

Creating Interactive Web Data Applications with Spreadsheets

Kerry Shih-Ping Chang and Brad A. Myers

Human-Computer Interaction Institute

Carnegie Mellon University

Pittsburgh, PA 15213 USA

{kerrychang, bam}@cs.cmu.edu

ABSTRACT

While more and more data are available through web services, it remains difficult for end-users to create web applications that make use of these data without having to write complex code. We present Gneiss, a live programming environment that extends the spreadsheet metaphor to support creating interactive web applications that dynamically use local and web data from multiple sources. Gneiss closely integrates a spreadsheet editor with a web interface builder to let users demonstrate bindings between properties of web GUI elements and cells in the spreadsheet while working with real web service data. The spreadsheet editor provides two-way connections to web services, to both visualize and retrieve different data based on the user input in the web interface. Gneiss achieves rich interactivity without the need for event-based programming by extending the “pull model” of formulas that is familiar to the spreadsheet users. We use a series of examples to demonstrate Gneiss’s ability to create a variety of interactive web data applications.

Author Keywords

Spreadsheets; web applications; web services; mashups; end-user programming; live programming

ACM Classification Keywords

D.2.2. Software engineering: Design tools and techniques – User interfaces; D.2.6. Software engineering: Programming environments – Graphical environments

INTRODUCTION

Many websites and mobile applications that people use daily are *web data applications* – applications that use backend data, supporting the searching, sorting, filtering and visualizing of the data based on the user input. For example, Yelp lets users search for restaurants and apply filters to show only the ones that accept credit cards. Expedia lets users search for hotels, sort the results by price, and

Permission to make digital or hard copies of all or part 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 the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

UIST '14, October 05 - 08 2014, Honolulu, HI, USA

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3069-5/14/10...\$15.00.

<http://dx.doi.org/10.1145/2642918.2647371>

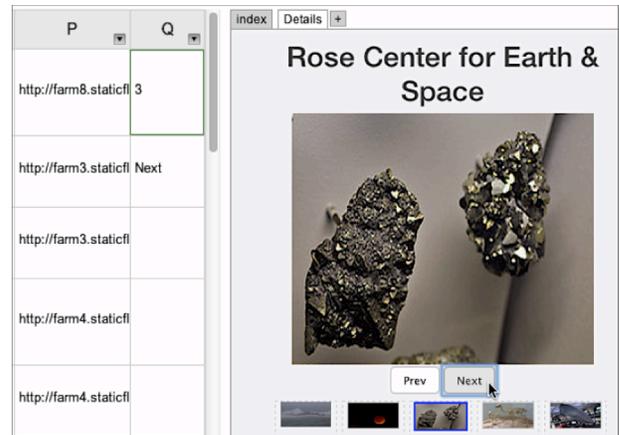


Figure 1: Gneiss lets people create interactive web applications in a spreadsheet editor. This is a screenshot of making a slideshow interface with photos from Flickr. At the left, column P stores a list of photo URLs, Q1 stores the current slideshow index (3), and Q2 stores the button that the user last clicked on (“Next”). At the right is the slideshow application. The thumbnails show the photos in column P. The large image is the photo at row 3 (the index value stored in Q1).

plot the results on a map. Creating this kind of application requires a person to program a responsive web interface that can dynamically manipulate backend data and create a display based on the results. This usually involves writing complex code that has multiple nested callbacks to handle events and perform the appropriate actions such as firing web service requests, retrieving new data, and repopulating the interface.

While most data sources provide a web interface and often a mobile application for accessing their data, the increasing demands of customized applications have led many companies to also provide web services that let people access the data programmatically. For example, Yelp and Expedia provide web services for restaurants and hotels. Flickr provides web services for photos (Figure 1). According to ProgrammableWeb.com, today there are over 10,000 web APIs available for use by the general public. Despite the popularity of web services, the programming skills required to incorporate them into a web application have limited their users to mostly professional programmers [24]. Prior studies found that end-users had the need to build web applications that access online databases but often failed to do so

[20]. Research on mashup tools has provided ways to let people extract and integrate web data from multiple sources without having to write conventional code [11,13,23], but it remains difficult for end-users to turn the collected data into anything more sophisticated than a table or a graph, let alone creating an interactive web application.

In this paper, we present Gneiss¹, a new tool that allows users to create interactive, data-driven web applications using the familiar spreadsheet language. Gneiss integrates a source panel for viewing and extracting web service data (Figure 2 at 1), a spreadsheet-like editor for manipulating the data and writing computational logic using spreadsheet formulas (Figure 2 at 2), and a web interface builder (Figure 2 at 3) for creating web interfaces. Like spreadsheets, Gneiss is a “live” programming environment, so new values are populated throughout the program and reflected in the output as soon as the user makes an edit. Gneiss uses a “programming-with-example” style [16], since example values are used while developing the program. These features allow Gneiss to achieve a different programming experience compared with conventional web programming tools where data are invisible and live changes are not supported. Gneiss is targeted at people who are already knowledgeable about spreadsheet programming and who know basic web design such as how to use an interactive editor.

A previous paper [6] presented how we extended conventional spreadsheets to work with web service data. In summary, the user can make a web API call and view the return data in the left panel (Figure 2 at 1), and use drag-and-drop to extract desired fields to the center spreadsheet (Figure 2 at 2). In the spreadsheet we use nested tables to preserve any structure in the extracted data. Data flow is two-way between the spreadsheet and the web service, as the user can bind parameters of the web API to spreadsheet cells.

This paper introduces how Gneiss allows users to create interactive web applications and graphical user interface (GUI) elements that visualize and control the data. By integrating with our previous work [6], data in the spreadsheet can be local data or web service data retrieved dynamically. Web application development with Gneiss utilizes the new right panel of the tool, which is a web interface builder where the user can drag a GUI element from the toolbar (Figure 2 at 4) and drop it to a desired location in the output pane (Figure 2 at 3). We introduce here two main extensions of the spreadsheet model to support interactive web applications. First, all GUI property values have all the capabilities of spreadsheet cells, and they can be referenced anywhere in Gneiss using a spreadsheet-like syntax. Similarly, the value of any GUI element property can be a con-

stant or a value computed using a spreadsheet formula that depends on other cells. This enables data flow to be two-way between the web interface and the spreadsheet editor. The user’s interactions with the web application could lead to changes in spreadsheet cells and further trigger different actions at run time such as to fire web service calls, retrieve new data, apply new sorting and filtering rules, and subsequently repopulate the web interface.

Second, while most spreadsheet tools do not allow circular references by default, our spreadsheet editor facilitates two-way communication by providing the “once-around” semantics of one-way constraint solvers, which has been shown to be useful in previous systems [14,15,18]. This, along with the dynamic properties of GUI elements that are triggered when the user interacts with the elements, and new formulas that control when a spreadsheet cell is re-evaluated, enable the user to program a wide range of interactive behaviors solely using a “pull model” that is consistent with the current spreadsheet formula evaluation paradigm. Gneiss eliminates the need for a conventional event-based model as is used by other GUI tools and toolkits. Some previous attempts to adapt spreadsheets for web programming have required an awkward hybrid of event-triggered (push) and formula-based (pull) execution [3,10], which Gneiss avoids.

Combining all these features, our work contributes a novel spreadsheet model for creating web applications that make interactive use of backend data. We evaluate Gneiss with a series of examples to demonstrate its ability to create a wide variety of web data applications without having to write anything more complicated than spreadsheet formulas.

In the rest of the paper, we first survey related work. We then describe Gneiss in detail and present the demonstrative examples. We conclude the paper with current limitations and future work.

RELATED WORK

Here we review related work on spreadsheets, mashups, and web programming tools for end-users.

Spreadsheet Tools and Programming Languages

Most conventional spreadsheet tools such as Google Spreadsheets and Microsoft Excel support one-way visualizations for spreadsheet data such as graphs and charts, but do not support building a web application where data flow between the web interface and the spreadsheet data is bi-directional. The idea of using the spreadsheet metaphor to support programming data-driven web applications was previously experimented in FAR [3]. FAR uses a table to present an e-commerce database and allows users to link a web interface to the database using spreadsheet syntax. Gneiss makes several novel contributions over FAR. First, while FAR only supports creating applications that connect to a single database, Gneiss allows users to create applications that use data from *multiple* local or online sources. Second, FAR uses special “query cells” to enable querying

¹ Gneiss (pronounced the same as “nice”) is a kind of rock. Here it stands for Gathering Novel End-user Internet Services using Spreadsheets. Gneiss is available from <http://www.cs.cmu.edu/~NatProg/gneiss>

a database but did not support other kinds of data actions such as sorting and filtering. In contrast, Gneiss enables all spreadsheet cells and GUI element properties to be used in formulas to perform searching, sorting, and filtering data and other common actions just like in regular spreadsheet tools. Finally, FAR uses the mix of the spreadsheet “pull” model and conventional callback (“push”) mechanisms to handle events. While convenient sometimes, this approach could potentially lead to bugs when the push and pull rules have conflicts and try to modify the same cell. In contrast, Gneiss’s language uses solely the “pull” model and is able to support more types of interactive behaviors, such as animations and dynamic pages, than using the hybrid method in FAR. Quilt [1] is a recent system that lets users use HTML attributes to connect a web page to a spreadsheet that serves as a backend database to fetch and store data. Our work uses a different approach of letting users use the spreadsheet language to create two-way bindings between GUI element attributes and spreadsheet cells. Our tool also supports creating interactive behaviors, and has a focus on using web data sources.

Another category of related work describes research tools that extend the spreadsheet metaphor to support other kinds of programming activities. While some of these support programming graphical or UI objects, none of them support creating applications that make interactive use of backend data. For example, C32 [17] uses a spreadsheet-like visualization for specifying user interface constraints, but did not address handling events and did not deal with data sources. NoPumpG [22] and Forms/3 [2] extend spreadsheet languages to help with building graphical interfaces and support animations and events. However, both NoPumpG and Forms/3 have unconventional spreadsheet designs that may not be intuitive for end-users. They also do not handle data sources and do not have the notion of a backend database. A1 [10] is a spreadsheet environment for programming system administrator tasks such as monitoring network usage. A1’s language is similar to conventional scripting languages and uses the callback mechanisms to handle events. It provides one-way graphical outputs such as graphs and charts, but does not support building arbitrary web interfaces with two-way connections to the spreadsheet editor.

End-user Mashup Tools

The majority of prior research on end-user mashup tools focuses on helping users extract or combine data from multiple web sources without writing conventional code. Many of these provide techniques to let users scrape web page data by demonstration such as novel ways to infer similar data using web page structure [9,21], or allowing users to automate the data scraping by recording a script [13]. Some tools support mashup data coming from a few web services hardwired into the tool (e.g., [11,23]). Gneiss currently supports *any* REST web service that returns JSON data but requires the user to show which parts of the data are desired using drag-and-drop [5]. In the future, we could extend Gneiss with graphical widgets for configuring some popular

built-in web services such as those in [23], to assist users who do not have knowledge about how to use web services.

While most mashup tools provide tables (e.g., [13,21,23]) and visualizations such as charts and maps (e.g., [11,23]) to display the collected data, very few of them support building web applications that provide interactive use of the collected data. d.mix [8] lets users copy desired functionality from example applications and recombine them in a wiki page to create their own web applications. However, this approach is limited by the number of example applications because it requires each example to be manually annotated by professional programmers. Building a web application in a wiki page also requires knowledge of HTML syntax and does not support creating interactive behaviors. Data flow language tool such as Yahoo! Pipes [25] can let users create input objects and wire them to data manipulation modules to accept runtime values. However, studies have found that the data flow representation is often difficult for end-users to understand [4]. Our work here extends the spreadsheet model, with which many end-users are already familiar [2]. Debugging may also be easier in our tool, as the data being manipulated are exposed in a spreadsheet and the user can see live changes in the data as she programs the application, in contrast to dataflow languages where the data are usually hidden unless the user specifically looks for them.

End-User Web Programming Tools

Web interface builders and WYSIWYG web editors have been widely used in commercial products and research projects to help end-users make web interfaces. However, connecting a web page using a regular interface builder (like Adobe Dreamweaver or Microsoft Visual Studio) to a data source and presenting dynamic content still require writing extensive code. Some research tools help people use examples to create static styles [5,12] or interactive behaviors [19] of a web page. But they do not help users understand or reproduce how the example web pages use and interact with the backend data sources.

USAGE SCENARIO

Here we describe a scenario where Ted, a college student and a spreadsheet user, uses Gneiss to create a web application that helps him decide where to visit in a city. The application has a textbox that searches an online place database and shows the search results in a grid list. The user can choose to sort the results by rating or price using two radio buttons. Each item in the grid list contains the name, rating, price level of a place, and a checkbox that lets the user display the place on a map. Finally, the user can click on a place name to go to a “details” page that shows photos of the place retrieved from an online photo database. Figure 2 is a screenshot of part of the end result (with the created web application in the right panel at (3)). We use this scenario to give an overview of our tool.

Ted first enters Google’s Place Text Search API in the URL box in the left panel (Figure 2 at 1). Hitting the refresh icon submits the query. The returned data, in JSON format, are

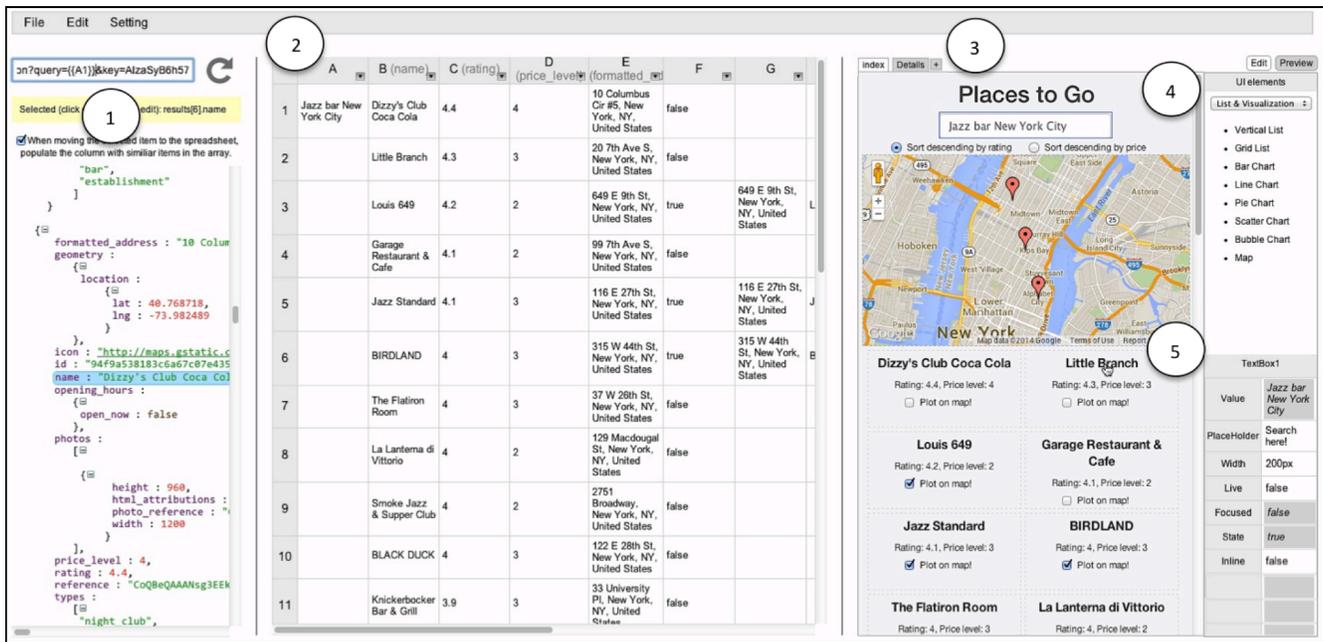


Figure 2. The Gneiss interface. (1) is the source panel where the user can load a web API in the URL box and extract desired fields from the return data to the spreadsheet editor through drag-and-drop. (2) is the spreadsheet editor that stores and manipulates the data to be used in the web application. (3) is the web interface builder where the user can create a web application by dragging-and-dropping GUI elements from the toolbar on the right (4) to the output page. The user can select a GUI element in the output page (the selected element is highlighted with a dark blue border, which currently is the textbox at the top of the page) and view its properties in (5). Property values are cells which can contain formulas and can be referenced by other cells.

shown below, and Ted can use this example data to demonstrate the next steps. After seeing that the API works with a constant string, Ted wants the value of the “query” parameter in the web API to be whatever the user types in the text box in the web application at runtime. To make the query string be dynamic, Ted first sets the value of the parameter to be the value of cell A1 in the spreadsheet, using the syntax `{{A1}}` (Figure 2 at 1, in the URL box). He then extracts the data he wants from the query result – the name, rating, price and address fields – to the spreadsheet in the center panel by drag-and-dropping each value into its own column (Figure 2 at 2, columns B-E). Because Ted has linked the query parameter in the web API to cell A1, every time cell A1’s value changes, the system will make a new API request with the new query value and update the extracted data with the latest returned data. This, in turn, will refresh the spreadsheet columns.

Ted does a few queries by changing cell A1’s value and makes sure everything works correctly. Because Gneiss is a “live” system, Ted can experiment with his code and view changes anytime. The spreadsheet shows data from Ted’s latest query, which Ted can later use as example values to continue developing his program. This “programming-with-examples” style has been shown effective in helping users understand and test their program as it develops [7].

Ted now starts to create the web application, using the interface builder shown in Figure 2 at 3. He first drags a text label (for the header) and a text box (for the query string)

from the tool bar at the right of the interface builder (Figure 2 at 4) to the output page. In the web interface builder, Ted can select any GUI element in the output page (which will then be highlighted using a blue border), and view and edit its properties, which are shown as a small one-column spreadsheet at the bottom right (Figure 2 at 5). Ted selects the heading text and changes its value to the constant “Places to Go”. Ted then selects the text box. The “value” property of the text box is a dynamic property depending on what the user enters at runtime (currently “Jazz bar New York City”). Back in the spreadsheet, Ted types `=TextBox1!Value` in cell A1 to make it be the text box’s value (we use this syntax to match current spreadsheets that use “!” when referencing cells in other worksheets or files). Now after Ted types in the text box and hits enter, cell A1 changes too and the system fires a new place search request based on A1’s value (it would alternatively be possible for A1 to update character-by-character by changing the “Live” property of the text box from “false to “true”, but that would not be appropriate for a web request).

Ted then drags in two radio buttons to control how the search results are sorted. Ted changes the label of the first radio button to “Sort descending by rating”. Then he clicks on the small arrow icon at the top of column C, the column that stores the ratings, to open up a dialog box (Figure 3) that controls the sorting and filtering of that column. Ted sets the sorting rule to come from a computed value, and enters the formula `=IF (RadioButton1!Checked, “Descending”, “None”)` to sort the results descending

by the rating column if the first radio button is checked. Ted uses the same method to set up the second radio button to control if the results are sorted by price (column D).

Ted continues to create the web application. He drags a grid list to the output page to show the search results. Ted drags two text labels and a checkbox to the first item of the grid list, and the system populates the rest of the grid items with corresponding UI objects. Ted wants the top text item in each grid item to be the name of the place. To do so, Ted selects the top text in the first grid item and changes its value to be =B1. The top text in each of the rest of the grid items automatically changes to be the value of the corresponding cell in column B. Ted uses the same method to change the second text to show the rating and the price level, and the label of the checkbox to be “Plot on map!”.

Ted next needs to collect the addresses of all the checked items in order to tell the system to plot them on the map. To do so, Ted first enters =CheckBox1!Checked in cell F1 to get whether the checkbox in the first grid item is checked. Ted then selects F1 and uses the standard spreadsheet mechanism to autofill down column F with the values of the “checked” property of each of the checkboxes. Now as Ted checks and unchecks a checkbox in the web interface, its corresponding cell in the spreadsheet column F changes its value too. In cell G1, Ted types in =IF(F1, E1, “”) and autofills the rest of column G. This formula fills in the cell in column G with the address cell (column E) in the same row if the checkbox cell in that row (column F) is true. Finally, in the web interface panel, Ted drags a map visualization from the toolbar to the output page, and sets the “addresses” parameter of the map to be all cells in column G by typing in =G:G (the standard spreadsheet syntax to reference a whole column). Now Ted has an interactive map that dynamically displays the checked places.

The last step to finish this web application is to allow the place’s name to be a hyperlink that goes to a new page that shows photos of the place from an online place database. This demonstrates how Ted can use Gneiss to combine data from *multiple* data sources. To do so, Ted creates a new page called “Details”, and sets the “link” property of the place name text to be “Details”. This causes the Details page to be brought to the front when a place name text is clicked. Ted then needs to use the latest clicked place name in order to fire a different API request to retrieve the photos for that place, and fill in the details page. In Gneiss, all UI objects have a “state” property that reflects how the mouse cursor interacts with it. For example, when the mouse moves over the text object and then the user clicks on the text, the state for the text changes from “idle” to “hovered” to momentarily be “clicked”. Ted autofills column I with the state property of the corresponding place name text. Then in J1 he types =IF(COUNTIF(I:I, "clicked")>0, LOOKUP("clicked", I:I, B:B), J1), which uses the standard spreadsheet functions COUNTIF and LOOKUP. This formula will count how many cells in

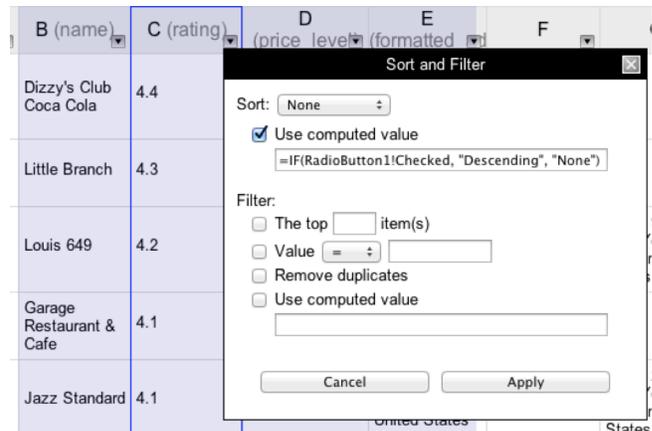


Figure 3. To sort and filter spreadsheet data by a column, the user can click on the arrow button at the top of the column to bring up a dialog box. Our tool allows sorting and filtering to be done by computed values that use other spreadsheet cells or GUI element properties. The system highlights the column it sorts and filters by with a blue border, and all columns sorted and filtered together in a purple background.

column I have value “clicked”. If there are any, then J1 becomes the cell in column B (the name column) that is in the same row with the “clicked” cell in column I. Otherwise the formula does not change J1’s value. This makes J1 become the name of the most recently clicked place. Ted then uses Flickr’s Photo Search API and cell J1 as the query string to search for the photos of the clicked place, and uses similar methods as before to extract the photo data returned by Flickr and show the photos using a grid list in the web interface builder (Figure 1). Lastly, Ted adds a “Back” button to the Details page to let the user go back to the search page. Note that values are passed around and interactions are triggered using standard spreadsheet functions and formulas, without the need for any event-based programming.

Ted now has finished creating the application! He plays with the application in preview mode, does a few test queries, and exports the application when done. Gneiss generates a URL that can be used to open this application anywhere in a browser from a laptop or a smartphone. Ted sends this URL to his friends to share the application.

KEY FEATURES

Here we described the key features of our system. We omit describing regular spreadsheet features and interface builder features since they are not the contribution of this work or were covered in [6].

Interacting with Web Services

The complete features and implementation details of how Gneiss allows users to interactively make web service requests and extract data were described in a previous paper [6]. Here we give a quick overview of some relevant features. Gneiss currently supports REST web services that return JSON data, the most popular type of web APIs today according to ProgrammableWeb.com. RESTful web APIs

are HTTP URLs that contain a base URI and a set API parameters provided as name-value pairs. The user can enter a web API in the URL box in the source panel (Figure 2 at 1). The desired fields from the returned data can be copied to the spreadsheet editor for further manipulation by selecting a field (the selected field will be highlighted in a light blue background as in Figure 2) and dragging it to the spreadsheet editor. By default, if the value is from a list, the system will extract all of the corresponding fields and fill in the rest of the column (this can be turned off using the checkbox in Figure 2 at 1). For example, in the usage scenario, to extract the names of all of the places returned, Ted only needs to drag the first place name in the search results to cell B1. The system will automatically extract the names of all other places and put them in the rest of the cells in column B. The bottom of column B is grayed out to indicate that this column is automatically maintained and may change lengths (for example, if a future query returns a different number of values), so the user should not try to use any part of the column for other purposes.

Gneiss also allows data in the spreadsheet to flow back to the web service, by letting users replace any part of the web API with a reference to a spreadsheet cell wrapped in double braces. For example, in the usage scenario, Ted makes the value of the query parameter in web API be `{{A1}}` to let the value of cell A1 be sent to the web service as the query string. The double brackets syntax is adopted from JavaScript template libraries (such as [26]) to bind expressions to HTML templates. When the cell value changes, Gneiss will fire the API call again using the cell's computed value, and then any spreadsheet cells depending on the returned data will be updated in turn. For example, in the usage scenario, if A1 changes to "museums in D.C.", the system will send a new API request using "museums in D.C." as query string, and update columns B-E in the spreadsheet with the new returned data.

Other innovative data extraction features in Gneiss, such as using nested tables to preserve the structure of the extracted data and allowing independent data extractions to run in parallel, are described in detail elsewhere [6]. Our work in this paper focuses on creating interactive web applications that present and control the dynamic backend data.

Creating a Dynamic UI using Spreadsheet Data

As in many commercial web interface builders, our tool lets users create UI elements in a web page by drag-and-drop. Gneiss currently supports text and image objects, input elements (such as text boxes and buttons), lists, and visualizations, with the full set of HTML5 elements, properties and GUI controls planned for the future. In general, the properties of these elements include string properties (e.g., the text in a heading, the label of a button), styling properties (e.g., color and width), link properties, and interactive properties that change values as the user interacts with the element. String and styling properties let users style GUI elements using conventional mechanisms. A Link property turns a

GUI element into a hyperlink that goes to the value of the property. The value can be a URL or the name of another page opened in the interface builder, allowing the user to create a multi-page application. The user can specify how the application transfers from one page to another, and how the contents of one page may depend on values from another, as described in the usage scenario.

In Gneiss, the user can edit a GUI element property just like editing a cell in the spreadsheet editor. A GUI element property can be a constant or a dynamic value computed from the spreadsheet data. Setting a property value to use spreadsheet cells will cause the property be recomputed every time that these cells change values. For example, if the user sets the "value" property of a text object to be `=CONCATENATE("Rating: ", C1)`, then each time that cell C1 in the spreadsheet editor changes, the text object will also change to show the latest value.

Dynamic Lists

Gneiss currently supports vertical lists and grid lists. Lists have a "Populate" property that if true will fill in all items of the list with the corresponding elements based on edits to the first item. For example, in the usage scenario, Ted only needs to drag two text labels and a checkbox into the first item of the grid list, and the rest of the items in the list are populated with the same objects as in the first item. Property values of a UI element in a list also populate in the same way, so changing the property of a UI element in the first item of the list will affect all of the corresponding UI elements in the rest of the items of the list. If the property value is a constant, all other UI elements are populated with the same value. If the property value is a formula that computes the value based on some cells in the spreadsheet, the system will populate other UI elements in the list with the corresponding cells in the same column but using the row based in the element's index. For example, in the usage scenario, Ted sets the value of the top text label in the first item of the grid list to be `=B1` (the name of the first place). The system automatically populates the top text label in the second list item to be `=B2`, and so on.

Items in a list can be shown and hidden dynamically at run time based on the spreadsheet data if the "NumberOfItems" property in the list object is set to "auto" (the default). The system will automatically adjust the number based on the number of non-empty rows in the column that the UI element's value depends on. For example, in the usage scenario, a different query can return a different number of places. The length of the grid list will dynamically adjust to fit the length of search results.

Alternative, the user can also set "NumberOfItems" to be a number and give the list a constant length or a length computed by a formula. The user can also set "Populate" to false. In this case, the list object becomes a pure layout object and the user can manually put different UI elements in different list items.

Two-way Data Flow

In Gneiss, user inputs into the user interface can also affect the spreadsheet data, making the data flow two-way. The user can reference any property of a UI element using the syntax `ElementID!PropertyName`, using a syntax similar to how a cell in another worksheet is referenced in conventional spreadsheets. References like this can be used not only in the spreadsheet editor but also in the GUI property sheets in the web interface builder as well. For example, the user can set `Text2`'s value property to be `=Text1!Value`, making the two text objects display the same content. We further added a convenient keyword `THIS` to let a GUI element property reference other properties in the same element using `THIS!PropertyName`.

“Autofilling” GUI Property Values

A common feature in spreadsheet tools is “autofilling” where the user selects one or multiple cells and drags a box in the lower right corner of the selection to fill in additional cells with corresponding values. We extend this feature to facilitate referencing properties of populated GUI elements in a list object. The user only needs to enter a reference in a cell for the desired property of the UI element in the first list item, and then select that cell and autofill down. For example, in the usage scenario, Ted types `Checkbox1!Checked` in cell F1, selects F1 and drags to F10. The system then fills in F2 to F10 with the “Checked” property of the checkboxes in the second to tenth items in the grid list.

Making Things Interactive

One of the most important innovations in Gneiss is the way that users can make their web applications interactive. Originally, we explored having Gneiss use a conventional event-based or callback architecture like Java and JavaScript, where UI elements would contain actions to be performed when operated. However, it is awkward to combine these “push” actions (where the action routine in a UI element would set other cells—*pushing* values to them) with the spreadsheet “pull” model (where cells compute their own values with formulas by *pulling* in the needed values). Therefore, we designed a novel way for the Gneiss user to define interactive behaviors without ever needing to write any callback or event procedures.

Instead, we designed interactive properties of GUI elements that change values at run time based on how the user interacts with them. Interactive properties can be used in the spreadsheet cells to compute different data based on the user's action. All GUI elements have an interactive property called “State” that shows how the mouse cursor interacts with them. Possible values for “State” include “idle”, “hovered”, “pressed” and “clicked”. The “clicked” value stays for a few extra milliseconds after the click event so the user can notice that it happened. Other interactive properties are mostly for input elements, such as “Value” for text boxes and sliders, and “Checked” for radio buttons and checkboxes. Interactive properties are not editable from the property sheet since their values are controlled at runtime by the

widgets, so their cells are shown using a grey background (Figure 2 at 5). In both the spreadsheet editor and the property sheet, values of interactive properties change dynamically in keeping with Gneiss's “programming-with-example” style.

To further enable more types of interactive effects to be created in our tool, we designed a timer formula `TIMER(ms, condition)` that starts a timer ticking every “ms” milliseconds if “condition” is true, and turns off the timer if “condition” is false. The formula returns true for one millisecond when the timer ticks, and false otherwise. The timer formula was designed specifically to follow Gneiss' “pull” programming paradigm and when used together with conventional spreadsheet formulas such as `IF` would enable timed transitions and animations in the spreadsheet, as demonstrated below.

Constraint Evaluation and Circular Constraints

By default when a spreadsheet cell's value changes, the cell will invalidate all other cells that directly depend on it. This causes the dependent cells to re-compute their values. If the value of that cell changes, then the cell continues to invalidate other cells that directly depend on it, and so on. Otherwise (if the value stays the same), the cell will not invalidate its dependent cells, and the propagation stops. This increases the system's performance and also ensures that web API requests are not run when a parameter value is updated to the same value as before. This tends to happen with constraints that contain `IF`s or other control structures, where the constraint must be re-evaluated, but ends up calculating the same value.

In rare cases, the user might want to re-evaluate the constraints that depend on a cell even if the cell's value does not change. For example, if the spreadsheet data are extracted from a news web service, the user might want to re-run the same query to get the latest news. We provide a formula `REFRESH(exp)` that when used in a cell will return the value of `exp` and invalidate all the children cells (causing them to be re-evaluated) even if `exp` is the same as before. So for instance, the user can set a cell that stores the query parameter to be `=IF(TIMER(600000, B1), REFRESH(“world cup”), “world cup”)` to get news about “world cup” every 10 minutes when B1 is set to true. Although the query cell's value is always “world cup”, the `REFRESH` formula invalidates all constraints that depend on the cell, causing the web service request to be triggered and refresh the spreadsheet columns in turn.

Like some other one-way constraint solvers [14,15,18] our spreadsheet formula solver provides “once-around” semantics for circular constraints. We use an example to explain how this works. Suppose the user sets A1 to be 0 and A2 to be $A1+1$, which makes A2 be 1. The user then sets A1 to be $A2-1$, which creates a circular reference. Most spreadsheet tools will return errors in both A1 and A2. In contrast, our system allows circular references, and always evaluates any

dependent cells exactly once when any cell in the cycle changes. Here, the system starts computing A1 by asking A2's value. Since A2 is $A1+1$, it goes back to A1 and finds that it has reached the beginning of the cycle. The system then stops the circular reference here and returns A1's old computed value, 0. That makes A2's value 1 ($0+1$), and A1's value stays 0 ($1-1$).

Supporting "once-around" circular constraints in the spreadsheet makes more types of expressions possible. For example, a spreadsheet cell can now reference itself. The system will return the cell's original value before it is recomputed. Combined with the IF statement, this allows the user to set a cell's value to something if a condition is true, otherwise having the cell *retain its original value*. This expression is useful to handle interactive behaviors as it allows the user to set a cell's value when a GUI element enters a certain state, but does not change the value back when the element enters another state. For example, suppose the user wants to change a hyperlink text to be another color once it is pressed. Even though the "State" of the text goes back to "idle" from "pressed" after the user releases the mouse button, using a self-reference, the text is able to stay the same color. Another use case of self-reference is that it allows a cell to compute its new value using its previous value. Used together with the TIMER formula, the user can set a cell to steadily increase or decrease its value, and bind the cell to a GUI element property to create an animation, such as expanding or shrinking the size of an element, or binding to a color property to create a fade-in/out effect. We provide more examples in the demonstrative examples section.

Dynamic Sorting and Filtering

An innovation in Gneiss is generalizing the sorting and filtering of the spreadsheet data to allow users to control this with a dynamic value computed using spreadsheet cells or GUI element properties. To sort or filter data in a column, the user can click on the small arrow button at the top of the column to open a dialog box (Figure 3). In the dialog box, the user can choose to apply constant rules for sorting and filtering as in conventional spreadsheets, or to apply a dynamic rule by checking the "Use computed value" checkbox and entering the rule in the corresponding textbox. This rule can use conventional spreadsheet functions, with which the user would already be familiar, and should return the same string values as shown in the popup menu. For example, for sorting, the computed value should be "Ascending", "Descending" or "None". Thus, entering `IF(A1>5, "Descending", "None")` as the rule will sort the column only when A1 is bigger than 5.

For filtering, the computed value should be a comma-separated string with the first item being the filtering method ("FilterTop", "FilterValue" or "FilterDuplicates") and the second item being the argument required by the method (except for "FilterDuplicate", which has no argument). For

example, entering `CONCATENATE("FilterValue, >=", A1)` as the rule will filter out cells that are less than A1.

Unlike regular spreadsheet tools where sorting and filtering data is a one-time action, our tool applies sorting and filtering rules to a column every time that a cell in the column changes its value. As described in depth in our previous paper [6] all of the columns corresponding to the same web query are grouped, so that sorting and filtering applies to all grouped columns equally. Thus, in the usage scenario, sorting on column C or D causes all of the columns B-E to be sorted together. If multiple columns in a group have sorting on, then the leftmost column is performed first, then any items that are identical are sorted by the next column, etc. (A more flexible approach with a user-defined order as in Excel is planned for the future.) When all columns in a group have sorting off, then the columns revert to the default order (as originally returned by the web service).

IMPLEMENTATION

The complete Gneiss system is implemented as a web application. We use ConstraintJS [18] to maintain the dependencies among the spreadsheet cells and GUI elements. We use Node.js as the backend server to send web service requests and host the user-created web applications.

DEMONSTRATIVE EXAMPLES

Here we use four more examples to demonstrate our system's ability to build elaborate interactive interfaces and support different types of web applications.

Photo Slideshow

Slideshow is a common UI design to show a list of photos. It usually contains a large photo at the center of the interface, a "previous" and "next" button that let the user change the photo displayed by going up and down the list, and thumbnails of all photos at the bottom of the interface. We will show here how the user can create a photo slideshow interface using our tool and regular spreadsheet formulas. The screenshot of the created application is in Figure 1.

First, the user would retrieve a list of URLs of photos using a photo web service (such as Flickr), in the same way as described in the usage scenario. Suppose the user filters the search results to see the first 5 photos and stores the photo URLs in cell B1 to B5. In the web interface builder, the user can drag an image object to the output page and two buttons before and after the image, changing the label of the buttons to be "Previous" and "Next". The previous and next buttons control the index of the photo being shown. The user decides to store the index of the current photo in cell A1. She types in "0" in A1 first. She then needs know if the "Previous" button or the "Next" button gets clicked. To do so, she types in A2:

```
=IF(Button1!State="clicked", "Previous",  
IF(Button2!State="clicked", "Next", ""))
```

where Button1 is the previous button and Button2 the next button. This nested IF formula changes A2 to be set to either "Previous" or "Next" when a button is clicked. Now

the user needs to add or subtract the index cell (A1) by one based on what button is clicked. She changes A1 to be:

```
=IF(AND(A2="Previous", A1>1), A1-1,  
IF(AND(A2="Next", A1<10), A1+1, A1))
```

This nested IF formula not only updates A1's value based on what button is clicked, but also makes sure it stays between 1-10, as the photo URLs are in rows 1 to 10 of column B. (To alternatively make the buttons wrap around would be a simple change to this formula.) From the property sheet in the web interface builder, the user sets the "Source" property of the center image object to =INDEX(B:B, A1). INDEX is a standard spreadsheet function that takes an array of cells (the first argument) and returns the cell at the given index (the second argument). The center image object now displays the photo at the index specified in A1, and will change when the user presses the previous or next buttons.

Lastly, the user has to create a list of thumbnails of all the photos. To do so, she drags a grid list to the bottom of the interface, drags an image object to the first item of the grid list, and set the "Source" property of that image object to be =B1. The system then populates the list to show photos from B2 to B5. The user wants the thumbnail displayed to as the center image to be highlighted with a blue border. To do so, the user enters the following formula in the "Border" property of the thumbnail image:

```
=IF(This!Source=Image1!Source, "solid blue  
1px", "none")
```

where Image1 is the center image. The formula sets the border of a thumbnail to be blue when the source of the thumbnail is the source of the center image. The user now has finished creating a web application that shows photos in a slideshow interface, using only four spreadsheet formulas!

Animations

Using the TIMER function, the user can create animations with our spreadsheet tool. Continuing the photo slideshow example, suppose the user wants the width of the image to slowly increase from 30 pixels to 60 pixels when the mouse enters, and to go back to 30 pixels when the mouse leaves. She first sets spreadsheet cell C1 to be 30 and sets the "Width" property of the thumbnail image to be C1. She then types the following formula in C1:

```
=IF(Image2!State="idle", IF(TIMER(15,  
C1>30), C1-1, C1), IF(TIMER(15, C1<60),  
C1+1, C1))
```

where Image2 is the thumbnail image. When the image is in the idle state, this formula subtracts one from C1 every 15 milliseconds until it becomes less than or equal to 30. Otherwise, when not idle, the formula adds one to C1 every 15 milliseconds until it becomes more than or equal to 60. The user can then select C1 and drag it down to autofill C2 to C5, and apply this behavior to all five thumbnails in the output page.

Posting Data

We have shown our system's ability to create applications that pull data from a web data source. Our system can also create application that *posts* data back to a web data source using a POST web API. Suppose the user wants to create an application where she can type in a textbox and press a "Send" button to send the data to a web data source. The user starts by creating a text box and a button in the output page. Then in spreadsheet cell A1, she types: =IF(Button1!State="clicked", TextBox1!Value, A1) to fill A1 with the text box value when the button is clicked. She can then bind A1 to the value of the data parameter of the POST web API, which will send it to the data source.

Customized Personal Activity Visualization

The increased usage of personal wellness applications has made more personal information available on the cloud through web services. For example, Fitbit's web service lets a user access her recent food records, physical activities and sleeping hours programmatically. While these data sources might have their own interfaces to present the data, sometimes the user might want to create customized interfaces to show the data, or combine data from multiple sources. For example, Fitbit's mobile application shows the physical activities and sleeping hours in different tabs, but the user might want to compare them together to see if working out seems to help her sleep. Gneiss provides a set of popular charts and visualizations. For example, the standard spreadsheet formula NOW() can be used to get the current date, and the API can be used to retrieve her sleeping hours and exercising hours in the past seven days, which can be visualized using the built-in scatter plot.

LIMITATIONS

Gneiss targets users who are familiar with spreadsheet programming and know basic web programming. The examples above show that advanced spreadsheet knowledge may be needed to achieve some desired outcomes. If the user wants to connect an application to a web data source, she must also have basic knowledge about RESTful web APIs such as their format. Gneiss currently only supports REST web services that return JSON data, although we believe the same interaction techniques could be applied to other kinds of web data such as XML or CSV data. Gneiss currently does not support web services that require authentication. Currently, the value of an input element only changes when the application user interacts with it (such as typing something in the textbox). It cannot be edited programmatically from the spreadsheet or initialized based on data. In the future, we plan to make it possible for the developer to set the value of an input element using formulas, such as specifying the default text for a textbox. Gneiss currently supports basic GUI elements and mouse events. We plan to extend Gneiss to support more HTML5 elements, properties and events such as "drag" and "swipe" for touch devices. We believe that Gneiss's pull-based language could adapt to these new elements and events with minimum changes.

FUTURE WORK

In addition to the extensions mentioned above, we are interested in augmenting current spreadsheet interface to better support programming activities. We are also interested in extending spreadsheets to handle time-series data such as data from streaming web services or mobile sensors. We plan to run iterative usability evaluations to refine Gneiss's user interface, and a summative study to evaluate how well end-user programmers with various levels of programming ability can use Gneiss.

CONCLUSIONS

Gneiss provides a novel spreadsheet model for creating interactive web data applications. It allows properties of GUI elements to use spreadsheet data as their values and to be referenced by other cells. We designed new functions to support timers and to capture user events using dynamic GUI element properties that signal when the user interacts with them. Supporting once-around circular references in the spreadsheet editor enables a variety of interactive behaviors to be programmed using a "pull model" that is consistent with the current spreadsheet paradigm. As shown by the various examples, Gneiss can be used to program a wide variety of web applications using a wide variety of web service data sources.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their valuable comments. 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. Benson, E., Zhang, A., and Karger, D.R. Spreadsheet-Driven Web Applications. *Proc. UIST*, ACM (2014)
2. Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H., and Yang, S. Forms/3: A First-order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *J. Funct. Program.* 11, 2, Cambridge University Press (2001), 155–206.
3. Burnett, M., Chekka, S.K., and Pandey, R. FAR: an end-user language to support cottage e-services. *Proc. HCC, IEEE* (2001), 195–202.
4. Cao, J., Rector, K., Park, T.H., Fleming, S.D., Burnett, M., and Wiedenbeck, S. A Debugging Perspective on End-User Mashup Programming. *Proc. VL/HCC, IEEE* (2010), 149–156.
5. Chang, K.S.-P. and Myers, B.A. WebCrystal: Understanding and Reusing Examples in Web Authoring. *Proc. CHI*, ACM (2012), 3205–3214.
6. Chang, K.S.-P. and Myers, B.A. A Spreadsheet Model For Using Web Service Data. *Proc. VL/HCC, IEEE* (2014).
7. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, MA, 1993.
8. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a sample: rapidly creating web applications with d.mix. *Proc. UIST*, ACM (2007), 241–250.
9. Huynh, D.F., Miller, R.C., and Karger, D.R. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. *Proc. UIST*, ACM (2006), 125–134.
10. Kandogan, E., Haber, E., Barrett, R., Cypher, A., Maglio, P., and Zhao, H. A1: End-user Programming for Web-based System Administration. *Proc. UIST*, ACM (2005), 211–220.
11. Van Kleek, M., Smith, D.A., Packer, H.S., Skinner, J., and Shadbolt, N.R. Carpe Data: Supporting Serendipitous Data Integration in Personal Information Management. *Proc. CHI*, ACM (2013), 2339–2348.
12. Kumar, R., Talton, J.O., Ahmad, S., and Klemmer, S.R. Bricolage: Example-based Retargeting for Web Design. *Proc. CHI*, ACM (2011), 2197–2206.
13. Lin, J., Wong, J., Nichols, J., Cypher, A., and Lau, T.A. End-user programming of mashups with vegemite. *Proc. IUI*, ACM (2009), 97–106.
14. Myers, B.A., Giuse, D.A., Dannenberg, R.B., et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer* 23, 11, IEEE (1990), 71–85.
15. Myers, B.A., McDaniel, R.G., Miller, R.C., et al. The Amulet Environment: New Models for Effective User Interface Software Development. *Trans. Softw. Eng.* 23, 6, IEEE (1997), 347–365.
16. Myers, B.A. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *Proc. CHI*, ACM (1986), 59–66.
17. Myers, B.A. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. *Proc. CHI*, ACM (1991), 243–249.
18. Oney, S., Myers, B., and Brandt, J. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. *Proc. UIST*, ACM (2012), 229–238.
19. Oney, S. and Myers, B. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. *Proc. VL/HCC, IEEE* (2009), 105–108.
20. Rosson, M.B., Ballin, J., and Rode, J. Who, what, and how: a survey of informal and professional Web developers. *Proc. VL/HCC, IEEE* (2005), 199–206.
21. Tuchinda, R., Szekely, P., and Knoblock, C.A. Building Mashups by example. *Proc. IUI*, ACM (2008), 139–148
22. Wilde, N. and Lewis, C. Spreadsheet-based Interactive Graphics: From Prototype to Tool. *Proc. CHI*, ACM (1990), 153–160.
23. Wong, J. and Hong, J.I. Making mashups with marmite: towards end-user programming for the web. *Proc. CHI*, ACM (2007), 1435–1444.
24. Zang, N., Rosson, M.B., and Nasser, V. Mashups: who? what? why? *CHI'08 Extended Abstracts*, ACM (2008), 3171–3176.
25. Yahoo Pipes. <http://pipes.yahoo.com/>.
26. Handlebar.js. <http://handlebarsjs.com/>.