Spreadsheet-Driven Web Applications

Edward Benson
MIT CSAIL
32 Vassar St., Cambridge MA
eob@csail.mit.edu

Amy X. Zhang
MIT CSAIL
32 Vassar St., Cambridge MA
axz@csail.mit.edu

David R. Karger
MIT CSAIL
32 Vassar St., Cambridge MA
karger@mit.edu

ABSTRACT

Creating and publishing read-write-compute web applications requires programming skills beyond what most end users possess. But many end users know how to make spreadsheets that act as simple information management applications, often with computation. We present a system for creating basic web applications using such spreadsheets in place of a server and using HTML to describe the client UI. Authors connect the two by placing spreadsheet references inside HTML attributes. Data computation is provided by spreadsheet formulas. The result is a reactive read-write-compute web page without a single line of Javascript code. Nearly all of the fifteen HTML novices we studied were able to connect HTML to spreadsheets using our method with minimal instruction. We draw conclusions from their experience and discuss future extensions to this programming model.

Author Keywords

Web design; Spreadsheets; End-user programming; Information architecture

ACM Classification Keywords

H.5.4. Information Interfaces and Presentation: Hypertext/Hypermedia - User Issues

INTRODUCTION

Spreadsheets are ubiquitous information management tools that cross technical and cultural divides. College students use them to plan parties, financial analysts use them to sway markets, and many companies use spreadsheets instead of databases to manage data [28]. According to the 2012 United States Census, over 55 million people in the United States "manipulate data with either spreadsheets or databases at work [26]." But while these authors use spreadsheets for a diversity of information management tasks, they have limited ability to view, edit, and share their data outside the grid-based interface spreadsheets provide. Showing a stylized schedule for an event planned with a spreadsheet, for example, typically requires copying and pasting raw data from the spreadsheet into the finalized design document.

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 2014, October 5–8, 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/10.1145/2642918.2647387

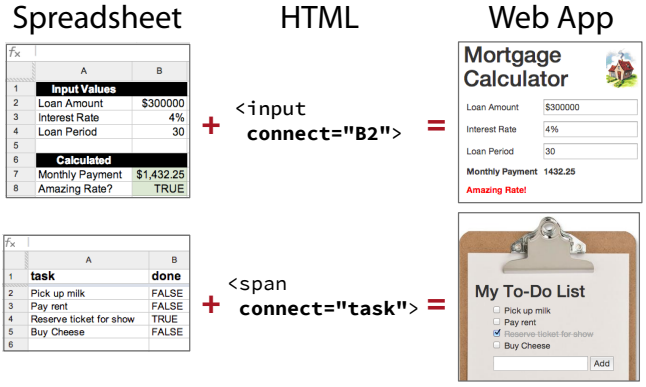


Figure 1: Our user study participants used Quilt to author the connective code to turn these HTML files and spreadsheets into reactive apps after a brief tutorial.

There is a separate overlapping community of people who know how to edit HTML. The size of this community is difficult to assess, but web literacy is undoubtedly growing. The ACM Computer Science Standards cite HTML and CSS authoring as basic components of computer literacy at the K-12 level [11]. Web authors build pages for a variety of purposes, ranging from personal to public to collaborative. Like spreadsheet authors, web authors face a steep hurdle once they try to move beyond the basic capabilities HTML offers. To create a web page with data storage and computation, an HTML author must learn to write client-server programs and possibly to design and administer databases.

In both cases, users have access to a tool that is specialized for one type of information-centric activity, such as data management or hypertext publishing, but does not natively support other activities. Past approaches to these two problems either focus heavily on providing pre-made widgets or templates [14] [27] or use new authoring and deployment environments that replace the user's existing ones [29] [22].

This paper stems from the intuition that spreadsheets and HTML are complimentary technologies waiting to be joined. Spreadsheet users enjoy structured data and computation but lack the ability to view and edit data in whatever flexible, stylized fashion suits them. HTML authors can create such styled displays, but studies show they both lack and greatly desire the ability to perform information management tasks such as data collection, storage, and retrieval [22] [24] [6], exactly what spreadsheets can step in to provide.

This paper presents and evaluates a system called Quilt[1].

[1]Source code is available at `treesheets.org` and a deployed version ready for use is at `cloudstitch.io`

Using Quilt, end users with basic spreadsheet and HTML-editing capabilities can skin their spreadsheets with a web page that serves as a more stylized alternative to the spreadsheet interface. From the web developer's perspective, Quilt enables rapid prototyping of reactive information management applications. Quilt uses relational annotations in HTML that do not require the author to understand dataflow. Using a tree synchronization algorithm, Quilt provides a reactive web experience based on these annotations. We show that simple information management applications, complete with sorting, searching, and table joins, can be crafted with Quilt, and demonstrate how inter-document formulas and spreadsheet scoping can support multi-user applications.

We then report results from a user study of 15 end users with basic HTML editing ability. After learning our method for fifteen minutes, they performed a series of authoring tasks: (1) connecting spreadsheets to web pages when provided both and (2) authoring (and then connecting) spreadsheets from scratch when provided only an HTML UI mockup. Participants were nearly all able to complete the tasks successfully in only a few minutes. We use videos, surveys, and interviews from this user study to evaluate Quilt's learnability and reflect on debugging with this method of authoring.

QUILT

We begin with a brief section to explain the Quilt language and operations. Assume a sample task of building a web interface to collaboratively plan a party with friends. The following spreadsheet contains a list of food items, their costs, the person responsible, and a computed cell for total cost:

	A	B	C	D
1	**Item**	**Cost**	**Person**	
2	Cheese	$15.00	Casey	total
3	Crackers	$5.00	?	$35.00
4	Wine	$15.00	?	

Quilt reactively connects an HTML web interface to the spreadsheet using four new HTML attributes: `connect`, `delete`, `show-if`, and `hide-if`. Depending on where they are used, these attributes can synchronize data between the web page and spreadsheet, add new rows, delete rows, and show/hide web elements depending on spreadsheet contents.

The following code snippet and output demonstrates how to connect HTML elements to spreadsheet cells. Cells can contain either plain text or computed values. In both cases, changes made on the spreadsheet flow automatically to the web page, replacing the element content. Changes to the web element content, through direct user action or Javascript manipulation, propagate automatically back to the spreadsheet.

```
<span connect="D3">cost</span>
```
→ $35.00

By placing the attribute `connect="rows"`, Quilt users can designate that an HTML element's contents should repeat once per row in the spreadsheet. Each repetition corresponds to the data in a row. Inside a repetition, one can `connect` HTML elements using the names of columns, or offer row deletion by placing `delete="row"` on a button. As before, runtime modifications, insertions, or deletions are synchronized between the web page and the spreadsheet.

```
<div connect="rows">
  <div class="party-item">
    <span connect="Item">item</span>
    <button delete="row">x</button>
  </div>
</div>
```
→ Cheese ⓧ Crackers ⓧ Wine ⓧ

When a form carries the attribute `connect="rows"`, submitting that form appends a new row. Input elements connected to column names provide the new values for that row. When the new row is appended to the spreadsheet, data flows back to any connected elements on the web page.

```
<form connect="rows">
  Item: <input connect="Item">
  Cost: <input connect="Cost">
  <button>Add</button>
</form>
```
Item: [ ] Cost: [ ] Add

Finally, the commands `hide-if` and `show-if` hide and show a web element based on whether the connected cell is "Truthy," represented by anything other than an empty cell, the string 'FALSE', or the number 0. This example limits the output to show only food items with no assigned person:

```
<div connect="rows">
  <div class="party-item" hide-if="Person"> … </div>
</div>
```

Using these annotations, the Quilt engine can turn a static web mockup and a spreadsheet into a simple read-write web app, with the spreadsheet adding computation as well.

METHOD

Our design goals were to reuse existing HTML and spreadsheets as much as possible, while minimizing the introduction of "programming-like" concepts such as data-flow and ordered statements. Our hypothesis was that doing so would transform a task that today involves programming into one that feels like a natural extension of spreadsheet and HTML authoring. Our method consists of two components: web relations and the spreadsheet-view pattern.

Web Relations

Quilt's programming model, **web relations**, is a refinement of our work on the Cascading Tree Sheets project [5], which provides a language for merging data structures across the web. Web relations adhere to what we call the **CARB properties**—container-centric, annotational, relational, and bidirectional—which lets authors feel as if they are simply authoring HTML, but enable a runtime engine to transform that HTML into a reactive information management interface. To explain these properties, consider the following two code fragments, written in traditional web template style on the left and web relations on the right:

Web Template Style:

```
<ul>
 {for row in rows}
  <li>{{row.Item}}</li>
 {end for}
</ul>
```

Web Relation Style:

```
<ul connect="rows">
 <li connect="Item">Title</li>
</ul>
```

Container-centrism concerns language semantics. When traditional web templates are rendered, the template instructions are removed. The output of a web template is thus not a web template itself, and it lacks provenance information to trace back regions of dynamic data to their source. Web relations annotate HTML *containers* with statements about their *contents*. The UI can be rendered without removing relations, meaning that the output document in the browser is also a working template that can be reused and respond to new data.

Annotationality concerns the programming interface. Traditional web templates intermingle sequential instructions with the template output. Viewed in a web browser, these instructions render awkwardly alongside the HTML. Web relations occupy the attribute space, allowing for example data that is overwritten at runtime. This means the dynamic production template is also a static design artifact.

Relational templating is a contrast to imperative-style templates. Imperative templates provide instructions for a specific task, such as looping over an array and printing something. Relational templates simply represent relationships between data structures. Just like database relations, they can be exploited in different ways depending on context. In Quilt's case, context is the type of object participating in a relation and the data-change events on those objects. For example, consider three different ways the `connect` relation is interpreted based on context. If a `div` is connected to `rows`, its contents will be repeated once per row. If a `form` is connected to `rows`, it will instead append a new row when the form submit event occurs. If a `div` is connected to `A4`, it will mirror the value of cell A4. The `connect` attribute is a relation between structures, not an imperative command, enabling the runtime system to automate a variety of context-dependent tasks that otherwise would require custom code.

Bidirectionality concerns both language semantics and runtime engine design. Traditional web templates are interpreted as a one-way functions that accept data and produce an HTML fragment that is indistinguishable from a static HTML. Web relations define structural relationships that can be exploited in either direction, and container-centrism guarantees the durability of these relationships through the rendering process. A web relational engine can produce dynamic HTML from a data structure, but then also update either the HTML or data structure when the other changes at runtime.

Spreadsheets as "Models with Benefits"

The second part of our method is replacing the abstract model of the Model-View pattern with spreadsheet software. This means that all data and computation is visible, editable, and debugable in a well known environment, not hidden behind controller code and database APIs. Many bugs can be identified and resolved entirely as spreadsheet issues, independent of the HTML interface, or vice versa.

Web spreadsheet software (like Google Spreadsheets) also supports many database-like operations not supported by desktop spreadsheets. Formula-based filtering, sorting, and table-joining, as well as inter-document addressing and web API calls are all possible and can be parameterized by other

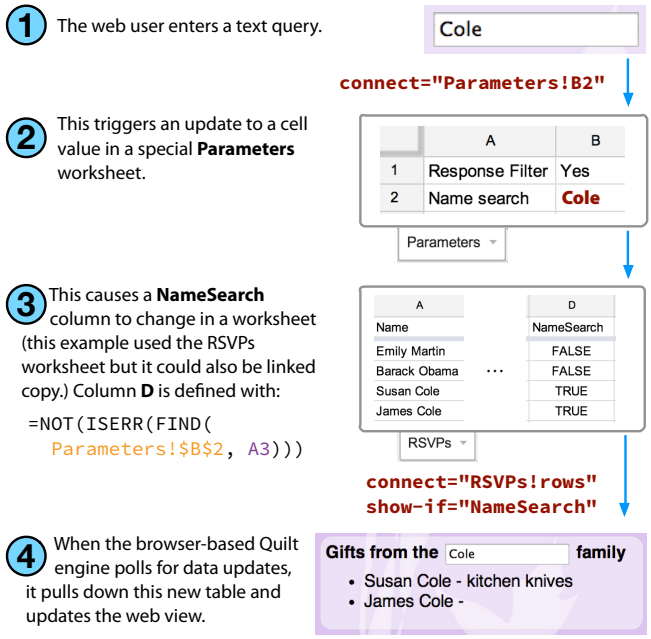


Figure 2: Demonstration of how Quilt's two-way data syncing enables data operations like reactive text search using web elements to parameterize spreadsheet functions.

cells. This open up real possibility for spreadsheets to provide a programming environment for non-trivial web applications. Figure 2 shows a portion of a multi-sheet wedding planning app we built to demonstrate this functionality. When the web user types into a search box on the web page, Quilt synchronizes that with a cell in a **Parameters** worksheet. This causes a column in the **RSVPs** sheet to recompute whether each row is a search result. Quilt synchronizes the new column data to the browser, where it toggles visibility of each list item.

The Discussion section delves into spreadsheet app design patterns, like partitioning different elements of the model and computation into separate sheets and handling multiple users.

SYSTEM ARCHITECTURE

Quilt is implemented as a Javascript library that integrates into a web page to simulate browser-native support for our extra HTML attributes. Figure 3 shows an annotated view of Quilt's internal architecture, along with the timeline of data operations Quilt performs. For re-implementors, the key point to extract from this figure is the way in which Quilt constructs **tree-shaped proxy representations** of all objects it is aware of. Each proxy tree listens to changes on the real object it represents (an HTML node or spreadsheet object) and triggers a synchronization process when a change occurs. This synchronization is enacted as manipulations of proxy tree nodes. A proxy tree is responsible for translating structural manipulations into changes in the root object it represents. While our implementation uses Google Spreadsheets, this proxy-tree design means any object (Excel, CSV, JSON, Database) could be integrated into Quilt's synchronization mechanism so long as it is wrapped in a proxy tree and provides a language to address nodes in that tree.

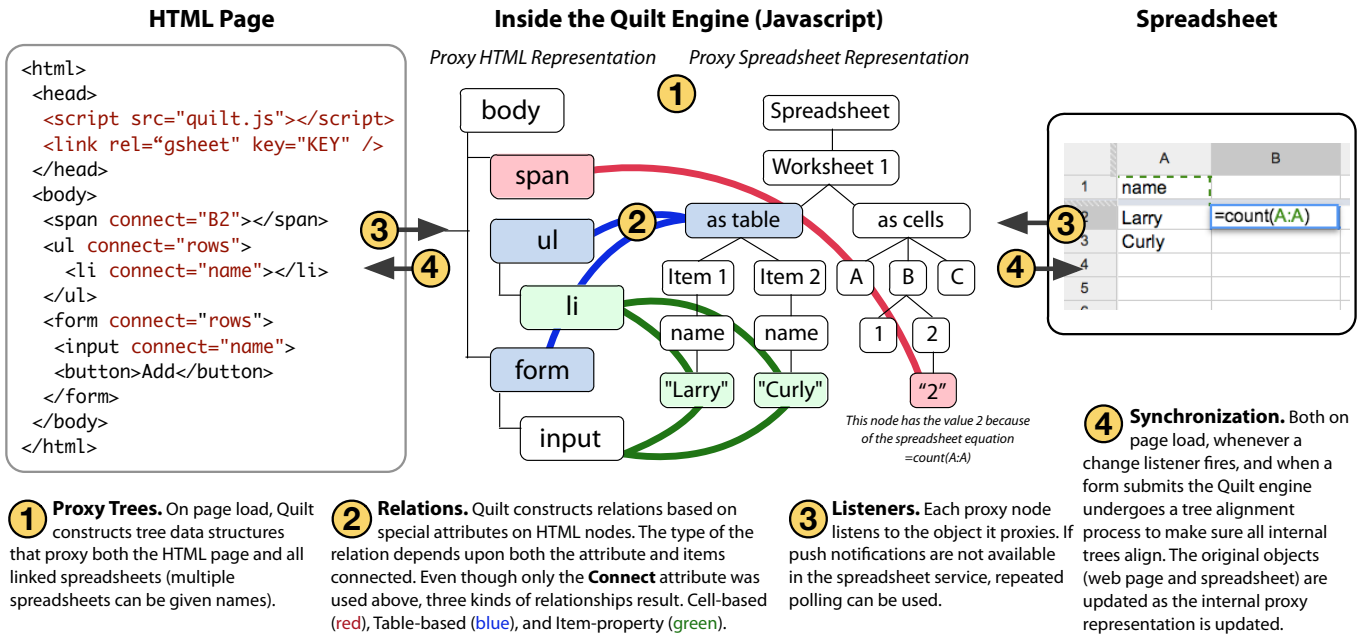


Figure 3: Quilt maintains mirrored, tree-shaped representations of the HTML and spreadsheet. It aligns these representations, rewriting the source objects whenever synchronization is appropriate, such as on web page load or user input.

EVALUATION

We performed user studies of 15 non-programmers to understand whether Quilt's approach can be learned and applied by individuals with only basic spreadsheet and HTML editing knowledge. Specifically, our study examines three questions: armed with a basic understanding of Quilt, can end users:

1. Connect an HTML interface mockup to a spreadsheet to create a web app?
2. Design and build a spreadsheet to support an HTML interface mockup provided to them, and then connect it?
3. Debug Quilt applications with only a browser and the spreadsheet?

Thirteen of the fifteen subjects were able to complete all tasks given to them. Combined with participant surveys and interviews, the data suggests Quilt's approach takes a step toward enabling this population to construct a class of simple interactive web applications they are otherwise unable to build.

Participants

We recruited 15 participants by visiting four local-area programming education organizations that provide introductory web page authoring classes. According to our survey, six participants had only been using HTML for "a few months" or less, nine for a year or less, and all but one for two years or less. Nine participants reported that they cannot write HTML without frequently referring to documentation, nine had never used the HTML `form` tag[2], and five had never used the `table` tag. All participants had used a spreadsheet, and all but one had used Google Spreadsheets. All but four had used a spreadsheet formula. Only four reported that they "understood the concepts" of programming. Of these four, only two had written a computer program before. Both had "about a year" of experience and self-rated as beginners.

Teaching Session

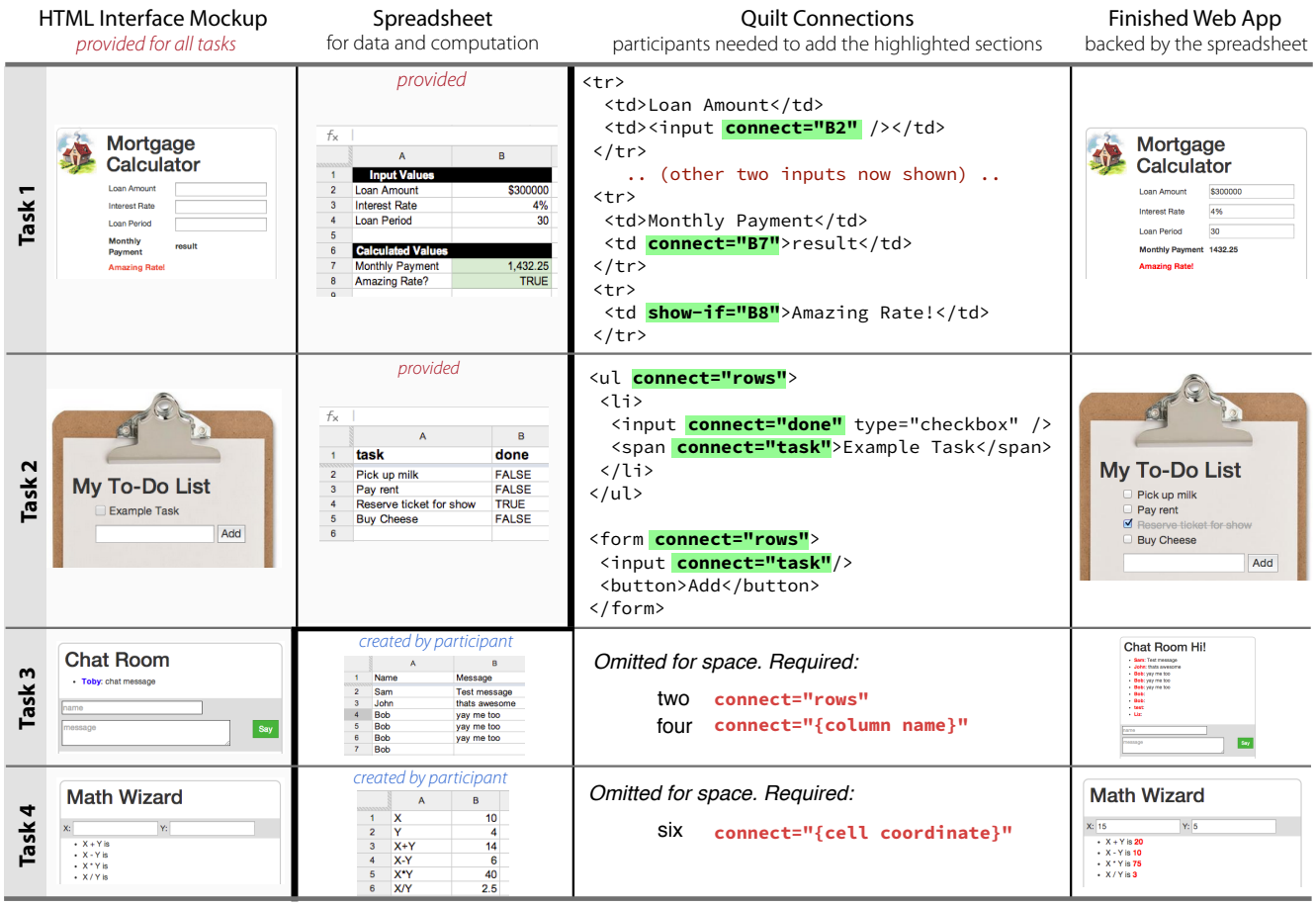
We provided each participant with a fifteen minute private teaching session that was scripted by a worksheet. Using examples, this three-page worksheet presented the main idea behind Quilt (2 minutes), explained the difference between "cell based" thinking—good for fixed inputs and outputs like a math equation—and "table based" thinking—good for lists of items with named properties (3 minutes), and introduced Quilt commands using an example weather report web site backed by a spreadsheet of forecasting data. Participants used this app to practice displaying data, accepting input, and showing/hiding HTML elements for both cell- and table-based addressing (10 minutes).

Tasks

We began the study with only two tasks, but expanded this to four half-way through because participants were completing the tasks with greater success and in less time than we anticipated. We wanted to both take advantage of the extra time and also proactively address the concern that our study was not targeted at the right level of difficulty. Eight subjects were only presented the first two tasks, and the final seven were given all four. Each task, as well as screenshots from participant sessions, is shown in Figure 4

For each task, we provided the participant with three windows: a web browser with a Google Spreadsheet, a text editor with an HTML page, and a web browser rendering that HTML page. The text editor was pre-loaded with an HTML mockup for an application we wanted them to finish. This

[2]That most participants had not used the `form` tag is indicative of how hard it is for this population to build read-write web pages, as `form` elements are only useful in conjunction with a server-side app.




```
<tr>
  <td>Loan Amount</td>
  <td><input connect="B2" /></td>
</tr>
    .. (other two inputs now shown) ..
<tr>
  <td>Monthly Payment</td>
  <td connect="B7">result</td>
</tr>
<tr>
  <td show-if="B8">Amazing Rate!</td>
</tr>
```

```
<ul connect="rows">
  <li>
    <input connect="done" type="checkbox" />
    <span connect="task">Example Task</span>
  </li>
</ul>

<form connect="rows">
  <input connect="task"/>
  <button>Add</button>
</form>
```

Figure 4: Tasks performed in our user study. For Tasks 1 and 2, participants were given an HTML interface mockup and a spreadsheet containing data. They were asked to author the necessary connections to connect the two documents as an application. The highlighted text in the HTML contains the necessary additions. For Tasks 3 and 4, participants were only given a user interface mockup and had to additionally design and implement the spreadsheet necessary to implement the mockup.

allowed us to study the participant's spreadsheet and Quilt activities without worrying about the separate task of web design, since in theory they all knew basic static HTML authoring. This UI mockup was pre-connected to the Spreadsheet (with a `script` and `link` tag in the `head`) but contained no other Quilt commands. The subject was briefed about the application purpose using a script and then asked to built it.

Tasks were always administered in the same order, and we recorded each task with a screen capture program. The participant was asked to work slowly and speak their thoughts aloud as they programmed. During each task the experimenter was silent except to notify the participant if they had made an undetected spelling error such as `conect` instead of `connect` (we informed them of the mistake's existence but not location).

Tasks 1 & 2 were designed to test whether the subject could learn Quilt and apply it without assistance. In these tasks, the user was provided a finished spreadsheet along with the HTML mockup. The activity required of them was to identify how this mockup should be connected to the spreadsheet to turn it into an application and to make those connections correctly. Task 1 was a mortgage calculator, requiring cell-based connections and the ability to perform output, input, and conditional visibility. Task 2 was a To-Do list, requiring table-based connections and the ability to perform enumeration, output, and row-appending operations.

Tasks 3 & 4 were designed to see if participants were capable of designing and implementing a spreadsheet-based data model to support a Quilt-based application. For these tasks, we provided them with only an HTML mockup and an empty spreadsheet. The activity required of them was to identify the data needs of the UI mockup, construct a spreadsheet to meet those data needs, and then use Quilt to connect the two. Task 3 was a chat website, in which users could post messages to a bulletin board. Task 4 was a math tutor that displayed various arithmetic functions of two input variables.

Result: Learning and Applying Quilt

Before learning Quilt, we showed each participant screenshots of Tasks 1 and 2 and asked how long it would take them to connect the finished web mockup, if provided, to a data

source. Most estimated it would take many hours or workdays. Several provided time estimates but added that they had no idea how to actually go about the task, and two said they would not be able to complete the tasks at all ("not with all the time in the universe" said one). Inexperience likely drives much of the high estimate variance, but these estimates at least indicate perceptions of task difficulty.

After learning Quilt, fourteen of the fifteen participants completed Task 1 without any assistance, and thirteen of fifteen completed Task 2 without any assistance. The time it took to complete these tasks was four minutes or less for most participants, dramatically less than their estimates. Figure 5 shows timing information.

While the web apps in these tasks were simple, they required a set of skills these participants largely lacked using traditional methods: reading, writing, and modifying dynamic web data, appending new data items, and conditionally displaying user interface elements. That they could learn to accomplish these tasks with Quilt in only fifteen minutes indicates that Quilt's programming model translates these core data management operations into an approachable interface for non-programmers. One participant had neither written a computer program nor used HTML's `form` or `table` before. Before learning Quilt, she was unable to describe how one might build the Mortgage Calculator or To-Do List with current technologies. After learning Quilt and completing both tasks successfully, she remarked:

> It feels amazing. I thought it would take me three 8 hour days to do something like this. And then to be able to just do it with Google Docs which is something that I use so frequently, and for it to be so easy and to make so much sense, yeah, it's really cool.

The tasks were not all completed correctly on the first try, however. Twelve participants experienced a non-spelling error in at least one task, and five experienced an error in both tasks. In all cases where the participant ultimately completed the task correctly, they were able to identify and fix the error by reloading and sometimes experimenting with the web page and then altering the code they had written. More about this debugging behavior is in the Debugging section.

Designing Spreadsheets to Back Web Apps

Recall that Tasks 3 and 4 were added half way through the user study to see what would happen when participants were burdened with more of the work. These tasks, shown in Figure 4, only provided a web interface mockup. Participants had to create a spreadsheet to support that mockup themselves and then connect it to the HTML page.

We intended to offer seven participants Tasks 3 and 4, but two had to leave after completing Tasks 1 and 2 and could not be offered the tasks. Of the five that remained and were offered the tasks, all completed both tasks successfully. Their times are shown in Figure 5.

How difficult was this additional data modeling step? While we can not make strong conclusions from only five data points, timing information from the videos along with interviews was interesting. All five participants spent less than one-third of time working inside the spreadsheet for the Chat application—a table-based app. But four of five participants spent more than half of the time working inside the spreadsheet for the Math application—a cell based app.

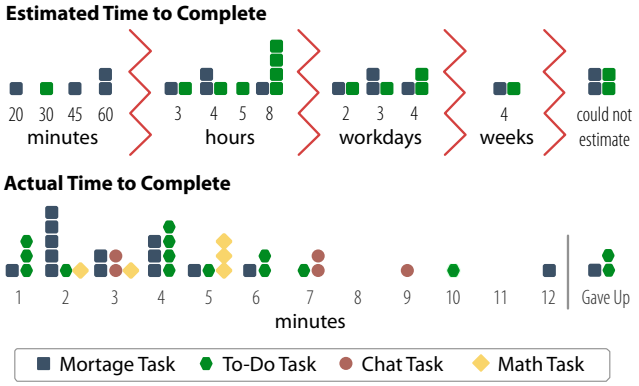


Figure 5: Estimated and Actual time to complete tasks in our study. Please take careful note of the x-axis units: most participants estimated that it would take them many hours or days to complete these tasks before learning Quilt. The actual time to complete these applications was a few minutes.

Some of this difference is simply due to the fact that the math application required more spreadsheet work. But the recorded videos and audio make it clear that the lack of spatial coherence when addressing cells individually posed problems as well. All five users wrote the Chat spreadsheet (a table) correctly on their first attempt, but only two wrote a working Math Tutor spreadsheet without having to make structural changes. The common problem seems to have been that the HTML mockup contained a *list* of math functions, but the data model to support this list did not have any particular list structure to it. This left them unsure of the physical arrangement of data in the spreadsheet to support this interface.

One caught her mistake almost immediately, but the other two created finished tables of math functions, repeating X and Y once per row with a different math result in the third column. They caught their errors when trying to connect the spreadsheet to the web mockup. At this point, the modeling error was apparent because it was not clear how to make the connection. When asked what he was thinking the moment he revised the table, one replied, "Once I went through the process, used my hands instead of my brain, it started making more sense. Once I started trying to connect it to the HTML."

Debugging Behavior

Quilt does not offer any particular debugging assistance past that already in the browser and spreadsheet. One important question is how to best service the debugging needs of SV app authors who are unfamiliar with traditional programming and debugging concepts. To make progress toward answering this question, we coded all *successful* Task 1 and 2 videos to mark when non-spelling errors occurred and how long it took to fix them after being noticed. In these 27 videos, we found 19 errors across 12 (of 15) participants, all of which were

identified and fixed by the participant without assistance. The time it took to fix these errors had high variance—the standard deviation ($77s$) was above the mean ($69s$)—likely due to different skill levels and error context.

While the data set is too small to make strong conclusions, the errors could be clustered into three categories. **Static errors** (13 recorded) were those immediately apparent upon viewing the web page, such as connecting and overwriting the wrong HTML element. No matter what the spreadsheet value, these are obvious. **Dynamic errors** (3 recorded) were those that might be immediately apparent, but might also require experimenting with different data inputs to surface, such as connecting a `show-if` to the wrong cell. If the right cell and the wrong cell had the same "truthiness" value, the error may not be apparent until data changes to reveal the inconsistency, tempting a novice to prematurely declare a correct implementation. **Omissive errors** (3 recorded) were those due to missing functionality or Quilt statements so mistyped that the page simply did not work but no visible clues were available to help diagnose, such as forgetting to connect the `form` element. These errors could be as easily fixed by "fiddling" with the HTML markup.

Figure 6 shows the most elaborate static error encountered: nearly the entire Mortgage web page was connected incorrectly, and the user went through a sequence of iterative modifications until figuring out the proper HTML nodes and spreadsheet cells to connect. At each step in Figure 6, the existence of the error was so clear that a nearest-neighbor walk toward correctness was possible. This participant recalled his thought process afterward, "I tested it out and it changed to the wrong thing so I'm like, 'OK. That's not right,' " Another participant who made an identical error said aloud while programming, "Alright so I put the `connect` in the wrong place. It needs to be connected to the input area."

While these categories may be useful in framing future work, other groupings may prove more appropriate with more data, e.g. connection, conditional, and form errors.

Overall Reactions

Overall reactions to our method were enthusiastic, as the population we studied appeared to be right in the middle of a demographic capable of quickly learning Quilt but not capable of building Quilt-style applications otherwise. Participants liked the immediate feedback and many saw it as a prototyping tool because of that. One participant cited the lack of needing to understand how to write sequential steps often required by programming as a benefit: "For somebody who is very familiar with spreadsheets it makes sense in my head. It doesn't require a lot of 'if this happens, this happens, if this happens this doesn't happen.' It just naturally makes sense,"

All reported that they would use a system like this if it was available to them. "As a beginning programmer, it feels incredible. I feel so much more empowered," one said. And another: "This is great because it lowers barrier to entry for someone who is never going to be a developer."

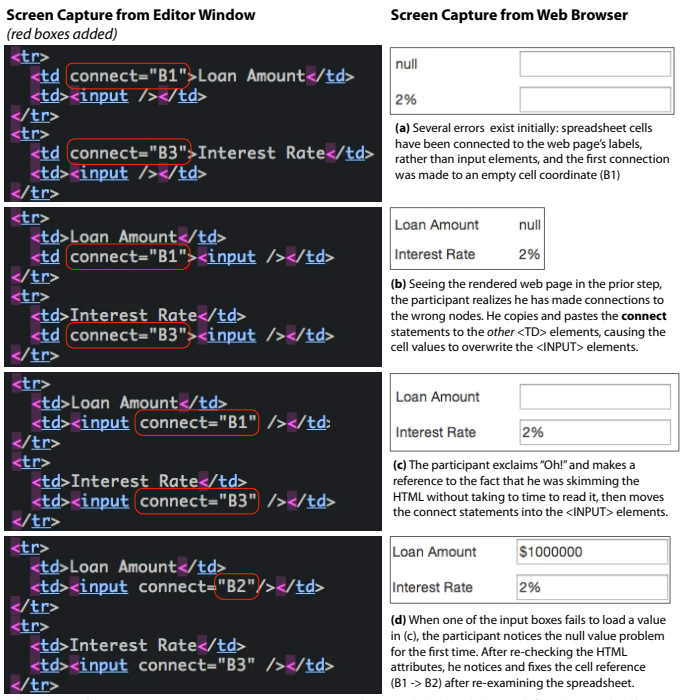


Figure 6: Most errors encountered could be identified and diagnosed by simply reloading the web page. This participant had the longest sequences of such activity. Other errors were harder to detect, requiring the user to experiment with different data values or manually inspect code to diagnose.

DISCUSSION

Low level end-user programming

Our approach is quite low-level compared to related work. Spreadsheet-backed visualizations using the Exhibit framework [14], for example, are built by invoking pre-made widgets from HTML tags. Quilt offers only the ability to connect raw HTML elements to spreadsheet entities. This puts the non-programming population in a position where they come very close to needing to think like programmers.

Concepts like our `show-if` command require understanding a bit about boolean logic. "`show-if` and `hide-if` were kind of confusing. `show-if`, show if what? It would make more sense if you called it 'show if true'," said one user. Others made mistakes concerning the notion of pointers, such as typing `connect="Loan Amount"` instead of `connect="B2"`. This suggests a more flexible addressing scheme that allows users to refence values by nearby cell lables may be useful.

There was also a physicality to some participant's understanding of the task at hand. Three users reported becoming confused when they realized they would need to connect a vertically organized UI element (e.g., Todo list) on the web page to the keyword `rows`, which they thought of as something horizontal. Despite these confusions, the fact that nearly all participants identified and corrected their bugs suggests that the web relations approach reduces the search space of possible programming statements to a set navigable by authors

with only partial understanding of the concepts involved.

Participants did report having to adjust expectations to get used to the idea that one relation (e.g., `connect`) could provide so many different behaviors (output, append, modify, etc). One remarked, "I'm doing something totally different but it's the same wording," and this ran counter to her understanding of programming having many commands. Another commented "I have a sneaking suspicion this won't work," while typing `connect` into an `input` element after having just used it on a `span`. Ultimately, participants seemed to like this approach after getting used to it. "I felt like there were very few functions I needed to remember," one said when asked what she liked about Quilt, and another: "[I felt] overwhelmed, but then I knew I need to figure out where I will put my connects." The relational approach to templating offers the possibility that, in many cases, knowledge that two data objects should be "connected" is sufficient, without understanding the details of how to implement this connection.

In some cases, Quilt's compact programming language *induced errors* because participants got so used to typing `connect` they forgot other commands existed. One participant used `connect` for everything in Task 1, before realizing one of the elements required a `show-if`. She remarked, "I think it was just a matter of momentum. I was just connecting everything. It just seemed like `connect` would just work because everything else just worked with `connect`." Another who made the same mistake said, "I didn't think it through. It was just a natural reaction, because I was connecting a lot of things." One caught himself making this error and muttered, "It ends up being very repetitive. I must not get cocky."

Design Patterns

Because Spreadsheets are turing complete, the spreadsheet-view model can support arbitrarily complex application logic in theory. But the kinds of applications for which SV programming is a *practical approach* for end users will likely be impacted by design patterns developed to handle common scenarios. This is an interesting and fertile ground for future work. We provide two examples here.

Functional View Pattern

It is clear that information management applications have data models. But they also have view models, which represent a transformation of the data model that corresponds to what should be shown in the UI at any given time. In web applications, the view model is often left implicit, scattered across HTML, Javascript, and server-side state. This obfuscates a clear definition of view state in a way that complicate development and debugging.

SV applications provide an attractive way to make these concepts isolated, explicit, and functionally dependent by defining them as separate spreadsheets. Figure 7 shows this pattern, which is similar to the organization we used in the wedding planning app in Figure 2. Using inter-document formulas, the view model sheet is defined as a set of spreadsheet formulas that compute upon the data model sheet. These computations are parameterized by data in the parameters sheet.

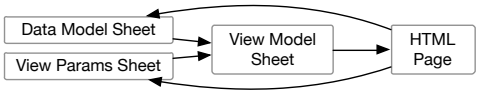


Figure 7: The Functional View Pattern, in which data for the HTML view is stored in a view model sheet. This sheet is a function of a data model sheet and view parameter sheet.

In this pattern, the HTML page displays data from the view model, but it writes data back into the data model and view parameter sheets. Google Spreadsheets natively causes these writes to propagate to changes in the view model, and Quilt causes changes in the view model to propagate forward into the web page. Quilt's spreadsheet addressing language is easily extended to specify which sheet, among many linked sheets on a page, is being referenced.

Quilt's current implementation supports this pattern, but exporting all view logic to a spreadsheet introduces UI latency. This is a problem of system implementation though, not one of end-user programming model design: related work [7] has shown it is possible to automatically rewrite portions a web app's data and computation and relocate it in the web browser for performance gain. The spreadsheet could thus serve as a useful authoring and debugging environment even if it is not the execution platform.

Introducing Multiple Users

Quilt supports multiple simultaneous users with a modification to the functional view pattern shown in Figure **??**. In this pattern, spreadsheets can be designated as either *shared* or *private* in the `link` element that identifies them in the web page's `head`. When a new user visits the web app, Quilt makes fresh duplicates of any sheets marked as private, creating a sandboxed data store for that user that can remain persistent through multiple visits. The private sheet may include both view parameters and user data, and the shared data sheet may aggregate values from many separate private user sheets. This pattern enables users to store private. It also avoids a problem in which simultaneous users might overwrite each other's view parameters (if they were sharing a global view parameter sheet).

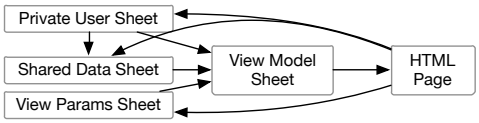


Figure 8: Modification of the Functional View Pattern to support multiple simultaneous users and private data.

UI Possibilities

Visual Programming

Many WYSIWYG environments for HTML authoring already exist, and spreadsheets themselves are a visual programming environment. By grounding Quilt in an HTML-based syntax and existing spreadsheet software, rather than creating a custom HTML+spreadsheet WYSIWYG application, we ensure that this method works independently of any WYSIWYG tools that may be layered on top. From an authoring UI perspective, Quilt only introduces the new step of

crafting declarative relations between HTML elements and spreadsheet elements. Quilt could thus easily be made a visual programming environment by making this relation crafting step visual.

Such an environment might present the author with a split-screen view of their spreadsheet and their HTML UI. The author would then visually drag and drop spreadsheet entities (cells, rows, columns) onto the web page (or vice versa) to draw relations. After drawing a relation, the user selects what kind it is (`connect`, `delete`, `show-if`, or `hide-if`). Because web relations annotate ordinary web pages, this visual development environment could enable the user to interact with and debug the live page while editing it.

Web Worksheets embedded in Spreadsheets

Quilt demonstrates the potential of offering more flexibly designed spreadsheet UIs inside the spreadsheet application itself. Large spreadsheets with many worksheets for data and computation could include "web worksheets" written as HTML to provide a better user interface to the analyses than a tabular view provides. Applications built this way could contain rich UIs while relying upon a well-understood infrastructure that additionally permits users to view and edit the data and computation layers.

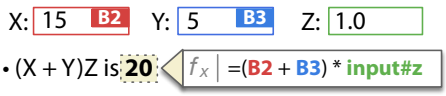


Figure 9: Mockup of what spreadsheet-style editing directly in a web page might look like, enabling spreadsheet elements (`B2` and `B3`) to be mixed with HTML elements (`input#z`).

These packaged web worksheets could even be editable in spreadsheet fashion, like the mockup in Figure 9. A user could select a web element, revealing a formula entry widget to bind the web element to a formula result. A user might even mix spreadsheet cell addresses with CSS selector addresses. The total amount for a restaurant bill might be the sum of column B, plus the Tip entered in a web input box, for example: `=sum(B:B, input#tip)`. A basic example would be re-implementing the spreadsheet UI as a `table` element, with each table cell representing a spreadsheet cell.

Future Work

In addition to performance and design pattern questions raised by our implementation, there are many intriguing questions to explore surrounding spreadsheet-view applications more generally. Understanding what class of applications can practically be authored using this approach, and the kind of user scaffolding required to create nontrivial applications is a key next step. Quilt's choice to use HTML and Spreadsheet software *as-is* enables it to work together with the many tools that already exist to support authoring in each respective medium. But there are also many benefits to be explored by taking a joint approach: modifying the spreadsheet software to embed awareness of the connected web page and explicit UI support for SV application authoring. Chang's work with Gneiss, done in pallel to our work on Quilt, explores this approach [9].

RELATED WORK

End User Web Programming. Building interactive web applications currently requires users to have some knowledge of programming. For the HTML author, this often means learning not only programming fundamentals but multiple new computer languages as well, considering most fully-featured web applications today combine five or more languages [23]. System-level design is also required to coordinate front-end and back-end systems, along with a data storage system such as a database. Research has found that novices struggle with these coordination and integration tasks [17] [16] [23].

The effects of this challenging environment can be seen in the difference between what is valued versus what gets built. Surveys of informal and professional web developers found there were considerable gaps between what web app features were deemed valuable and what got developed, and these gaps were wider for non-programmers [24]. Some of these features include login-based access, databases, online forms and surveys, member registration, and content management.

Mashup-based tools help end users create web applications that integrate data from multiple sources. These tools often require little or no programming, instead using direct manipulation [19] [4], graphical dialog boxes [29], or code samples [13]. Unlike Quilt, these applications only extract data from existing sources and have no interface for storing or managing the user's own data, unless the user can set up their own database. Other tools assist end users with modification and customization of their web experiences [8] [20] [2], but many of these require programming knowledge and also only have the ability to alter existing web pages.

More recently, industry frameworks such as Meteor [3] and Firebase [1] have pushed large parts of a web application into the browser. But both require heavy use of Javascript and learning a reactive data synchronization API. Firebase removes the need to write a back end but provides no support for computation and little support for editing raw data. Quilt also offers reactive data updates but requires no Javascript and leverages a spreadsheet for data editing and computation.

Web design tools for HTML and CSS do not offer support for bridging HTML documents with back-end databases, largely because there are no drop-in frameworks for animating a design. Indeed, a survey of professional web designers found an overwhelming desire for tools that can easily prototype data-driven interactions and provide database integration, even amongst users that knew a moderate to advanced level of programming [21]. In nearly all cases, designers found programming more difficult and time-consuming than web design.

Spreadsheets as End User Information Management Tools. Studies have shown that spreadsheets are an easily available and familiar information management tool for end users, but they present difficulties when scaling to many collaborators and across different physical locations [28]. Few end users convert their spreadsheets to more scalable enterprise database systems, though, because they are notoriously difficult to set up, modify, and query [15].

Much work has been directed at reducing the spreadsheet

errors that occur at larger scale. The Topes project helps end users define and perform data manipulation and validation tasks [25], and other recent work enables data cleaning by example [12]. Online services like Google Spreadsheets scale spreadsheets to many concurrent users, with operational transforms and UI indicators to support simultaneous edits. But because the spreadsheet UI is optimized for numeric calculations, text-heavy data editing remains a usability challenge for spreadsheets. Our method addresses this challenge by allowing spreadsheet data to be cast into HTML interfaces.

Spreadsheet-backed Visualizations. Several efforts have enabled better visualization of spreadsheet data outside the spreadsheet interface. Microsoft Excel has long supported APIs for interacting with spreadsheet data, and recent web-based spreadsheets such as Google Spreadsheets offer web-equivalent APIs as well. A focus of prior work in this area has been to develop ways to package and reuse widgets that make use of this data. The WebCharts [10] project and Google's Charting Widget API both provide a Javascript interface for authoring reusable visualization widgets that draw data from a spreadsheet. But these efforts require the user to learn Javascript and are read-only in nature—the goal is essentially to embed live charts rather than provide an alternate spreadsheet interface. Google Forms creates web-embedable information capture widgets, but they offer limited visual customization and can only append rows to a spreadsheet.

The FORWARD [18] project and Exhibit [14] offer web libraries of data visualization widgets. FORWARD widgets are read-write, but require learning a custom declarative programming language, writing SQL queries embedded in HTML, and using a full database. Exhibit widgets work with spreadsheets and can be invoked with a simple extension to HTML, but they are read-only. In both cases, the systems constrain users to building only those interfaces that can be constructed by snapping together the widgets provided.

Quilt's approach is read-write and widget-agnostic. Rather than provide a way to embed visualizations like charts and maps, Quilt targets the case in which a user seeks a better—or custom—interface to view, edit, and collect spreadsheet data. Because Quilt, like Exhibit, uses a small extension to HTML to provide these interfaces, Quilt apps inherit many attractive sharing and publishing qualities that web publishing provides.

CONCLUSION

This paper presents Quilt, a system that helps end users connect HTML pages to spreadsheets. Quilt authors learn only four new HTML attributes which mark ways in which HTML nodes are related to spreadsheet concepts. Quilt uses these "web relations" to turn an HTML UI mockup into a reactive data-backed user interface. Quilt enables a variety of simple information-management applications, with more complex ones requiring more complex spreadsheet design. Our user studies showed that HTML-savvy end users can quickly learn to use Quilt to connect HTML pages to spreadsheets and to design spreadsheets to support these pages.

REFERENCES

1. Firebase. **`https://www.firebase.com`**, 2014.
2. Greasemonkey. **`http://greasemonkey.mozdev.org`**, 2014.
3. Meteor. **`https://www.meteor.com`**, 2014.
4. Yahoo! pipes. **`http://pipes.yahoo.com`**, 2014.
5. Benson, E., and Karger, D. R. Cascading Treesheets and Recombinant HTML: Better Encapsulation and Retargeting of Web Content. In *WWW '13* (2013).
6. Benson, E., and Karger, D. R. End-Users Publishing Structured Information on the Web: An Observational Study of What, Why, and How. In *CHI* (2014).
7. Benson, E., Marcus, A., Karger, D., and Madden, S. Sync kit: a persistent client-side database caching toolkit for data intensive websites.
8. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R. C. Automation and customization of rendered web pages. In *UIST '05* (2005).
9. Chang, K. S.-P., and Myers, B. A. Creating Interactive Web Data Applications with Spreadsheets. In *UIST* (2014).
10. Fisher, D., Drucker, S. M., Fernandez, R., and Ruble, S. Visualizations everywhere: A multiplatform infrastructure for linked visualizations. In *IEEE Trans Vis and Comp Graphics* (2010).
11. Force, C. S. T. A. T. Csta k-12 computer science standards. ACM (2011).
12. Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug. 2012).
13. Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. Programming by a sample: Rapidly creating web applications with d.mix. In *UIST '07* (2007).
14. Huynh, D. F., Karger, D. R., and Miller, R. C. Exhibit: Lightweight structured data publishing. In *WWW '07* (2007).
15. Jagadish, H. V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., and Yu, C. Making database systems usable. In *SIGMOD '07* (2007).
16. Ko, A. J., and Myers, B. A. Human factors affecting dependability in end-user programming. *SIGSOFT Softw. Eng. Notes 30*, 4 (May 2005).
17. Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. In *VLHCC '04* (2004).
18. Kowalczykowski, K., Ong, K. W., Zhao, K. K., Deutsch, A., Papakonstantinou, Y., and Petropoulos, M. Do-it-yourself custom forms-driven workflow applications. In *CIDR '09* (2009).
19. Lin, J., Wong, J., Nichols, J., Cypher, A., and Lau, T. A. End-user programming of mashups with vegemite. In *IUI '09* (2009).
20. Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M., and Kandogan, E. Koala: Capture, share, automate, personalize business processes on the web. In *CHI '07* (2007).
21. Myers, B., Park, S. Y., Nakano, Y., and Mueller, G. How designers design and program interactive behaviors. In *VLHCC '08* (2008).
22. Rode, J., Bhardwaj, Y., Pérez-Quiñones, M. A., Rosson, M. B., and Howarth, J. As easy as click: End-user web engineering. In *Web Engineering*. Springer, 2005, 478–488.
23. Rode, J., Rosson, M., and Pérez-Quiñones, M. The challenges of web engineering and requirements for better tool support. In *Virginia Tech Computer Science Tech Report TR-05-01* (2005).
24. Rosson, M. B., Ballin, J., and Rode, J. Who, what, and how: a survey of informal and professional web developers. In *VLHCC '05* (2005).
25. Scaffidi, C., Myers, B., and Shaw, M. Topes: Reusable abstractions for validating data. In *ICSE '08* (2008).
26. Scaffidi, C., Shaw, M., and Myers, B. Estimating the numbers of end users and end user programmers. In *VLHCC '05* (2005).
27. Valderas, P., Pelechano, V., and Pastor, O. Towards an end-user development approach for web engineering methods. In *Advanced Information Systems Engineering*, vol. 4001. 2006.
28. Voida, A., Harmon, E., and Al-Ani, B. Homebrew databases: Complexities of everyday information management in nonprofit organizations. In *CHI '11* (2011).
29. Wong, J., and Hong, J. I. Making mashups with marmite: Towards end-user programming for the web. In *CHI '07* (2007).