the parameter “term” in the API (which represents the query term) to `{{A1}}` (Fig. 1 at 1). Now every time that A1 changes, a new restaurant search request is sent using A1's value as the query string, and the source panel updates to show the latest return data.

Alice does a few tests and makes sure this part works correctly by typing different search strings in A1. She then starts to extract the fields she wants. Alice first wants to use the name of each restaurant. She clicks on the “name” field of the first restaurant. The field gets highlighted with a blue background. She then drags the field and drops it on cell B1 in the spreadsheet. The tool automatically extracts the names of all restaurants and put them in column B, and a gray text appears next to column B's label to show the field name from where the data was extracted (Fig. 1 at 3). Alice then extracts the rating, address and reviews of each restaurant in the same way. The API returns multiple reviews for a restaurant. Alice selects the `text_excerpts` of the reviews of the first restaurant and drops them on cell D1. The tool populates the rest of the cells in column D and shows the reviews in nested cells (Fig. 1 at 4) .

Alice only wants the top 3 rated restaurants. To get this, she clicks on the arrow button at the top of column C, the column that stores the ratings, to bring up a dialog box that lets her sort and filter the extracted data (Fig. 2). She sorts the column in descending order and filters to show only the first three items.

The last step is to use a direction web service to calculate the time from Alice's school to the restaurant. Alice opens a new tab in the source pane and enters Google's direction API in the URL bar, and she binds the value of parameter “origin” and “destination” of the API to the value of cell A2, which has the address of her school, and D1, the address of the first restaurant. The API returns the

ARTICLE IN PRESS

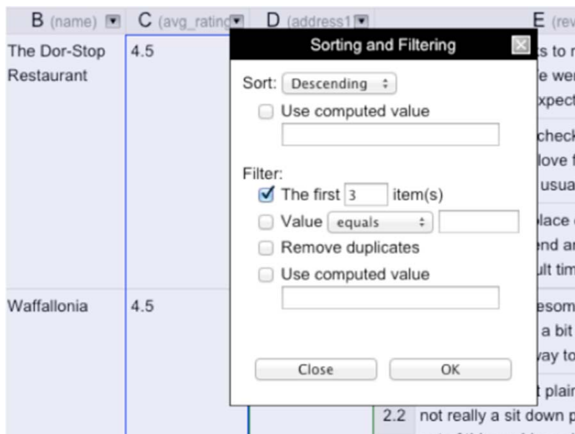

Fig. 2. Pressing the arrow icon at the top of each column brings up a dialog box that allows the user to apply sorting and filtering to that column. The rules can be constant or computed from other spreadsheet cells using spreadsheet functions. Data extracted from the same source (in neighboring columns) are sorted and filtered together and are highlighted with a purple background. The column from which the dialog box is opened is highlighted with a purple border. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

time, and she drags-and-drops it into cell F1, while F2 and F3 stay empty. To send two more direction requests using the other two restaurants' addresses as destinations to fill in F2 and F3, Alice selects F1 and moves the mouse to its bottom-right corner. The mouse becomes a "plus" sign for the familiar "auto-fill" command. She presses the mouse and drags down to F3 to fill in F2 and F3 with the time-to-destination of the two other restaurants.

Alice has now finished her data extraction program that uses Yelp's and Google's web services to find top-rated restaurants and the driving time from her school to the restaurants! She can easily look for another type of restaurant by changing A1, or instead view the time-to-destination from her house to the restaurant by changing A2. The spreadsheet will update to show the results based on the latest values in A1 and A2. While the direction requests can only be sent after the restaurant search request returns so all the addresses are filled in, among the direction requests there are no dependencies. Therefore, Gneiss sends the three direction requests in parallel to speed up the performance. Alice saves the spreadsheet she just created. The next time when she needs to do another restaurant query, she only has to load the spreadsheet back into the tool. She can further create a web page that shows the spreadsheet data (see Fig. 1 at 5). Gneiss' web interface builder lets her specify data bindings between a web page and the spreadsheet using simple spreadsheet languages [15] (Fig. 1 at 6).

5. Key user interface features

In this section, we describe the key features in Gneiss designed to assist people in using web service data in a spreadsheet. Again, details about how Gneiss supports creating data-driven web applications using spreadsheet languages are described in a separate paper [15].

5.1. Selecting and extracting desired data from web services

In Gneiss, raw data returned from web services are shown in a browser-like source pane at the left of the interface. The source pane supports multiple tabs which allow the user to load data from multiple web services. The user can select a desired field by directly clicking it. The selected field is highlighted using a blue background. She can extract it to the spreadsheet by dragging-and-dropping the field to a spreadsheet cell. If the "Select similar items based on my selection" checkbox (Fig. 1 at 1) is checked, the system will collect similar fields based on the selected field's path and put the values into a spreadsheet column. For example, in the usage scenario, to create a column where each cell in the column has a restaurant name, the user only needs to select one restaurant name and drag it to the top of a column. With the "select similar items" checkbox checked, the system fills in the rest of the column with the other restaurant names.

When the user drags some web service data to a column, the column becomes reserved to only show those data. Empty cells are grayed out to show that they are not available for manual edits (see Fig. 1). Since the web service data come in dynamically, we adopted this design to avoid the situation where the user's data gets accidentally erased if the new web service data is longer than the first demonstration. For example, in the scenario, the number of restaurants returned from the web service might be different each time based on different query terms. The user can select a column and choose to clear all contents to start over. Users are free to type anywhere in columns that just contain user-typed data, like column A.

To view the raw return data of any extracted cell, the user can right click on the cell and choose "view source". The corresponding tab in the source pane that shows this web service data will be brought to the front (or if the corresponding tab was closed by the user, the system will open a new tab and reload the data there). So for example, in the scenario, after Alice finishes extracting data from Google directions, she can right click on any restaurant cell and choose "view source". The tab that shows the raw restaurant search data returned from Yelp will be brought to the front. She could then do more edits or extract other fields.

5.2. Streaming data from web services to spreadsheets

Our tool also allows the user to stream data from web services [13]. This feature could be useful when the web service returns real-time data, such as financial data, weather or traffic information. The user can check the "stream this source" checkbox in the source pane to let the system periodically fetch data from a web service (where the time is configurable in a dialog box), and extract a desired field to a spreadsheet column similarly using drag-and-drop. The system then stacks the column with the latest value returned from the web service. The spreadsheet becomes highly dynamic as the streaming columns can grow and change over time and the spreadsheet functions (such as `SUM` and `AVERAGE`) can compute summaries of data in real-time. More details on Gneiss' user interface supporting streaming data are in a separate paper [13]. In the current paper, we present in the system architecture session Gneiss' backend architecture to interact with web services.

5.3. Working with hierarchical data

Many web services return hierarchical data. For example, every restaurant returned from Yelp has a "reviews" array that contains a list of reviews. A convention in spreadsheet usage, we observe is that data in the same row are often considered as belonging to the same item. Also, the "key" of the data is often put at the leftmost column of a table. We use these observations to design a new method to visualize hierarchical data in Gneiss (the method is described in detail in [14]). In summary, this method uses the *relative hierarchical relationship* between a column and the column to its left to decide how to visualize the data. When the user drags a field to a spreadsheet column, the system first checks if the adjacent column to the left comes from the same hierarchy tree in the same document (see Fig. 3 for an example):

- If the data in its left column comes from an *ancestor* field, the system fills in the new column with nested spreadsheet cells to

Fig. 3. Four screenshots of Gneiss showing Star Wars' data extracted from Rotten Tomatoes' movie search web service (movies 1–6 only). (1) The original returned data is a JSON document grouped by movie titles. Each movie has an actor array that has a list of actors, and each actors has a character array that stores a list of characters played by the actor in the movie. The user can regroup a JSON document by an arbitrary field by moving the field to the beginning of the table. For example, if the user wants to group the Star Wars data by actors, she (2) first moves the actor names to column A using drag-and-drop. The system reshapes the data into a flat table showing a character in the same row with the actor who played it and the movie the character was in. The user then (3) right-clicks column A and selects the "Group By" option to merge cells that have the same values in column A. (4) is the final result, showing a Star Wars character, all the actors who had played that character, and the name of the movie the actor was in.

put the descendants in the same row with their ancestor (such as Fig. 3 at 1).

- If the data in its left column comes from a *descendent* field, the system repeats the ancestor value in this column to again let the ancestor be in the same row with its descendent (such as Fig. 3 at 2).
- If the data in the left column comes from a sibling field, the system puts the siblings in the same row.
- If the adjacent column to the left is empty, does not exist or does not come from the same hierarchy tree in the same document, the system shows data in this column as a regular column with no nested spreadsheet cells.

This method enables users to dynamically create different views of a same hierarchical object by changing the order of the columns using drag-and-drop. The user can easily reshape and regroup a hierarchical object by any field to explore and analyze the data. For example, in Fig. 3, the user can first look at Star Wars data by movie titles (Fig. 3 at 1), and then regroup the data by character names to look at who had played the character and in which movie (Fig. 3 at 4).

Gneiss also extends spreadsheet language syntax to support using nested data in spreadsheet formulas. The row label for a nested cell is `parentIndex.thisIndex`. Referring to a nested cell is similar to referring to a regular cell using the cell's column name and row number. For example, to refer to the cell with value "Ewan McGregor" in Fig. 3 at 1, the user enters `B1.3`. Similarly, in Fig. 1 at 4, to refer to the third review of the second restaurant, the user would enter `E2.3`. To select a range of nested cells is also similar to how conventional spreadsheet works, using the syntax `beginCell: endCell`. For example, in Fig. 1, `E1.2: E2.3` returns five review text items from the first restaurant's second review to the second restaurant's third review. Entering a parent cell name will select all nested cells in it. For example, in Fig. 1 entering `E1` will return all reviews of the first restaurant. Values in the nested cells can be used in conventional spreadsheet functions just like regular cells. For example, in Fig. 1 `=COUNTIF(E1.1: E1.3, "mussel")` counts the occurrences of "mussel" in the top three reviews of the first restaurant.

Gneiss also supports sorting, filtering and creating new fields in a hierarchical object. For example, in Fig. 3 at 4, the user can find out which character appears in the most movies by inserting a new column B next to column A, entering `=COUNT(C1)` in B1 to get the number of appearances of the first character, autofilling column B to get the values for all characters, and then sorting the entire data by column B to bring the character that appears in the most movies to the top. Finally, Gneiss also supports *joining* multiple hierarchical objects by common fields, resulting in a new view where the hierarchies are connected. The user interface and interaction techniques to do these operations are described in detail in [14]. In the current paper we present how we implemented these features to use hierarchical data in Gneiss below in Section 6.

5.4. "Auto-filling" cells with web service data

"Auto-fill" (also called "fill down") is a common feature in spreadsheet tools where the user selects one or multiple cells and drags them to fill in additional cells. In Gneiss, when the user selects and drags a cell whose value comes from a web API that uses values of other cells in the same row, as the user drags it down, the system will replace these cells in the web API with the corresponding cells in the new row. For example, in the scenario, cell F1's value comes from the direction API that uses the value of cell A2 and E1 as the value of the origin and destination parameter. When the user selects F1, the system recognizes that E1 is in the same row as F1. When the user drags down to rows 2 and 3, the system replaces E1's value in the web API with E2's and E3's values, sending the required API requests, and extracting data using the same path to fill in F2 and F3.

5.5. Dynamic sorting and filtering

The regular sorting and filtering in Microsoft Excel or Google Spreadsheets only sorts and filters the current data in a column and do not apply to future edits. For example, after sorting a column in alphabetical order, adding new values to the column does not reorder the cells. In contrast, in Gneiss, once sorting and filtering are applied to a column, they are executed every time when the data in the column change. This allows the user to apply selection rules to the dynamic web service data using sorting and filtering and

ARTICLE IN PRESS

enhances the reusability of the resulted data extraction program. For example, in the scenario, Alice gets the top-rated restaurants by sorting the rating field in descending order and filtering to show the top three items. The sorting and filtering will execute every time when she makes a new query and a new list of restaurants comes in.

To apply sorting and filtering to a column, the user clicks on the arrow icon at the bottom-right corner of the column title (see Fig. 2). A dialog box will appear to let the user specify how they want the column to be sorted and filtered. Currently our system supports sorting ascending and descending, filtering by the number of items, by the cell's values or by the cell's fetched time (for streaming data), and removing duplicates. Our system also allows sorting and filtering rules to be decided dynamically using spreadsheet formulas. To set up dynamic rules, the user selects the "use computed value" checkbox in the dialog box and enters the formula in a text box. For example, entering the formula `=IF(D1="Yes", "Descending", "Ascending")` as the computed value for sorting will dynamically change how the data are sorted based on cell D1's value.

5.6. Two-way data flow and parallel-running programs

As described in the usage scenario, the user can easily replace any part of the web API URL in the source panel with the value of a cell in the spreadsheet, using the syntax `{{cellName}}` (Fig. 1 at 4). This allows data in the spreadsheet to be sent to a web service. As we showed in the scenario, this data can either be constant (like A1) or computed based on other cells (like E1). As described earlier, data from a web service can be extracted and stored in the spreadsheet through drag-and-drop. Together, these two features enable the data to flow two-ways between our tool and the web service. If a spreadsheet cell like B1 stores data coming from a web API that uses another spreadsheet cell A1's value, every time that A1's value changes, an API call will be made to update B1's value as well. This makes the data extraction program created in Gneiss easily reusable, as the user can make a new request to a web service and view the extracted data by simply editing spreadsheet cells, as demonstrated in the scenario.

While the user's demonstration and manipulation of cells is always performed sequentially, the spreadsheet metaphor allows our tool to construct a parallel-running program using the dependencies among the spreadsheet cells. Cells that do not have any dependencies on each other can be computed independently in parallel. This could make a big improvement on performance, especially when extracting a large amount of data. For example, if the user wants to collect data using 50 online shopping web services to compare prices, she can easily create a spreadsheet program that sends 50 web API requests in parallel, and get the data within seconds. This is in contrast to web scraping programs like Vegemite [28], that not only can only execute sequentially in the same order as when the user demonstrated them, but must insert fairly long delays into the program (for example, 10 s in the example in [28]) in hopes that this will be long enough to make sure the web pages have finished loading. Since our tool uses web services, we know when the call has completed, and no extra delays are required.

5.7. Error Handling and maintenance

A web service call could sometimes fail due to various reasons such as scheduled maintenance, change of protocols, a bad Internet connection, or even a bug in the user's specification of the URL. Gneiss is robust in handling this, as it will fill the error cells with a special "error" value and propagate this value to all dependent cells when it receives a HTTP error when accessing the web service. The user can easily see the errors in the spreadsheet and trace back to the origin using conventional spreadsheet mechanisms. Our system's ability to execute programs in parallel also allows other parts of the program that do not depend on the error cells to run as usual. Debugging is also easier in our tool because the user can see all the values in the spreadsheet and see them changing at run time, unlike in other programming languages, such as Yahoo Pipes, where data are typically hidden unless the user looks for them.

The user can save the spreadsheet program and load it back up when she wants to perform similar queries. Our system also provides an option to keep updating the spreadsheet data at the server even when the user closes the spreadsheet in the frontend. This could be useful when the user wants to stream data over time. Reusing a spreadsheet program is easy since the user can edit a spreadsheet into another one and reuse appropriate parts. For example, in our scenario, if Yelp stops supporting its web service after the spreadsheet program is created, the user could use another web service to search for restaurants (such as Google Places) and replace the restaurant columns in the same spreadsheet with appropriate fields from the new data source, without having to modify the location part.

6. System architecture

Gneiss is implemented as a web application using HTML, CSS and JavaScript.

6.1. A client-server architecture

Gneiss uses a client-server architecture with a client side that is the spreadsheet tool with which the user interacts through the web browser, and a backend server that handles all of the web API calls and stores the retrieved data. We use Node.js as the server and MongoDB as the database. The server is mainly designed to help use streaming data, as the size of the data could get very large over time. Having a server enables our system to store more data without slowing down the user's machine, and even to keep streaming new data in the background when the client spreadsheet is closed. For example, the user can create a spreadsheet in the morning that retrieves stock market prices from a finance web service, set the system to fetch data every 30 min, and save and close the spreadsheet. In the evening when the user reopens the spreadsheet, she can see how the stock price changes over the entire day. When the user sets up sorting and filtering rules for streaming data using the data's fetch time (such as to show only today's data), the server will sort and filter the data first to only send the necessary data to the client. While we have not implemented the server-side sorting and filtering for non-streaming data (as the sizes of data returned by most of the non-streaming commercial web services we have seen are very small), it is completely doable. Our architecture could be extended easily for users who have such needs.

6.2. The use of constraints

On the client side, we use ConstraintJS [2] to maintain the dependencies among the spreadsheet cells and handle the communication with the server. The client mainly uses three types of constraints:

- *Web API constraints*: We use two sets of constraints to connect the web queries and the data retrieved. The first keeps the server-side information consistent with what the user has specified. It sends a web API string (such as the one in the URL bar in Fig. 1 at 1) along with information about the use of that API (such as the sorting and filtering rules or the streaming frequency set by the user) to Gneiss' backend server to send and retrieve data to the actual web service. Another set of constraints tie the data returned from the server (when waiting for the server's response, the constraint's value is set to

"Loading…") with the data displayed in the spreadsheet. The constraint solver automatically reevaluates these constraints (which will therefore send a new request to Gneiss' server) when the API string or any usage information changes its value (for example, if the web query uses the value of a spreadsheet cell and that cell changes).

- *Spreadsheet cell constraints*: Every spreadsheet cell is implemented using a constraint object that returns the cell's value. A cell constraint can be a constant value or a computed value that depends on other constraints. For a cell that holds web service data extracted from the source pane, its constraint is set to a function `getWebServiceData(apiURL, path)`. This function uses the spreadsheet cell's column, `apiURL` and `path` to tie the cell with a constraint that returns an internal JSON object representing a *hierarchical table*, and extracts the data for that cell to renders the corresponding display (see details in the next section).
- *GUI element constraints*: As briefly mentioned earlier, Gneiss has a web interface builder that lets the user create web pages where elements in the web page can use spreadsheet data as its attribute value (see Fig. 1 at 6). Each attribute of a web interface element created in Gneiss (such as the color of a text label or the value of a textbox) is a GUI element constraint object. Similar to spreadsheet cell constraints, GUI element constraints can return constant values or computed values depending on other constraints.

6.3. Internal JSON data models

In Gneiss, every *hierarchical table* is stored internally as a JSON object. A hierarchical table is a set of adjacent spreadsheet columns that come from the same document and belong to the same hierarchical tree in the original document returned from web services. For example, in Fig. 1, column B–E form a hierarchical table as the data in these columns are from the same document returned from the same web service. Internally, there is a system function that maintains the "profiles" of all hierarchical tables in the spreadsheet. The profile of a hierarchical table includes the columns that form this table, and for each of the columns, the column path to its data in the original document (for example, for column B in Fig. 1, its column path is `$[*]["name"]` as it stores all restaurant names), and any sorting, filtering and grouping rules the user has set on this column. This function will check every time when the user drags a column to a different location or sets a new sorting, filtering and grouping rule, to see if any part of the hierarchical table's profile needs to be updated or if a new hierarchical table is created. If so, the function will trigger the reconstruction of that table's JSON object, using the following rules:

- It first creates an empty JSON object with one array field whose name is the name of the leftmost column. The system collects all values of this column from the original document using the column path, and creates the same number of items to put in the array. Each array item has two fields: a "`value`" field that stores the value, and a "`cell_path`" field that stores the path to this value. The system also stores the name of this column in a temporary variable called "`root_array`".

  For example, in Fig. 1, column B is the start of a hierarchical table. The system creates a JSON object like this:

```
{"B": [{"value": "The Dor-Stop Restaurant",
        "cell_path": "$[0]['name']"},
       {"value"; "Waffolonia",
        "cell_path": "$[1]['name']"},
    …
 ]}
```

  `root_array` is now "B".
- Then for the rest of the columns, the system compares a column's path to its immediate left column's path.

1. If this column's path is at the same array level as the immediate left column's path, the system creates a new *object* whose name is the name of this column inside each item in `root_array`. The new object also has a `value` and a `cell_path` field. The system uses the `cell_path` field in the `root_array` to construct the `cell_path` for the new object and get the new `value`.
   Continuing the previous example, in Fig. 1, column C's column path (`$[*]["rating"]`) is in the same array level as column B's column path (`$[*]["name"]`). The JSON object after processing column C becomes:

```
{"B": [{"value": "The Dor-Stop…"
        "cell_path": "$[0]['name']"
        "C": {"value": 4.5,
              "cell_path": "$[0]['rating']"
             }
        },
      …
 ]}
```

   `cell_path` in the first item in `root_array` (B) starts at array index 0. So the `cell_path` for the new object C in the first item also starts at array index 0.
2. If this column's path is at a deeper array level than the immediate left column's path, the system creates a new *array* whose name is the name of this column inside each item in `root_array`. Each item in the new array also has a `value` and a `cell_path` field. Again, the system uses the `cell_path` field in items in the `root_array` item to construct the `cell_path` for each item in the new array. After the system updates the JSON object, `root_array` become the current column.
   Continuing the previous example, in Fig. 1, column E's column path (`$[*]["reviews"][*] ["text_expert"]`) is in a deeper array level as column D's column path (`$[*]["address"]`). `cell_path` in the first item in `root_array` (B) starts at array index 0. Therefore, the array D in the first item in B has the path `$[0]["reviews"] [*]["text_expert"]`. The JSON object after processing column E becomes:

```
{"B": [{"value": "The Dor-Stop…"
        "cell_path": "$[0]['name']"
        "C": {"value": 4.5,
              "cell_path": "$[0]['rating']"
             },
        "D": {"value": "1430 Potomac Ave",
              "cell_path": "$[0]['address']"
             },
        "E": [{"value":"Went here thanks…",
               "cell_path": "$[0]['reviews']
 [0]['text_expert']"},
              {"value": "I was excited to…",
               "cell_path": "$[0]['reviews']
 [1]['text_expert']"},
              …
             ]
       },
     …
 ]}
```

`root_array` is now "D".

ARTICLE IN PRESS

3. Finally, if this column's path is at an upper array level than its immediate left column's path, the system creates a new *object* whose name is the name of this column inside each item in `root_array`. Similarly, each new item has a `value` and a `cell_path` field. Again, the system uses the `cell_path` field in items in the `root_array` to construct the `cell_path` for each item in the new object and get the new value.

Continuing the previous example, in Fig. 1, suppose the user drags the country field of the restaurants to column F. Column F's path now becomes `$[*]["country"]`, which is in the upper level array of column E (`$[*]["reviews"][*]["text_expert"]`). So the system creates an object name "F" in each item in array E. For the first item in array E in the first item in array B, its `cell_path` starts at index 0 (`$[0]['re-views'][0]['text_expert']"}`). Therefore, the `cell_-path`in the new object F in the first item in array E in the first item in array B becomes `$[0]["country"]`. Now when we look at the second item in array E in the first item in array B, its `cell_path` also starts at index 0 (`$[0]['re-views']`[1] `['text_expert']"}`). So the `cell_path` new object F in the second item in array E in the first item in array B is still `$[0]["country"]`, creating a duplicate value. The JSON object after processing column F becomes:

```
{"B":[{"value": "The Dor-Stop…"
       "cell_path": "$[0]['name']"
       "C":{"value": 4.5,
            "cell_path": "$[0]['rating']"
            },
       "D":{"value": "1430 Potomac Ave",
            "cell_path": "$[0]['address']"
            },
       "E":[{"value":"Went here thanks…",
            "cell_path":"$[0]['reviews']
[0]['text_expert']",
            "F":{"value":"USA"
            "cell_path":"$[0]
['country']"}
              },
              …
           ]
       },
     …
]}
```

Once the system finishes building this JSON object, sorting, filtering and grouping the object become straightforward. Sorting and filtering a column affect only the current array level. For example, sorting by column E (review text) will only change the item order in array E using the `value` field in each item. It does not affect the item order in array B. Sorting by column C (ratings), however, will change the item order of array B, as object C is directly in array B. If a column is selected for grouping, its next (right) column will be become an array (if not one already) to store the merged values. Every time an internal JSON object changes, all spreadsheet cells linked to this object will refresh and show the latest visualization.

The implementation of the internal JSON data model enables users to easily restructure a JSON document by drag-and-dropping columns. Different from prior work that handles hierarchical data in spreadsheets by flattening them into relational tables (e.g., [32]), our tool operates directly on hierarchical objects. This enables users to further manipulate the data using custom structures, such as to perform hierarchical sorting and filtering as we just discussed. It allows Gneiss to easily support hierarchical visualizations such as treemaps [33] in the right pane.

7. Limitations

Gneiss has a number of limitations. First, it currently supports only RESTful web services that return JSON data. It does not support web services which use other protocols, such as SOAP, or which return other kinds of data formats, such as XML or CSV. However, we believe that all the key features of our tool should still be able to be applied to these kinds of web services with minor changes in the source panel to accommodate the differences. Second, many search APIs limit the number of return values in a single return document, usually controlled by a parameter in the API. To access additional results, the user needs to ask for the "next page" of this query, usually controlled by another parameter. Our tool currently does not have a way to automatically ask for the *next* set of results, so the user would need to change the parameters manually. Finally, in Gneiss we do not focus on helping users *understand* how to use a web API. This potentially limits our target users to people who already have some basic knowledge about web APIs, such as being able to understand the JSON format and the API documentation in order to know what the parameters are. Our prior work, Spinel [34], provides an architecture that enables new web services to be added to an application as plugins. Spinel includes an end-user tool that lets people create a plugin following the instructions of the application developer without writing any code. In the future, we can leverage this idea to provide a few popular web APIs as built-ins that can be configured using graphical widgets (such as in [17]) and use Spinel to allow end-users to create plugins that would display such widgets for a new web API to be used in Gneiss.

8. User evaluation

We conducted a lab study that focused on evaluating how Gneiss enabled users to work with hierarchical data. In the study, participants did not directly interact with web services. Instead, they used two local JSON documents that were retrieved from a real web database by the experimenter before the study. Participants had to extract fields to the spreadsheet and manipulate the data using Gneiss' new spreadsheet language syntax and interaction techniques (e.g., sorting, filtering and grouping) to complete the study tasks. The study is described in detail in [14]. Here we quickly summarize the study and its findings.

The study was a between-subject study that had three groups: one group used Gneiss, another group used Microsoft Excel, and the third group wrote JavaScript or Python in a text code editor. Each group had 6 percipients. Participants using Gneiss and Excel were intermediate to advanced spreadsheet users who were not professional programmers and had almost no experience working with JSON data. Participants writing code were all professional programmers familiar with JSON. We designed five data exploration tasks for all participants to finish. The two JSON documents we used were the CHI'15 conference data about papers and sections. We converted the JSON files to CSV files for the Excel group to do the task in Excel.

Overall, participants using Gneiss finished the tasks significantly faster than both participants using Excel and writing conventional code [14]. We found that the Gneiss participants were able to learn and use the nested cell visualizations, the new spreadsheet language syntax and interaction techniques to manipulate the data after a 20-min tutorial. Compared to Excel, Gneiss' hierarchical nested cells

avoided having many repetitive values in the data required in Excel to flattening the structure, making the data in Gneiss cleaner and thus easier to work with. Gneiss also let participants calculate summaries of data using familiar spreadsheet formulas instead of requiring advanced features like pivot tables as in Excel. Compared to writing conventional code in a text code editor, Gneiss provided a visual environment that let participants see the actual data while modifying it, which helped them notice errors in their manipulations sooner. More details of the study are in [14].

9. Demonstrative examples

We now use a series of examples to demonstrate our tool's ability to create spreadsheet programs that use web services in a variety of ways.

9.1. City trip planner with a map

A common activity when a person plans to visit a new city is to search for all attractions in the city, pick the ones he is interested in, and plot them on a map. We will create a mashup that does this using a search interface that has a check box next to each return item for the user to select what he wants, and a map showing only the checked items. While many prior end-user mashup tools let people search for locations and plot them on a map using some built-in web services, none of the tools to our knowledge allows people to create such a flexible application that handles user selections and only shows the *selected* items on the map, without requiring the user to change the program every time when making a new query. We will show that a user, with some knowledge of regular spreadsheet formulas, can create such a data extraction program using Gneiss.

The user starts by using Google's Place Search API to get a list of attractions. She changes the value of the query parameter in the API to `{{A1}}` to bind it to cell A1. From the return data, she drags the name, latitude and longitude field to columns B to D. She decides to make column E the "input column" – if she likes a place and wants to plot it on the map, she enters a shorthand label for the place in that place's row in column F. Otherwise she leaves the cell blank. She enters `=IF(ISBLANK(E1), "", C1)` in cell F1. This formula sets F1 to be the latitude of the first place if E1 is not blank. Otherwise F1 is empty. The user then autofills column F using F1 to get the latitudes of all marked places. She uses the similar method to get the longitudes of all marked places in column G.

The user then creates a map visualization in the right pane by dragging it from the sidebar. She clicks on the map visualization and edits its properties from the property table in the lower right corner. She sets the "latitudes" property to be `=F:F`, and the "longitudes" property to be `=G:G`. The resulting program is shown in Fig. 4 at the top.

The created spreadsheet program is highly reusable. The user can search for another type of place by editing cell A1, or interactively add or remove a place from the map by entering or deleting contents in its corresponding cell in column E. None of these requires the user to change any programming logic in the spreadsheet. The user could easily share this city trip planner spreadsheet with her friends, with a little explanation about what the columns are and how to use them.

9.2. Real-time market price report

Gneiss' ability to periodically stream data from web services allows the user to create dynamic spreadsheet contents that change in real-time. For example, the user can create a spreadsheet that extracts data from a stock market price service, such as Yahoo Finance, and configures the system to fetch new data every 15 min. The spreadsheet then updates every 15 min and stores all historical data. The user can further sort and filter the data using temporal information, such as to view data in a certain time period. The user can also easily perform real-time data analysis, such as to use a line chart to see how the price changes (Fig. 4 at the bottom).

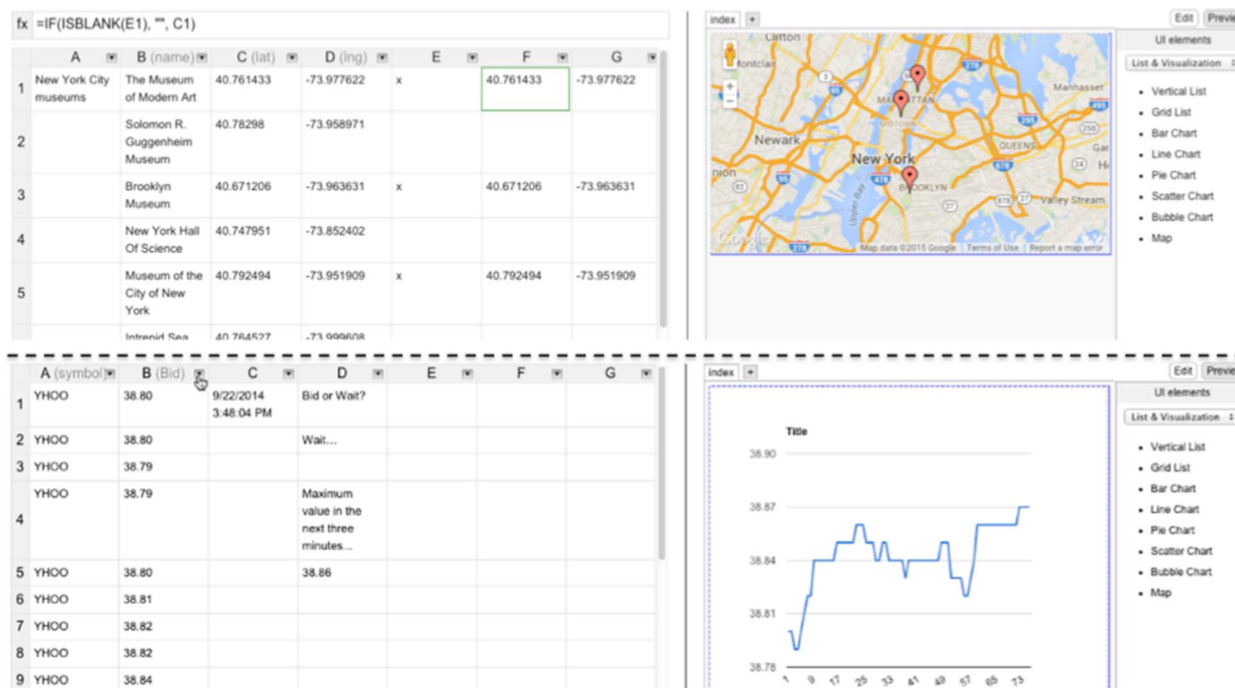

Fig. 4. Two example spreadsheets created with Gneiss. The top spreadsheet searches a place database based on the value in cell A1 and interactively plots places on a map based on the values in column E. The bottom spreadsheet streams Yahoo's stock prices periodically in column B and visualizes how the price changes over time using a line chart.

Please cite this article as: K.S.-P. Chang, B.A. Myers, Gneiss: spreadsheet programming using structured web service data, Journal of Visual Languages and Computing (2016), http://dx.doi.org/10.1016/j.jvlc.2016.07.004

ARTICLE IN PRESS

9.3. Co-author finder

Using a research paper API such as Mendeley, Gneiss is able to let the user create a data extraction program that searches all papers written by a person, for example, after 2010 and generate a list of co-authors of that person (a similar use case was described in a prior mashup paper [35]). To do so, the user first sends the API request and binds the value of the query parameter to a spreadsheet cell where she enters a person name. The user then drags the title, year, publication venue and authors of all papers to the spreadsheet columns A-D, and filters the data by year to only show papers published after 2010. For each paper, the "authors" field (column D) is an array that contains a list of authors. To get a list of distinct author names, the user first enters `=D:D` in E1 to fill in column E with all co-author names. She then filters column E to remove duplicate values and get a clean list. If the user wants, she can further use the co-author list to find more related papers by sending another web API request using the co-authors' names.

10. Conclusions

This paper presents Gneiss, a spreadsheet tool that assists people in using structured web service data. Gneiss demonstrates that the familiar spreadsheet metaphor can be extended to support constructing two-way data communication with web services and viewing and manipulating structured data extracted from complex hierarchical documents. Gneiss uses a client-server architecture to support streaming and using large amounts of data without slowing down the client machines. The implementation of the internal JSON objects not only enables users to easily reshape an object using drag-and-drop but also provides a way to manipulate the data using the dynamic, user-defined structures. It also introduces a new way for people to create data-driven applications by using spreadsheet languages to connect GUI elements in the application to a spreadsheet that works as a simple database storing data from multiple sources. Combining all these innovations together, Gneiss provides a live and visual programming environment for using web services and structured data and could benefit both professional programmers and end-user programmers who are interested in doing custom data analysis or building custom applications.

Acknowledgments

This research was funded in part by the NSF under Grants IIS-1116724 and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

References

[1] R.T. Fielding, R.N. Taylor, Principled Design of the Modern Web Architecture, ACM Trans. Internet Technol. 2 (2) (2002) 115–150.

[2] S. Oney, B. Myers, J. Brandt, ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States, in: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, 2012, pp. 229–238.

[3] N. Zang, M.B. Rosson, V. Nasser, Mashups: who? what? why? in: CHI'08 Extended Abstracts on Human Factors in Computing Systems, 2008, pp. 3171–3176.

[4] Yahoo Pipes [Online]. Available: 〈http://pipes.yahoo.com/〉.

[5] Microsoft PopFly [Online]. Available: 〈http://popfly.ms/〉.

[6] J. Cao, K. Rector, T.H. Park, S.D. Fleming, M. Burnett, S. Wiedenbeck, A Debugging Perspective on End-User Mashup Programming, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2010, pp. 149–156.

[7] S.K. Kuttal, Variation support for end users, in: Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2013, pp. 183–184.

[8] M. Burnett, J. Atwood, R. Walpole Djang, J. Reichwein, H. Gottfried, S. Yang, Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm, J. Funct. Program. 11 (2) (2001) 155–206.

[9] Microsoft Excel. [Online]. Available: 〈http://office.microsoft.com/en-us/excel/〉.

[10] Google Spreadsheet. [Online]. Available: 〈https://drive.google.com/〉.

[11] OpenRefine. [Online)]. Available: 〈http://openrefine.org/〉.

[12] K.S.-P. Chang, B.A. Myers, A spreadsheet model for using web service data, in: Proceedings of the 2014 IEEE Symposium onVisual Languages and Human-Centric Computing (VL/HCC), 2014, pp. 169–176.

[13] K.S.-P. Chang, B.A. Myers, A spreadsheet model for handling streaming data, in: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, 2015, pp. 3399–3402.

[14] K.S.-P. Chang, B.A. Myers, Using and exploring hierarchical data in spreadsheets, ACM CHI (2016).

[15] K.S.-P. Chang, B.A. Myers, Creating interactive web data applications with spreadsheets, in: Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology, 2014, pp. 87–96.

[16] B. Hartmann, L. Wu, K. Collins, S.R. Klemmer, Programming by a sample: rapidly creating web applications with d.mix, in: Proceedings of the 20th annual ACM symposium on User interface software and technology, 2007, pp. 241–250.

[17] J. Wong, J. I. Hong, Making mashups with marmite: towards end-user programming for the web, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2007, pp. 1435–1444.

[18] M. Van Kleek, D.A. Smith, H.S. Packer, J. Skinner, N.R. Shadbolt, Carpé data: supporting serendipitous data integration in personal information management, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2013, pp. 2339–2348.

[19] XPath [Online]. Available: 〈http://www.w3.org/TR/xpath/〉.

[20] E. Bakke, D.R. Karger, R.C. Miller, Automatic layout of structured hierarchical reports, IEEE Trans. Vis. Comput. Graph. 19 (12) (2013) 2586–2595.

[21] Z. Chen, M. Cafarella, Integrating spreadsheet data via accurate and low-effort extraction, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014, pp. 1126–1135.

[22] F. Hermans, M. Pinzger, A. van Deursen, Automatically extracting class diagrams from spreadsheets, in: Proceedings of the 24th European Conference on Object-oriented Programming, 2010, pp. 52–75.

[23] D.F. Huynh, R.C. Miller, D.R. Karger, Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities, in: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology, 2006, pp. 125–134.

[24] B. Chan, L. Wu, J. Talbot, M. Cammarano, P. Hanrahan, Vispedia: interactive visual exploration of wikipedia data via search-based integration, IEEE Trans. Vis. Comput. Graph. 14 (6) (2008) 1213–1220.

[25] Scraper" 2010. [Online]. Available: 〈http://mnmldave.github.io/scraper/〉.

[26] ScraperWiki 2014. [Online]. Available: 〈https://scraperwiki.com/〉.

[27] R. Tuchinda, P. Szekely, C.A. Knoblock, Building Mashups by example, in: Proceedings of the 13th international conference on Intelligent user interfaces, 2008, pp. 139–148.

[28] J. Lin, J. Wong, J. Nichols, A. Cypher, T.A. Lau, End-user programming of mashups with vegemite, in: Proceedings of the 14th international conference on Intelligent user interfaces, 2009, pp. 97–106.

[29] W. Kongdenfha, B. Benatallah, J. Vayssière, R. Saint-Paul, F. Casati, Rapid development of spreadsheet-based web mashups, in: Proceedings of the 18th International Conference on World Wide Web, 2009, pp. 851–860.

[30] P. Baglietto, F. Cosso, M. Fornasa, S. Mangiante, M. Maresca, A. Parodi, M. Stecca, Always-on distributed spreadsheet mashups, in: Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups, 2010, pp. 8:1–8:8.

[31] S.P. Jones, A. Blackwell, M. Burnett, A User-centred Approach to Functions in Excel, in: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, 2003, pp. 165–176.

[32] Z. Chen, M. Cafarella, Automatic Web Spreadsheet Data Extraction, in: Proceedings of the 3rd International Workshop on Semantic Search Over the Web, 2013, pp. 1:1–1:8.

[33] B. Johnson B. Shneiderman, Tree-maps: a space-filling approach to the visualization of hierarchical information structures, in: Proceedings of the IEEE Conference on Visualization, 1991 Visualization '91, 1991, pp. 284–291.

[34] K.S.-P. Chang, B.A. Myers, G. Cahill, S. Simanta, E. Morris, G. Lewis, A plug-in architecture for connecting to new data sources on mobile devices, in: IEEE Symposium on Visual Languages and Human-Centric Computing, 2013, pp. 51–58.

[35] S. Gardiner, A. Tomasic, J. Zimmerman, R. Aziz, K. Rivard, Mixer: mixed-initiative data retrieval and integration by example, in: Proceedings of the 13th IFIP TC 13 International Conference on Human-Computer Interaction - Volume Part I, 2011, pp. 426–443.

Please cite this article as: K.S.-P. Chang, B.A. Myers, Gneiss: spreadsheet programming using structured web service data, Journal of Visual Languages and Computing (2016), http://dx.doi.org/10.1016/j.jvlc.2016.07.004