Fusion: Opportunistic Web Prototyping with UI Mashups

Xiong Zhang
University of Rochester
Rochester, NY, USA
xzhang92@cs.rochester.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Modern web development is rife with complexity at all layers, ranging from needing to configure backend services to grappling with frontend frameworks and dependencies. To lower these development barriers, we introduce a technique that enables people to prototype opportunistically by borrowing pieces of desired functionality from across the web without needing any access to their underlying codebases, build environments, or server backends. We implemented this technique in a browser extension called Fusion, which lets users create web UI mashups by extracting components from existing unmodified webpages and hooking them together using transclusion and JavaScript glue code. We demonstrate the generality and versatility of Fusion via a case study where we used it to create seven UI mashups in domains such as programming tools, data science, web design, and collaborative work. Our mashups include replicating portions of prior HCI systems (Blueprint for in-situ code search and DS.js for in-browser data science), extending the p5.js IDE for Processing with real-time collaborative editing, and integrating Python Tutor code visualizations into static tutorials. These UI mashups each took less than 15 lines of JavaScript glue code to create with Fusion.

Author Keywords

opportunistic programming; UI mashups; web prototyping

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

The web is a powerful platform for building interactive applications, but there can be an irritatingly high barrier to getting started on creating anything non-trivial. Modern web development is rife with complexity at all layers. For instance, developers first need to select and configure a motley mix of hosting providers, cloud APIs, and authentication services on the backend. Then they must often wrestle with tangles of web frameworks, build systems, JavaScript/CSS transpilers, and dependency managers on the frontend before starting to write any useful application code [13, 18].

This level of complexity is often necessary for building production-scale web apps. However, for non-professional programmers such as students, researchers, and UX designers who want to prototype new ideas, it would be ideal to be able to quickly get something simple up and running without all of this boilerplate code and configuration overhead. How can we lower the barriers to prototyping web applications?

In this paper, we propose that one way to quickly get started on prototyping is to use the millions of web applications that others have already built as inspiration and directly borrow pieces of UI functionality from them. To support this *opportunistic prototyping* strategy [16, 24], we introduce a technique that lets developers extract components from existing websites and write small pieces of glue code to bind them together rather than implementing the desired features from scratch. Our approach does not require any access to the underlying source code or backend hosting servers of the fused apps. This eliminates the hassles of setting up and maintaining a full-stack web development environment and drastically reduces the amount of code that needs to be written.

This work extends the long lineage of mashup systems [21, 22, 25, 30, 39] by ***letting users mash up existing web UIs to prototype new web apps opportunistically***. Whereas prior web mashups were mostly data-centric, ours is UI-centric.

We implemented our approach in a Chrome browser extension called *Fusion*. Here is an example usage scenario: Imagine that Alice is a UX (User Experience) designer working at ShareLaTeX [11], a company that makes a web-based collaborative LaTeX editor often used by researchers when writing papers together. One day Alice comes up with a new idea to render previews of math equations inline in the text editor whenever the user highlights a relevant selection. That way, users can quickly debug and fine-tune their equations without waiting for the full PDF document to render. She wants to see how receptive users are to her new idea in order to determine whether it is worthwhile for developers on her team to put in the time and effort to implement it in the actual product.

Figure 1 shows how Alice can use Fusion to quickly prototype this feature so that she can test it on potential users:

(a) She finds a simple webpage called the Interactive LaTeX Editor [32] that allows users to enter a line of LaTeX code into a text box and instantly see the math equation that it compiles into.

UIST '18, October 14–17, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5948-1/18/10...$15.00
https://doi.org/10.1145/3242587.3242632

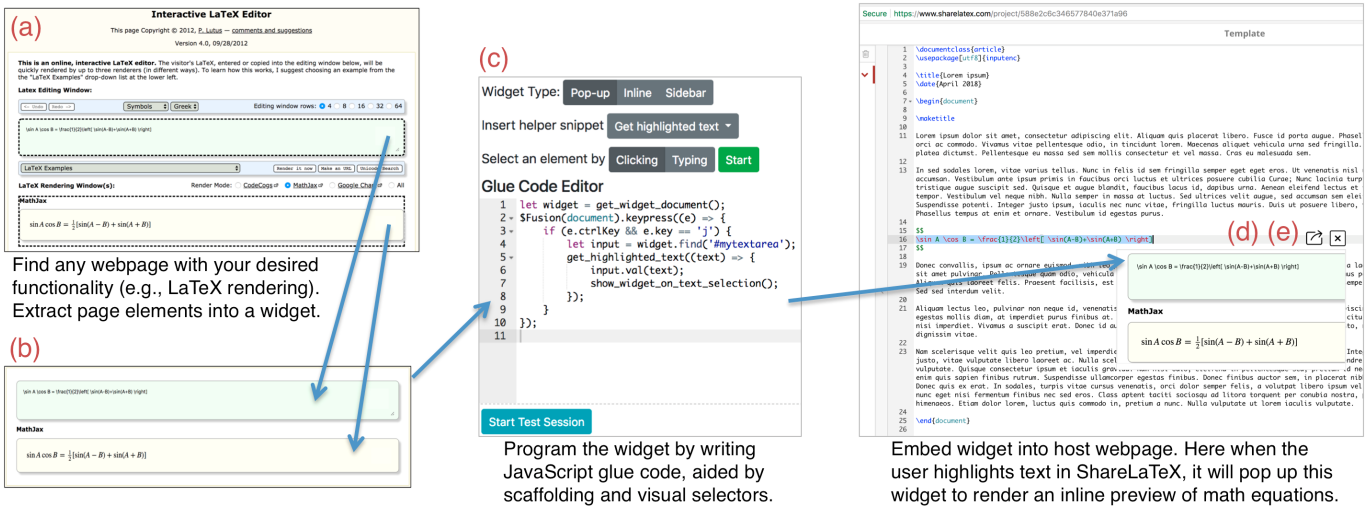


Figure 1: Create UI mashups with Fusion by extracting elements from any webpage into a widget, programming it using a scaffolded development environment, and embedding it into a host webpage.

(b) She activates the Fusion browser extension and extracts two relevant panes from that page—the input text box and the rendered output display—to turn them into a *widget*.

(c) She writes 10 lines of JavaScript glue code to program that widget. Specifically, whenever the user makes a text selection in the browser and hits a keyboard shortcut, Fusion should copy that selection into the widget's input text box, which triggers it to render the equation in its output display. Then Fusion should pop up the widget directly below the user's current cursor. To facilitate this programming process, Fusion includes a code editor, visual element selectors, helper code snippet generators, and test harnesses.

(d) She opens her original ShareLaTeX webpage and uses Fusion to embed that widget into it as an iframe. Now she can highlight any LaTeX within ShareLaTeX and instantly see a pop-up rendered preview of it. (Note that her widget is not specifically tied to ShareLaTeX; she can also embed it into other pages to achieve similar functionality on there.)

(e) She creates a unique URL to share her prototype app with anyone who has the Fusion browser extension installed.

Using Fusion, Alice completes a functional prototype of her inline equation preview feature by finding an existing web app that implements a major part of her desired functionality (Interactive LaTeX Editor) and writing JavaScript glue code to "fuse" it to ShareLaTeX (Figure 1). She did not need access to the codebases or hosting servers of either web app.

If Alice did not have Fusion, then she would have needed to: 1) set up a full-stack web development environment for the ShareLaTeX project, 2) find and install libraries for web-based LaTeX rendering, 3) set up a UI widget library for displaying inline pop-ups, and 4) integrate all of that code into the ShareLaTeX codebase, which is itself likely tangled in a jumble of dependencies. (Note that Alice would not be able to use traditional web mashup tools here since what she wants to mash up is UI functionality from those two websites, *not* data streams from APIs.) Since Alice is a UX designer and not a software developer, this barrier to getting started may be too immense, so she may instead resort to lower-fidelity prototypes such as paper or wireframes. However, she wants her test users to be able to write their own real LaTeX code and feel the live experience of seeing pop-up equation previews, so she wants to create a higher-fidelity web app prototype.

The potential significance of Fusion lies in its ability to lower the barriers to prototyping web applications by letting people use examples they find on the web not only as inspiration but rather directly as programmable widgets that they can manipulate within a domain-specific IDE. It enables anyone with basic JavaScript skills (e.g., researchers or tech-focused UX designers such as Alice) to create *works-like* prototypes by borrowing functionality from existing websites. The key technical contribution that underpins Fusion is a novel approach to extracting components from existing webpages and turning them into self-contained widgets that encapsulate both UI functionality and state. These widgets can then be embedded into other webpages in flexible ways.

We demonstrate the generality and versatility of our approach via an informal case study where we used Fusion to build seven web UI mashups in domains such as programming tools, data science, web design, and collaborative work (Figure 7). Our mashups include emulating the Blueprint [15] system for integrating inline search into a web IDE, extending the p5.js IDE for Processing [6] with real-time collaborative editing, and integrating Python Tutor [23] code visualizations into static tutorials. These UI mashups each took less than 15 lines of JavaScript glue code to create with Fusion.

The contributions of this paper are:

- A novel technique for opportunistic web prototyping by creating UI mashups out of existing unmodified webpages.
- Fusion, a web UI mashup system using direct manipulation, iframe-based transclusion, and JavaScript glue code.

RELATED WORK

Mashup Creation Systems

One major family of related work comprises tools that help people create web mashups. A mashup is *"a web application that takes information from one or more sources and presents it in a new way or with a unique layout"* [33]. Programmers create mashups by writing code to call web service APIs (e.g., Google Maps, Twitter) or to scrape structured data from webpages (e.g., Craigslist, Wikipedia). Researchers have attempted to lower these creation barriers by letting users make mashups with direct manipulation and spreadsheet-like UIs.

For instance, systems such as Marmite [39], MashMaker [21], and Vegemite [30] let users select webpage elements to parse and export into spreadsheets; users can then pipe that structured data to pre-made visualizations such as maps or use it to trigger further actions such as filling out web forms. d.mix [25] allows users to visually select elements to sample from selected webpages (e.g., Flickr) that have been augmented with manually-written site-to-service maps; the system generates the corresponding web service API calls, which users can then paste onto a wiki page to see mashed-up data that update live. Gneiss [17] lets users create web applications using a spreadsheet and GUI interface builder connected to web service API calls. C3W [22] lets users extract form input elements from webpages and connect them together using a spreadsheet-like interface to create interactive data views.

Our work differs from all prior web mashup systems in a fundamental way: Rather than helping people mash up web-based data sources to create aggregate interactive views of data, Fusion is designed to help programmers mash up UI functionality from existing webpages in order to prototype interactive experiences such as the example in Figure 1. To this end, Fusion provides affordances for capturing UI elements and events rather for obtaining and parsing data. Also, instead of providing a code-free end-user development environment, Fusion lets users write arbitrary JavaScript code. This design decision has the advantage of greater expressiveness but at the expense of higher barriers to entry: Fusion's users must still know how to write basic JavaScript.

Beyond web mashups, researchers have also developed techniques for altering the behavior of desktop GUI apps and for creating desktop UI mashups without requiring access to their source code. These techniques can operate at the pixel level (e.g., Prefab [19], WinCuts [38]), window manager level (e.g., UI Façades [37]), or UI toolkit level (e.g., Scotty [20]). Fusion continues the spirit of these techniques by operating at the web's DOM level. It benefits from the standardization of DOM events and layout structure, along with the abundance of free examples to draw upon when creating UI mashups.

Webpage Automation Tools

Another class of related work includes tools that automate user actions on webpages. Browser extensions such as Chickenfoot [14] and Greasemonkey [35] allow programmers to write JavaScript automation code within existing webpages. Demonstration-based tools such as ActionShot [29], CoScripter [28] (formerly Koala [31]), and Selenium [10] allow users to record a sequence of actions, which are automatically turned into reusable scripts. These tools are similar to Fusion in that they provide convenient ways to create scripts that run directly on existing webpages. However, they were originally designed for purposes such as web automation and testing, not for prototyping web UI mashups.

Reusable Web Components

The Webstrates platform [26] allows programmers to create self-contained components using standard web technologies and then combine them in flexible ways with real-time synchronization. Fusion was inspired by three major ideas from Webstrates: a) using iframes to implement *transclusion* (i.e., including a live copy of one webpage within another), b) storing user-written JavaScript alongside page contents (as shown in the Codestrates [36] follow-up system), and c) letting user-written code manipulate DOM elements across iframes. Fusion differs in a fundamental way because it is meant to be used to extract elements from existing unmodified webpages and turn them into reusable components without requiring any server-side support. In contrast, users of the Webstrates platform must create their own components from scratch (or fork ones that others have made) using a dedicated Webstrates server and accompanying JavaScript client library.

More distantly related to Fusion are tools that help users copy-and-paste static content from webpages. Products such as Evernote and Microsoft OneNote allow users to extract components from existing webpages and paste them into digital notebooks with their look-and-feel preserved. Research platforms such as Clui [34] extend these ideas by coupling invisible metadata (e.g., a person's contact information) with webpage elements, which allows richer semantic data to be transferred between pages via drag-and-drop. These systems only support robust copying of web data whereas Fusion allows websites to be connected live via user-written code.

Finally, a more distant predecessor that inspired Fusion is HyperCard [27], a pre-web hypermedia system where each card is a self-contained editable component with both visual elements and runnable code. Cards can be joined into stacks to create custom apps. Fusion lets users snip parts out of existing pages on the web to turn them into self-contained HyperCard-like "web cards" which can be "stacked" (i.e., iframe-embedded) with other components to create web apps.

FUSION SYSTEM DESIGN AND IMPLEMENTATION

We designed Fusion to be used during prototyping when a developer wants to augment an existing web application (which we call the *host*) with new functionality. Instead of writing all of that code from scratch, they browse the web to find the desired functionality on another website and use Fusion to extract it into a self-contained *widget*. Then they write a few lines of glue code to "fuse" that widget into the host webpage.

Fusion is implemented as a Chrome browser extension and accompanying client-side web application with $\sim 1,900$ lines of JavaScript code. It imports jQuery and jQuery UI to ease element selection and widget creation. It does not require a server or backend; all work is done in the user's browser.

Workflow: Creating UI Mashups from Existing Webpages

Fusion allows users to create self-contained widgets from existing webpages and use them as components on other pages. There is a four-step workflow for creating these UI mashups:

1. **Create** a self-contained widget by extracting UI elements from any existing webpage.
2. **Program** the widget by writing and testing JavaScript code using the *Fusion Editor*, a scaffolded coding environment.
3. **Embed** the widget into a host webpage to create a mashup.
4. **Share** the resulting mashup with a self-contained URL.

This workflow does not require any special privileged access to the widget or host websites; everything is done client-side within the user's browser using the Fusion Chrome extension.

We now illustrate these steps using the LaTeX live preview example from Figure 1.

Step 1: Create a Widget from an Existing Webpage

The first step is to find a desirable webpage that the user wants to borrow functionality from. When they are on the desired page they can bookmark the webpage URL into Fusion's local storage and tailor it with direct manipulation (Figure 2).

Tailoring means that the user selects only elements of interest on the page, and excludes all the others, so that they can turn it into a self-contained widget with a more focused UI when it is inserted into the host webpage. For example, if we want to make a LaTeX equation preview widget from an existing web app, we may be interested only in its equation input field and LaTeX preview pane. All the other elements (buttons, text descriptions, navigation, advertisements, etc.) are irrelevant.

To select and remove UI elements from a webpage, the user loads the Fusion extension in Chrome and clicks the "Create Widget" button. This activates a visual element selection tool. Upon activation, the user can hover over any visible page element, and a gray translucency layer will appear over it, indicating a selection. They can click it to select this part to include in the widget, and repeat for other elements. If they hover on elements with the Alt/Option key pressed, that layer will turn red to indicate an exclusionary selection; clicking it will exclude the highlighted elements from the widget.

Note that Fusion does not delete excluded elements, since the webpage's functionality may depend on the structure of the DOM and elements inside of it. Instead, to be minimally invasive, it hides elements using CSS `display: none`. When the user is satisfied with the selection they can click the "Done" button to complete the widget. Fusion will save the DOM elements to include or exclude (in the form of CSS selectors) along with the page's URL.

Step 2: Program the Widget in the Fusion Code Editor

After creating a widget from an existing webpage (Step 1), the next step is to write JavaScript "glue code" to interface with host webpages. In our LaTeX previewing example, we want to allow users to highlight a line of LaTeX in any web-based text editor (its host webpage) and see a live preview of the rendered equation. Thus, we need to write glue code that:

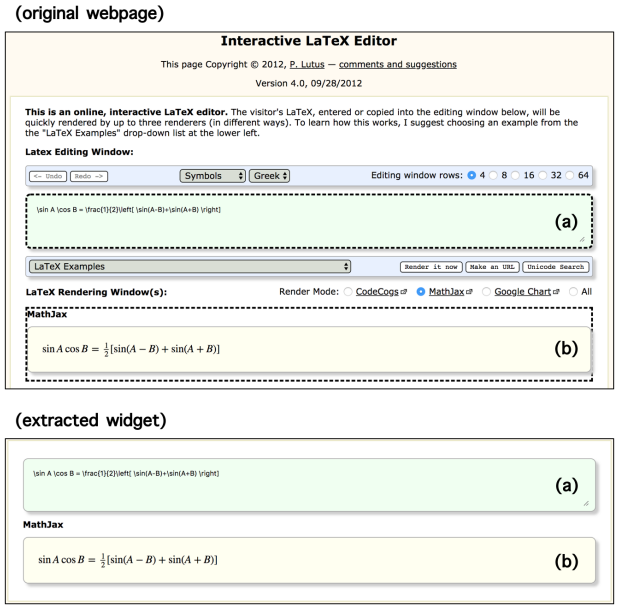


Figure 2: To create a widget from an existing LaTeX rendering web app [32] (top), use Fusion's visual selector to choose both its text input pane (a) and rendered LaTeX output pane (b), which both get highlighted. The resulting extracted widget (bottom) shows only those two panes; all other page elements are hidden but still exist in the DOM so that the original web app can still function.

1. Grabs the currently-highlighted text from the host webpage when a keyboard shortcut is activated.
2. Copies the highlighted text into the widget's equation input field (Figure 2a). This automatically triggers the widget to render a LaTeX preview in the bottom pane (Figure 2b).
3. Pop up the rendered LaTeX preview directly next to the highlighted text in the host webpage.

Since Chrome extensions have access to the JavaScript execution context of all webpages in the browser, the user can in theory write raw JavaScript code to implement this functionality. However, this process can be tedious, so to make it easier in practice, we created a web application called the *Fusion Editor* to provide the necessary scaffolding (Figure 3).

To program a widget in the Fusion Editor, the user first opens an extracted widget they saved from Step 1, which will load it as an iframe in the left pane (Figure 3a). Then they give that widget a name, description, and specify a type, which determines how it should be positioned on the host webpage when it gets embedded in Step 3 (Figure 6). Fusion supports three types of widget embeddings: pop-up, inline, and sidebar.

Pop-up: This type of widget is initially hidden but is activated with a shortcut key to pop up at a given location on the host webpage. This is useful for small dialogs such as our LaTeX preview example (Figure 1). If the user chooses this widget type, Fusion inserts the following starter code into the editor:

```
let widget = get_widget_document();
$Fusion(document).keypress((e) => {
  if (e.ctrlKey && e.key == <key>) {
    // write code to run when Ctrl+<key> is pressed
  }
});
```

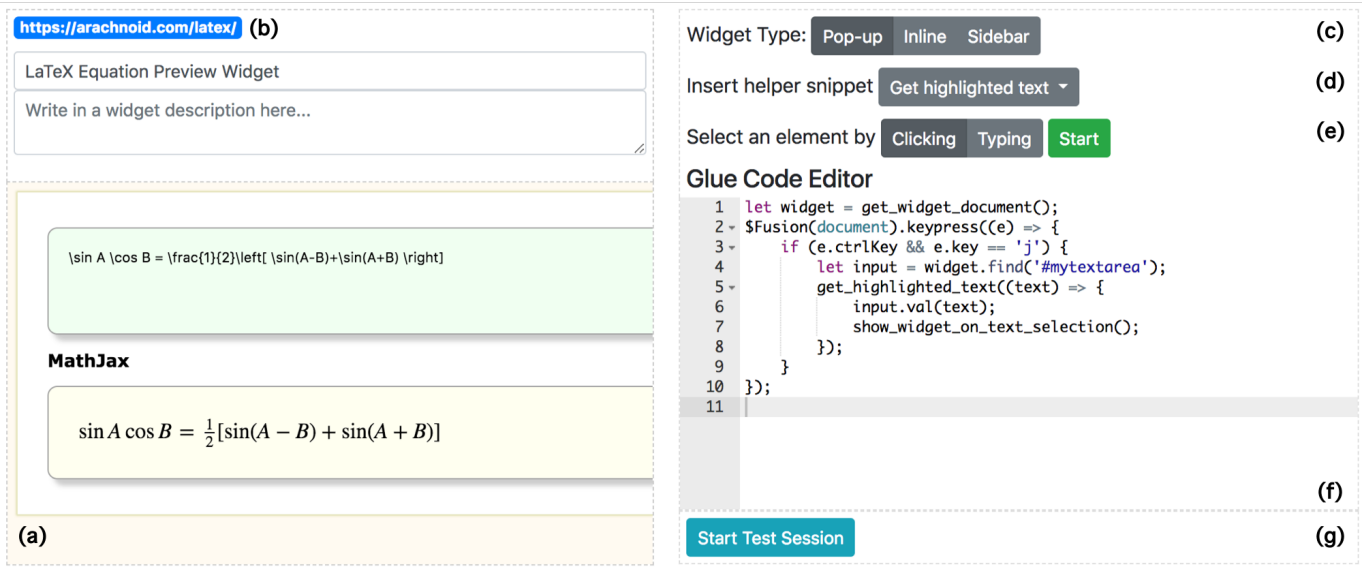


Figure 3: The Fusion Editor is a web application that allows users to (a) import a Fusion widget extracted from any existing webpage (Figure 2), (b) give it a name and description, (c) select a widget embedding type, (d) insert pre-made helper snippets into the code editor, (e) select elements by demonstration to generate additional helper code, (f) write arbitrary JavaScript code in an inline editor, and (g) test the widget in a harness (Figure 5).

The first line provides a handle to the DOM root of the widget, and the second snippet defines a keyboard shortcut event handler. `$Fusion` is a wrapper around a specific version of jQuery and jQuery UI that Fusion uses so that it does not conflict with existing versions on the widget or host webpages. This starter code is a convenient starting point for programming the widget, but the user can freely replace it with other kinds of event handlers such as mouse or scroll handlers.

Inline: This type of widget is always visible and embedded inline directly into a specific DOM element on the host webpage. (The user needs to specify the host element at the time of embedding.) This is useful for times when one wants to "graft" a widget directly onto a portion of the host webpage to augment its functionality; for an example, see our inline code visualizer case study in Figure 7e. Fusion inserts the following starter code to provide convenient access to both the widget and the element on the host webpage that it is being grafted onto, which is specified at embedding time:

```
let widget = get_widget_document();
let host_element = get_host_element();
```

Sidebar: This type of widget is inserted alongside the host webpage in a fixed location. This is useful for augmenting a page with an always-on sidebar pane, such as our code documentation case study in Figure 7c. Here Fusion inserts only starter code to reference the widget itself:

```
let widget = get_widget_document();
```

Now that Fusion's code editor has been seeded with starter code, the user can write any desired JavaScript to implement the widget's functionality. This code will run as a Chrome extension, so it has privileged access to the widget and host webpages, as well as browser-provided information such as the currently-highlighted text region.

All widget code runs once right after the widget gets embedded into a host webpage (see Step 3), so it should perform one-time initialization and define event handlers to respond to events such as keyboard, mouse, and other inputs.

To assist the user in writing this widget glue code, we implemented three scaffolding features into the Fusion Editor: a select-by-demonstration visual element selector, a set of interaction helper snippets, and a simulated test harness.

Select-by-Demonstration Visual Element Selector

In order to programmatically manipulate a widget's UI, the user must first write code to select elements of interest. In theory they can dig into the widget's source code to find this information, but it is tedious to do so, especially if elements are deeply nested or code is minified. To help the user find relevant elements to operate on, we created a demonstration-based selector tool within the Fusion Editor (Figure 3e).

This tool provides two demonstration modes: clicking and typing. The user chooses a mode and clicks the "Start" button to begin a session. Then they can interact with the desired webpage element to specify the one they want to select. a) Clicking mode works in a similar way as the webpage element extraction tool from Step 1: The user hovers over an element and sees a gray highlight and a button next to that element. Clicking that button will generate JavaScript code with a CSS selector for that element and insert that code into the editor. b) In typing mode, the user needs to find an input text box and type inside it. Fusion will capture this typing event and again allow the user to insert JavaScript selector code into the editor. We added the typing mode since we noticed there is a common need to manipulate widget elements that users can type into; we observed it is often easier to find a unique CSS selector by capturing typing events than by directly clicking, since some input fields are occluded or nested deep beneath boilerplate DOM elements from frameworks.

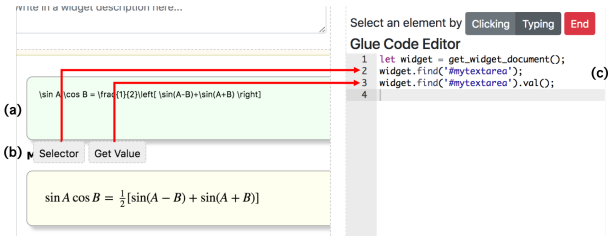


Figure 4: To generate selector code by demonstration, the user types into any input field (a), and Fusion pops up two buttons next to it (b). The "Selector" button inserts code for selecting that element, and the "Get Value" button inserts code for grabbing its current contents (c). (Not pictured is generating selector code by clicking on a DOM element.)

Figure 4 shows the element selector used in our LaTeX preview example. The user activates typing mode and types text into the LaTeX input box in the widget. Fusion detects the CSS of that element and pops up two buttons next to it: a) generate CSS selector code, b) generate code to grab its contents. Clicking a button inserts that code into the editor.

Interaction Helper Snippets

It can be tedious to write raw JavaScript code to add desired features to a widget, so we created a set of helper code snippets to encapsulate typical interactions between widgets and host webpages. We empirically designed these snippets from our experiences of creating a diverse variety of UI mashups using Fusion (see the Case Study section) and looking for ways to abstract away common repetitive code.

Table 1 shows all helpers and how each one works. To use a helper function, the user can choose it from a dropdown snippet menu in the editor (Figure 3d), and Fusion inserts its code into the editor so that it is ready for the user to customize.

For example, our LaTeX preview widget needs to grab the currently-highlighted text in the browser so that it can copy that text into its input box to render as LaTeX. To program this feature, the user clicks the "Get highlighted text" snippet from the menu, and this code gets inserted into the editor:

```
get_highlighted_text((text) => {
  // do something with highlighted text
});
```

This code should go inside of an event handler, such as one that responds to a keyboard shortcut to activate the LaTeX preview. Note that many snippets (like this one) are asynchronous with callbacks since the underlying browser extension and web APIs they depend on are asynchronous.

Simulated Test Harness

The Fusion Editor allows users to develop a widget in isolation before embedding it into host webpages. This lets users create more generic widgets that are not coupled to the specific nuances of individual hosts, although they can still write code that is specific to each host. However, a problem that arises here is that there is no way to test a widget in isolation without first embedding it in a host. And once a widget is embedded into a host, the host webpage's complexity may make testing and debugging more difficult. To address this problem, the Fusion Editor includes a *simulated test harness* with a set of UI elements that are often found on host webpages.

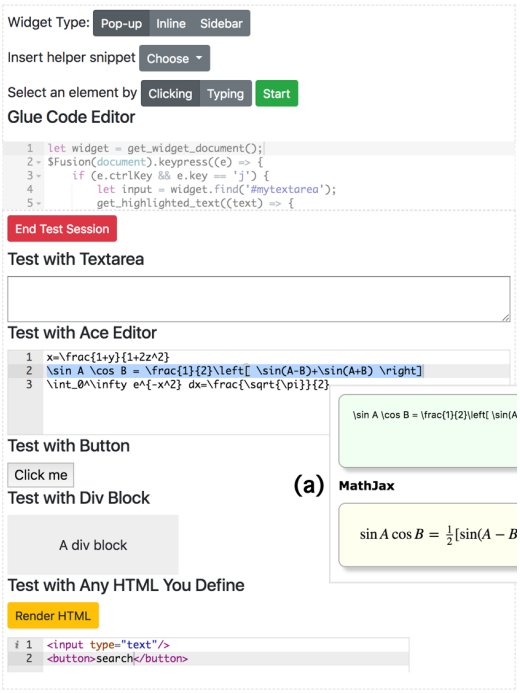


Figure 5: The simulated test harness contains common HTML elements on which to test widget interactions. (a) Here the user is testing that the LaTeX preview widget pops up properly next to the Ace editor's cursor.

When the user activates the test pane in the editor (Figure 5), five HTML elements appear: an input textarea, an Ace code editor, a clickable button, an empty div, and an HTML editor where the user can define arbitrary HTML elements to test.

Fusion then runs the code that is currently in the editor. This simulates embedding the widget into a simulated host webpage that consists only of elements in the test pane and setting up the proper event handlers.

The user can interact with the simulated host elements in the test pane and the widget itself to test their code. They can use built-in Chrome developer tools to inspect elements and print debugging output to the console. Figure 5 shows a test session in our LaTeX preview example, where the user is testing that widget pops up at the location of the Ace editor's cursor.

When they are finished coding and testing, they can save the widget, and it is now ready to be embedded into real hosts.

Here is the final glue code for our LaTeX preview widget:

```
let widget = get_widget_document();
$Fusion(document).keypress((e) => {
    if (e.ctrlKey && e.key == 'j') {
        let input = widget.find('#mytextarea');
        get_highlighted_text((text) => {
            input.val(text);
            show_widget_on_text_selection();
        });
    }
});
```

The majority of these ten lines can be generated from helper snippets. However, the user must properly customize them and then test the widget in the simulated harness.

ID	Helper name	Generates code to ...	Used in Case Studies
A	Get widget document root	return the DOM root for the widget iframe	1, 2, 3, 4, 5, 6, 7
B	Get the embed host element	return the user-selected element when inserting an inline widget	4, 5
C	Get user-highlighted text	extract the text that is currently highlighted in the browser and pass it to a user-defined callback function.	1, 2, 3
D	Show pop-up at mouse location	display a pop-up widget at the current position of the mouse.	2
E	Show pop-up at text cursor	display a pop-up widget at the position of the edit cursor in the focused text input area.	1
F	Add keyboard shortcut	monitor the user pressing a certain key combination	1, 2, 3
G	Add mutation observer	observe an element and call a user-defined callback function when that element changes	6
H	Prompt to select an element	ask the user to visually select an element for them to use in a user-defined callback function	6
I	Send keys to input field	simulate key typing to send text into an input field (e.g., textarea, Ace editor)	5

Table 1: Interaction helper snippets for common widget functions. Selecting one will insert a code snippet into Fusion's code editor to help the user implement the given functionality. Table 2 summarizes the case studies that used each of these snippets.

Step 3: Embed the Fusion Widget into a Host Webpage

Once a widget has been created, programmed, and saved into Fusion's browser storage, it is ready to be embedded into any host webpage to create a mashup that combines their respective features. In theory a widget can be embedded within any webpage, but in practice it makes sense to only embed it into webpages that would directly benefit from that widget's features. For instance, the LaTeX live preview widget does not make sense to embed into a photo sharing website.

The user embeds a widget into the browser's current webpage by selecting it from a menu. This will transclude [26] it as an iframe within the host webpage's DOM. Depending on the widget's type, it will be inserted in different ways, as shown in Figure 6. Pop-up widgets are inserted as an invisible iframe and appear only when glue code runs to make it pop up. Inline widgets are inserted in a user-specified location on the host. Fusion displays a visual selector (similar to Figure 2) to let them choose the DOM element in the host to serve as the root for the inline iframe embed. Sidebar widgets are inserted in a pane beside the host webpage.

After the widget's iframe finishes loading, Fusion runs the glue code that the user wrote in the code editor when originally creating the widget. This sets up the proper initialization code and event handlers to connect the widget to the host. In our example, we embed the LaTeX preview widget as a pop-up into ShareLaTeX, a collaborative LaTeX editor [11].

Since the host and widget webpages likely come from different domains, how can JavaScript code interact with both the host and widget? Modern browsers block this sort of cross-domain scripting for security reasons. Since Fusion is meant to be used in a prototyping context, we assume that UX designers or prototypers trust the glue code. Thus, they should start Chrome with a developer flag to disable cross-domain checks, such as `--disable-web-security` to disable the Same-Origin Policy for scripts. This is the most straightforward and convenient way to set up the browser for Fusion.

However, sometimes it is necessary to leave the same origin policy mechanism enabled, perhaps when deploying to beta testers online. In this case, accessing DOM elements from one iframe to another will be prohibited by the browser, so

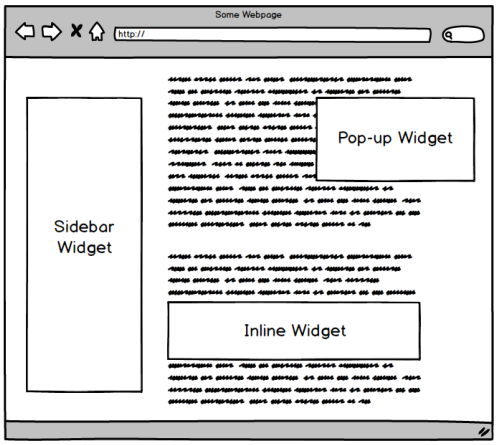


Figure 6: Three ways that widgets can be embedded into host webpages.

we created an alternative setup: We set up a proxy server to work as middleware between users' browsers and the servers where the widget and host webpages reside. Users need to go to a special URL and load both the widget and host pages from there (they will both be iframes in this case). In this way, all webpages (loaded as iframes) will be under the same domain (the proxy's domain), and scripts can freely interface with all iframes without breaking the same origin policy.

Step 4: Share Fusion Mashups with a URL

Fusion provides an easy way to share the mashups that users create by embedding widgets into host webpages. Fusion has a "share" button that, when clicked, generates a URL consisting of the host's URL along with a query string containing a full serialized copy of the widget. This includes the widget's name, description, type, base URL, selected CSS elements, and JavaScript glue code written in the Fusion Editor. This URL can be shared with others or run through a shortener.

If someone has the Fusion Chrome extension installed, when they visit one of these URLs, it will bring them to the host page and embed the widget whose data is fully encapsulated in the URL itself. This mechanism enables sharing of mashups without requiring a hosting server. One concern is that URLs may limit the size of widgets that can be shared this way. However, our case study (next section) shows that

in practice, a wide array of useful widgets can be written in less than 15 lines of JavaScript code, which easily fits within the 2MB URL length limits of modern browsers [1].

DISCUSSION: SYSTEM SCOPE AND LIMITATIONS

We envision Fusion as a medium-fidelity web prototyping tool for use when working in an opportunistic manner, which *"emphasizes speed and ease of development over code robustness and maintainability"* [16, 24]. It is higher fidelity than non-code-based prototyping tools such as Balsamiq [2], InVision [5], and Sketch [12] since it is able to create functional web apps. However, Fusion is lower fidelity than manually coding up the desired features from scratch, so it is not meant to be used to create production-scale finished products.

Fusion works off the premise that a developer can find some website that implements part of their desired functionality and turn that into a widget to fuse into their host webpage. This link between a widget and its host is often feasible due to three properties of web apps: 1) Useful web functionality is often bound to state changes of DOM elements, which JavaScript can detect, 2) third-party glue code can attach extra event handlers to existing elements without disturbing the original intended functionality of the apps, and 3) assuming cross-domain issues are handled (see Step 3: Embed), JavaScript can freely manipulate the UIs of multiple web apps embedded as iframes as though they were a single unified app. Thus, Fusion fails if the developer cannot find a relevant webpage to turn into a widget, if that page's owner disallows iframe embedding, or if they cannot discover the relevant UI elements and event handlers to put into their glue code (which can be challenging in complex web apps like Google Docs).

Fusion is *not* an end-user development environment; it targets users who have some programming background, such as web developers, technical UX designers, and researchers. As our case studies show, it is possible to create a variety of Fusion mashups with only a few lines of code (see next section). However, this still requires users to know how to write JavaScript and understand the concepts of event handling, callbacks, and DOM APIs. The Fusion Editor has scaffolding to make it easier to write and test common kinds of glue code. But that scaffolding cannot cover all possible use cases. Thus, users must still write their own glue code, which presents some barrier to entry. We originally designed Fusion as a code-free visual environment but quickly brushed up against expressiveness limitations. Thus, we opted for a limited coding environment with domain-specific scaffolding.

Fusion does not make attempts to generalize user intent or selections, in contrast to programming-by-demonstration systems for scripting web browsing [28, 29, 31]. For instance, it saves the literal CSS paths of UI elements that the user selects while creating and programming widgets; if those webpages change in layout in the future, then those selectors may break.

Finally, Fusion is a UI-centric mashup tool, so it is not well-suited for deeper integrations between web apps that require, say, manipulating a shared backend or data store that is not exposed in the UI. In those cases, the user must manually write the backend code to implement the desired integration.

CASE STUDY OF BUILDING FUSION MASHUPS

What kinds of web app prototypes can feasibly be built with Fusion? How much effort does it take to implement them? How does Fusion compare with alternative approaches to creating these sorts of web app mashups? To investigate these questions, we performed an informal case study by building seven Fusion mashups and reporting on our experiences.

First we summarize the seven app prototypes that we built:

Live Inline Preview of LaTeX Equations

This is the running example that we used throughout this paper. When editing LaTeX documents, it is helpful to see an inline preview of equations to facilitate debugging. However, collaborative editors such as ShareLaTeX do not have this feature; the user must wait for the entire document to recompile and then navigate and zoom to find their equation of interest. To implement this feature, we found a simple web app where the user types in a LaTeX equation and it shows a rendered version (Figure 2). We created a pop-up widget from it and wrote 10 lines of glue code to have it pop up whenever the user makes a text selection within the Ace editor in ShareLaTeX. Figure 7a shows our final prototype.

Within-IDE Code and Docs Search (emulating Blueprint [15])

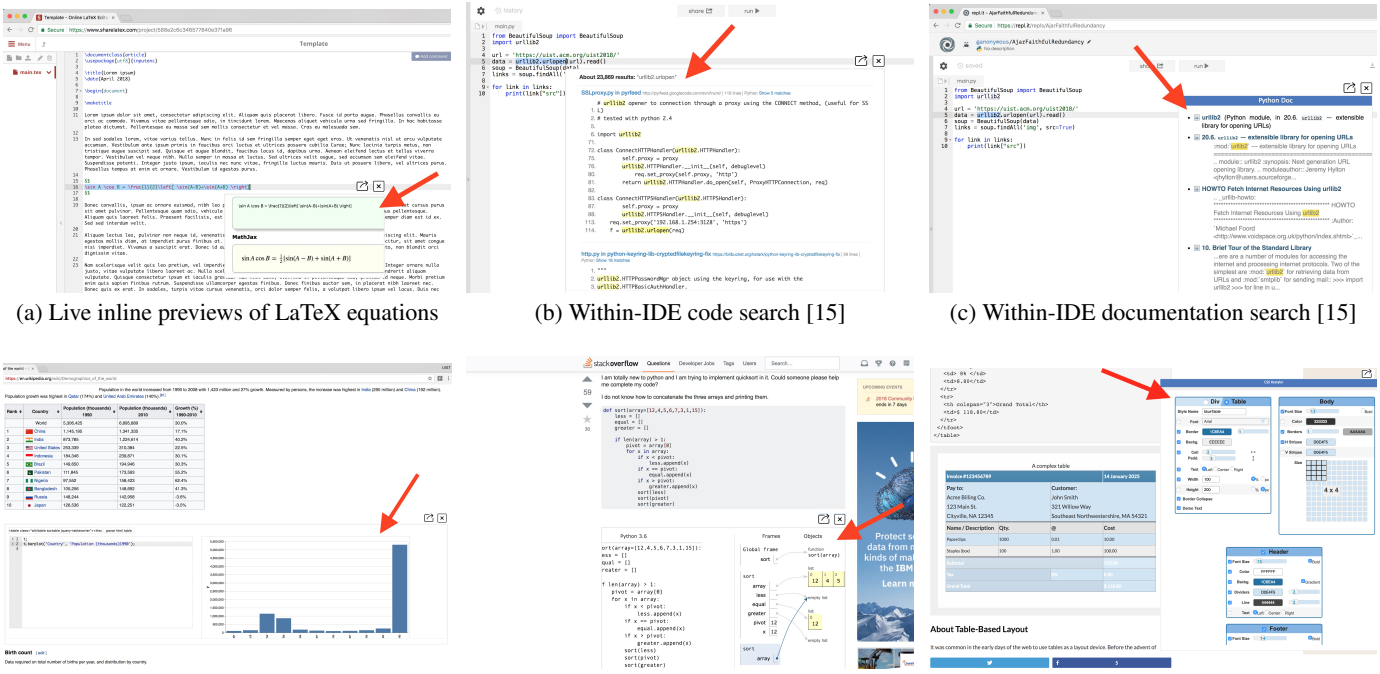
Blueprint [15] demonstrated the utility of integrating inline search for code snippets and documentation into an IDE. To emulate basic parts of its functionality, we prototyped two extensions to Repl.it [8], a web-based IDE.

Figure 7b shows how we created a pop-up widget from searchcode [9], a popular code search engine. In a similar way as the LaTeX equation previewer, the user can highlight any string in the IDE and use a shortcut key to search for relevant code snippets on the web where that string appears. This widget is useful for quickly finding usage examples of how a particular function, class, or method is used in the wild.

Figure 7c shows how we embedded the official Python documentation [7] as an always-on sidebar widget to the right of the IDE. When the user highlights a piece of code in the IDE, it will send that code to the search box in the Python docs webpage and click the search button so that it brings the user to the documentation for that code. This is useful for quickly looking up the definitions of standard library APIs. We chose a sidebar widget format here to give the viewer more space, since documentation pages are often longer and more extensive than individual code snippets.

In-Browser Data Science Environment (replicating DS.js [40])

DS.js [40] is a web-based data science environment that contains an inline JavaScript code editor, data manipulation API, and common data visualizations. It was originally implemented as a specialized bookmarklet that injected its editor UI into existing webpages such as Wikipedia. We took the editor component of DS.js and turned it into a Fusion widget. This way, users can embed it into any webpage that contains tabular data, and it will parse the data on the host page and import it into the DS.js editor (Figure 7d). This fully replicates DS.js's embedding functionality using Fusion instead of relying on its original bookmarklet-based code injector.



(a) Live inline previews of LaTeX equations (b) Within-IDE code search [15] (c) Within-IDE documentation search [15]

(d) In-browser data science environment [40] (e) Code visualization [23] in tutorial webpage (f) CSS restyler [4] for HTML table

Figure 7: Fusion mashups that we created for our case study. The red arrows point to the widgets that are embedded as iframes in the host webpage. Note that the p5.js real-time collaborative IDE mashup is not shown since its embedded Firepad widget is invisible.

Enhancing Tutorials with Code Execution and Visualization

Programming tutorials often contain code examples. However, these are usually static webpages that do not provide a way for learners to execute those code examples. To enhance these tutorials, we created a widget out of Python Tutor [23], a web-based code execution and run-time state visualization website. Anyone with Fusion installed can embed this widget inline below a block of code on any website, and Fusion will copy that code into Python Tutor, click a button to trigger its execution, and display the resulting step-by-step visualization of its run-time state (Figure 7e). The user can edit that code in the inline Python Tutor widget to experiment with variants. This mashup saves the effort of copying-and-pasting tutorial code into the Python Tutor website in separate browser tabs.

Adding a CSS Restyler to Existing Webpages

Designers who want to prototype CSS style adjustments on webpages that they are developing either edit CSS files or use the browser's built-in developer tools. To find inspirational ideas, they often turn to style picking apps on the web that incorporate more advanced aspects of design theory. To facilitate this process, we created a mashup that embeds a CSS style picking web app [4] as a sidebar into any existing host page (Figure 7f). This way, the designer can manipulate styles in the widget and see them show up live on the host page by automatically changing, say, the styles of an HTML table. We wrote glue code to bind the selected table styles in the widget with a trigger to dynamically change the CSS of the host page. This interaction allows designers to experiment with table styles in a more advanced style picking app while seeing their results show up live on the host page.

Adding Real-Time Collaborative Editing to Web IDEs

Many web-based IDEs now exist, and some are coupled with rich output modalities such as graphical canvases. For example, p5.js is a web-based variant of the Processing environment for programming visual designs [6]. However, these IDEs usually do not support real-time collaborative editing. We wanted to augment existing web IDEs with this feature so that people can engage in remote pair programming. To do so, we took the demo web app from Firepad [3], a collaborative text editor like Google Docs, and turned it into a Fusion widget. We hooked up this widget to the code editor component of web IDEs such as p5.js so that textual changes get sent bidirectionally. That way, when two (or more) users with this widget installed visit the p5.js IDE webpage, any code that they write will get sent through the Firepad widget and synchronized with the code editor in their partner's browser. The unique aspect of this widget is that it remains invisible, since we only need its text sync feature; in essence, we are using its UI as an interface to a real-time synchronization backend.

Reflecting on Our Mashup-Building Experiences

Table 2 summarizes the seven mashups that we built, which each took less than 15 lines of glue code ("LOC" column). It took us less than 1 hour to complete the glue code for each, but that was largely because we were already familiar with Fusion's API. The most challenging part of the process was finding the right DOM elements to operate on and testing that our event handlers were firing properly. During this process, we iteratively refined the visual element selector and test harness in the Fusion Editor (Figure 3) to facilitate these tasks.

ID	Host Webpage	Widget Webpage	Widget Type	Glue Code LOC	Glue Code snippets
1	ShareLaTeX	Equation Preview [32]	Pop-up	10	A,C,E,F
2	Repl.it IDE	SearchCode [9]	Pop-up	12	A,C,D,F
3	Repl.it IDE	Python Docs [7]	Sidebar	11	A,C,F
4	Wikipedia	DS.js [40]	Inline	4	A,B
5	StackOverflow	PythonTutor.com [23]	Inline	4	A,B,I
6	HTML.com	Table Style Picker [4]	Sidebar	12	A,G,H
7	p5.js [6] IDE	Firepad [3]	Pop-up	13	A

Table 2: Summary of Fusion mashups that we created, along with lines of code (LOC) and helper snippets (Table 1) used to help write that code.

We also iteratively refined our set of helper snippets (Table 1) by noticing when we were writing similar code across different mashups and finding ways to generalize those patterns. However, we still needed to write a few lines of custom code for each widget since not everything could be generalized.

In our experience, the power and fun of Fusion came from the fact that we did not need to know how these web apps were implemented under the hood or what their software dependencies were; we only needed to come up with ways to mash up their UIs to do what we wanted. With only a *superficial understanding* of how these web apps work at the UI level, we could combine them together in opportunistic ways [16, 24] to perform useful new tasks.

That said, choosing the proper webpages to turn into widgets was critical for making these mashups work well in practice. The target webpage needs to expose the desired functionality in its UI and allow itself to be iframe-embedded into other pages (which is true by default unless the creator disables it). For our case study, we were either trying to replicate or embed existing HCI systems (e.g., Blueprint [15], DS.js [40], Python Tutor [23]), or create mashups in domains we were familiar with (e.g., LaTeX writing, web design, programming). Thus, we had a strong sense for which webpages to pick. However, an open question that still remains is: How can we train new users to choose the proper webpages for their mashups?

Zooming out, we believe an even bigger challenge to adopting Fusion is not necessarily writing the glue code but more about *understanding the scope of what kinds of apps are even feasible to build with it*. As its creators, we were very familiar with the scope and limitations of Fusion (see prior Discussion section). However, if we gave this system to brand new users without any context, it would likely be hard for them to come up with properly-scoped mashup ideas. To provide the necessary scaffolding, we plan to turn our case studies into a live example gallery (Figure 7) so that potential users can see the breadth of what is possible to build and consult our glue code as examples to adapt into their own mashups.

In sum, this informal case study helped us to refine Fusion's core features and demonstrate the range of mashups that are possible to build with it. The next step is for us to develop the instructional scaffolding and training materials required for people to be able to adopt it in practice. Then we need to perform a formal evaluation of Fusion on first-time users to assess its learnability, expressiveness, and usability.

Comparing to Alternative Mashup Approaches

Finally, we reflect on the following question: *What would we need to do to create these mashups if we did not have Fusion?*

The most time-consuming but robust way to create the mashups in our case study is to manually integrate the underlying code of their host and widget websites. For example, to embed Python Tutor into StackOverflow, one needs to integrate the Python Tutor editor and visualizer codebase into the StackOverflow forum rendering codebase. This approach requires having access to the code, dependencies, and build environments of their web apps, which presents a high barrier to entry. Even if one owns that code or acquires it as open-source components, one must still set up the proper development tools and provide hosting for the resulting mashups.

Another approach is to use browser automation tools such as Chickenfoot [14], Greasemonkey [35], or Selenium [10] to create these mashups. One can even write code directly into the browser's JavaScript console. While these tools can technically be used, they were not specially designed for creating UI mashups. For instance, they do not provide convenient means for hooking up widgets to their hosts across iframes, which is a core part of making web UI mashups. In contrast, Fusion is specifically designed with affordances for extracting and connecting components across iframes. It also includes a domain-specific coding environment (Figure 3) with scaffolding to create UI mashups end-to-end in a unified location.

Lastly, the rich lineage of data-centric mashup tools [17, 21, 22, 25, 30, 39] cannot be used to create the kinds of UI mashups that Fusion is meant for, such as those in our case study. Data-centric mashup tools require either a web service backend to provide data APIs or structured data to scrape from webpages. And the output of such tools is usually some aggregated interactive view of data in a spreadsheet, HTML table, or annotated map. In contrast, Fusion relies on binding together UI actions from multiple webpages, and its output consists of new user interactions derived from the UIs of the widget and host webpages. Despite sharing the terminology of "mashups," these tools serve very different purposes.

CONCLUSION

In this paper, we introduced the idea of opportunistically prototyping web apps by creating UI mashups. The key insights that make our idea feasible is that web apps expose a wide variety of useful functionality in their UIs and that iframe-based transclusion [26] with small amounts of JavaScript glue code can effectively bind together disparate web apps without needing access to their underlying codebases or development environments. To facilitate making web UI mashups entirely client-side, we built a Chrome browser extension called Fusion. Fusion points toward a future where web prototyping becomes more accessible to a broader range of stakeholders including UX designers, researchers, and students who may not have deep knowledge of full-stack web programming.

ACKNOWLEDGMENTS

Thanks to Jonathan Edwards, Kandarp Khandwala, Alok Mysore, and the UCSD Design Lab for their feedback.

REFERENCES

1. 2017. Maximum length for url in chrome browser. https://stackoverflow.com/questions/15090220/maximum-length-for-url-in-chrome-browser/25383986#25383986. (2017).
2. 2018. Balsamiq. Rapid, effective and fun wireframing software. https://balsamiq.com/. (2018).
3. 2018. Firepad: Open source collaborative code and text editing. https://firepad.io/. (2018).
4. 2018. HTML Table Styler CSS Generator. https://divtable.com/table-styler/. (2018).
5. 2018. InVision: Digital Product Design, Workflow & Collaboration. https://www.invisionapp.com/. (2018).
6. 2018. p5.js. https://p5js.org/. (2018).
7. 2018. Python 3.6.5 documentation. https://docs.python.org/. (2018).
8. 2018. repl.it - Online REPL, Compiler & IDE. https://repl.it/. (2018).
9. 2018. searchcode: Search over 20 billion lines of code from 7,000,000 projects. https://searchcode.com/. (2018).
10. 2018. Selenium - Web Browser Automation. https://www.seleniumhq.org/. (2018).
11. 2018. ShareLaTeX: LaTeX, Evolved The easy to use, online, collaborative LaTeX editor. https://www.sharelatex.com/. (2018).
12. 2018. Sketch - The digital design toolkit. https://www.sketchapp.com/. (2018).
13. Raphaël Benitte, Sacha Greif, and Michael Rambeau. 2017. The State of JavaScript 2017. https://stateofjs.com/. (2017).
14. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST '05)*. ACM, New York, NY, USA, 163–172. DOI: http://dx.doi.org/10.1145/1095034.1095062
15. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 513–522. DOI: http://dx.doi.org/10.1145/1753326.1753402
16. Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1589–1598. DOI: http://dx.doi.org/10.1145/1518701.1518944
17. Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 87–96. DOI: http://dx.doi.org/10.1145/2642918.2647371
18. Eric Clemmons. 2015. Javascript Fatigue. https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4. (2015).
19. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1525–1534. DOI: http://dx.doi.org/10.1145/1753326.1753554
20. James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 225–234. DOI: http://dx.doi.org/10.1145/2047196.2047226
21. Rob Ennals and David Gay. 2007. User-friendly Functional Programming for Web Mashups. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 223–234. DOI: http://dx.doi.org/10.1145/1291151.1291187
22. Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. 2004. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 175–184. DOI: http://dx.doi.org/10.1145/1029632.1029664
23. Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368
24. Björn Hartmann, Scott Doorley, and Scott R. Klemmer. 2008. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing* 7, 3 (July 2008), 46–54. DOI: http://dx.doi.org/10.1109/MPRV.2008.54
25. Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a Sample: Rapidly Creating Web Applications with D.Mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 241–250. DOI: http://dx.doi.org/10.1145/1294211.1294254

26. Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 280–290. DOI: http://dx.doi.org/10.1145/2807442.2807446

27. Matthew Lasar. 2012. 25 years of HyperCard—the missing link to the Web. https://arstechnica.com/gadgets/2012/05/25-years-of-hypercard-the-missing-link-to-the-web/. (2012).

28. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1719–1728. DOI: http://dx.doi.org/10.1145/1357054.1357323

29. Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 723–732. DOI: http://dx.doi.org/10.1145/1753326.1753432

30. James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

31. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 943–946. DOI: http://dx.doi.org/10.1145/1240624.1240767

32. Paul Lutus. 2012. Interactive LaTeX Editor. https://arachnoid.com/latex/. (2012).

33. Daniel Nations. 2016. What is a Mashup? Exploring Web Mashups. https://www.lifewire.com/what-is-a-mashup-3486655. (2016).

34. Hubert Pham, Justin Mazzola Paluska, Rob Miller, and Steve Ward. 2012. Clui: A Platform for Handles to Rich Objects. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 177–188. DOI: http://dx.doi.org/10.1145/2380116.2380141

35. Mark Pilgrim. 2005. *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks)*. O'Reilly Media, Inc.

36. Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 715–725. DOI: http://dx.doi.org/10.1145/3126594.3126642

37. Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. 2006. User Interface FaçAdes: Towards Fully Adaptable User Interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 309–318. DOI: http://dx.doi.org/10.1145/1166253.1166301

38. Desney S. Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems (CHI EA '04)*. ACM, New York, NY, USA, 1525–1528. DOI: http://dx.doi.org/10.1145/985921.986106

39. Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-user Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 1435–1444. DOI: http://dx.doi.org/10.1145/1240624.1240842

40. Xiong Zhang and Philip J. Guo. 2017. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 691–702. DOI: http://dx.doi.org/10.1145/3126594.3126663