Attaching UI Enhancements to Websites with End Users

Michael Toomim[1], **Steven M. Drucker**[2], **Mira Dontcheva**[3], **Ali Rahimi**[4], **Blake Thomson**[1], **James A. Landay**[1]
[1]University of Washington DUB Group {toomim, thomsbg, landay}@cs.washington.edu
[2]Microsoft LiveLabs sdrucker@microsoft.com
[3]Advanced Technology Labs Adobe Systems mirad@adobe.com
[4]Intel Research Berkeley ali.rahimi@intel.com

ABSTRACT
We present reform, a system that envisions roles for both *programmers* and *end users* in enhancing existing websites to support new goals. First, programmers author a traditional mashup or browser extension, but they do not write a web scraper. Instead they use reform, which allows novice end users to attach the enhancement to their favorite sites with a scraping by-example interface. reform makes enhancements easier to program while also carrying the benefit that end users can apply the enhancements to any number of new websites. We present reform's architecture, user interface, interactive by-example extraction algorithm for novices, and evaluation, along with five example reform enabled enhancements. This is a step toward write-once, apply-anywhere user interface enhancements.

Author Keywords: web data extraction, mashups, programming by example, end-user programming

ACM Classification: H5.m. Information interfaces and presentation: User Interfaces.

INTRODUCTION
Webmasters cannot afford to support all end-user goals. Every site design prioritizes some features and uses over others, and every site's implementation is limited by the webmaster's time, incentives, and knowledge. For example, many sites present lists of addresses without a map, forcing users to perform tedious copying and pasting to a map website. Few sites implement a mobile-optimized version for a user's favorite phone. Online phone bill designs do not include visualizations to help users switch to cheaper plans and spend less money. Online shopping carts do not offer coupons or better deals at other stores. Although there are many website features that would enhance important end-user tasks, the webmasters in charge lack the time, incentives, or knowledge to implement them.

Instead, third-parties develop mashups, browser extensions and scripts [2,9], and web proxies [19] to enhance the web post-hoc. Unfortunately, there are not enough developers to reform all websites: there are 175 million websites on the Internet, yet in comparison the United States employs fewer than 2 million programmers [18,24]. Scripts and mashups must be updated when a website's layout changes, and each site can be enhanced in multiple ways. This website-by-website strategy cannot scale to the entire web without tasking every programmer on earth with the development and maintenance of multiple site enhancements.

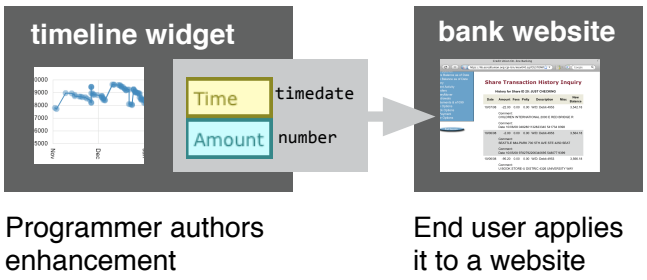

Figure 1: reform **divides web enhancement into roles of authoring, for programmers, and attaching, for end users.**

We propose instead leveraging the Internet's 1.4 billion end users, allowing a single programmer to enhance many websites at once. A programmer authors a single site-independent web enhancement, and end users attach it to all the sites they use in the context of their existing tasks. This architecture of write-once apply-anywhere web enhancements divides web enhancement into two roles: programming and attaching. This allows end-users to do the attaching, and bring enhancements to many more sites.

The key is enabling end users to teach an enhancement how to attach to a new website and understand its data representation, a difficult problem traditionally studied as *web information extraction* or *web scraping* [11,16]. We present a new interactive machine learning technique designed for novice end users, allowing them to scrape a variety of data layouts by example, without seeing the underlying webpage representation.

Our prototype is a library for Firefox extensions called reform. Rather than hard-code HTML or DOM patterns to access parts of a webpage, web enhancements (Firefox extensions) query the reform library with a schema expressing the general type of data they expect a webpage to contain. reform then prompts the user to click on parts of the page that match the schema, interactively training its

CHI 2009, April 4–9, 2009, Boston, Massachusetts, USA.

scraper by example. For instance, a map enhancement will use reform to prompt the end user to click on example addresses. The reform library then generates and applies an extraction pattern, provides the enhancement with its requested integration points, and stores the pattern in a central database for future use. A programmer can invent a new AJAX photo viewing widget that works on multiple photo sites, even ones he has not seen. He can write a shopping aggregator that plugs into web retailers that did not exist when he wrote it. He can script a new feature into his favorite webmail system, and end users can repurpose it to work with their own webmail systems.

Using the reform library, we built five web interface enhancements in Firefox. We built two *data visualizations*: a universal Google map, and a timeline graph that can visualize patterns in time series data, such as a phone bill or bank account, on any scrape-able page. We built a *multi-site aggregator* as a shopping assistant that learns layouts of coupon rebates and merchant deals, and notifies the user when viewing a product for which there is a coupon or better deal elsewhere. We implemented an *interface facade* that makes an iPhone-optimized version of a website after an end user shows it example article titles and summaries. We implemented an *interactive AJAX widget* that replaces the standard "click next page and wait for a page refresh" idiom on multi-page sites with a single automatically-fetching infinitely-scrolling page. These illustrate a space of enhancements that one can build with the reform library.

This work contributes an architecture for web enhancement that allows end users to integrate existing enhancements with new websites. It also introduces an interaction technique and learning algorithm that allows end users to train a web scraper. We evaluated reform in three ways: we built a few example enhancements to validate the system architecture; we measured the machine learning generality by testing a sample of websites from the Internet; and we ran a small usability study to see if the interactive machine learning was accessible to the novice end users reform targets.

In the rest of this paper we first situate reform within related systems. Then, we describe the user interface in detail, and explain the machine learning algorithm and its limitations. We then explain the five interface enhancements we implemented, our user study, and how we evaluated the algorithm's cross-site generalizability.

Related Work

reform contributes both an architecture and interactive attachment algorithm. Here we describe the alternative architectures for attaching enhancements to websites.

Programmer Hard-Codes Attachment to a Single Site

Whereas reform allows end users to attach an enhancement to new sites, traditional mashups and scripts are hard-coded to specific websites by the original programmer. Tools like Chickenfoot [2], CoScriptor [17], Greasemonkey [9], Marmite [25] and Highlight [19] make the task of developing such enhancements easier, but do not separate the tasks of development and attachment. Thus, the original developer must adapt each enhancement to each new website.

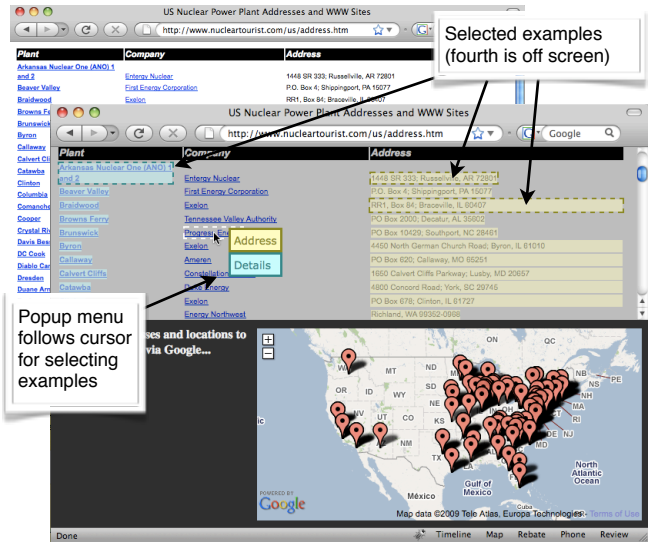

Figure 2: Before and after an end user made a map for a page of U.S. nuclear reactor locations. The process took five clicks.

Programmer Leverages Structural Heuristics

Some web enhancements tune extractors to the structural patterns of particular data layouts. For instance, Web Summaries [6] implements an XPath extractor with support for the list & detail layouts common on shopping pages, and Sifter [10] has a mostly automatic algorithm tuned to find, filter, and sort common search result layouts. These extractors are difficult to implement and generalize to support many websites. Furthermore, many extraction problems are by nature ambiguous and require user input, yet mostly-automatic systems like Sifter [10] offer the user little help when the extractors fail. Karma [4] and Mashmaker [7] can learn from positive but not negative examples. Mashmaker users must drop into a lower level pattern editor to make a pattern more selective. reform is a step towards generalizing such extraction techniques.

Programmer Leverages Predefined Webpage Semantics

Many systems allow enhancement of any site that includes predefined semantic markup in formats such as RSS feeds, Semantic Web RDF, microformats, or web APIs. For instance, Vispedia [4] allows visualization of Wikipedia articles by leveraging the RDF predefined for each topic as part of the DBpedia project. d.mix [12] allows experts to define a library of patterns that end users employ. Visual programming mashup makers like Yahoo! Pipes [26] require web data to be prepared with special wrappers. Since semantic markup is not yet pervasive on the web, this requirement limits the websites that can be enhanced.

End User Combines Separate Extraction & Use Systems

Systems often combine extraction tools with enhancements in separate modular steps. Traditional extraction systems are not directly connected to a user goal; they instead extract data to an intermediate representation, which can indirectly be fed to enhancements. Dapper [5] has a robust by-example interface, but extracts data to intermediate formats like XML and RSS. Mashmaker [7] and Karma [22] also support some by-example extraction, but Mashmaker has a

separate data extraction step where users specify a hierarchical schema before connecting data patterns to widgets, and Karma decomposes the mashup process into the steps of extraction, data cleaning, source modeling, and data integration. Irmak [14] presents a by-example extraction algorithm that uses similar features as our own, but was not designed for end users or placed within an end-to-end system. Our formative studies described later found intermediate representations to be obstacles to end-user extraction.

End User Must Understand HTML/DOM.
Many commercial scrapers and extractors, such as Lixto [8], require the user to understand how web pages are represented, and specify pattern-matching expressions in terms of HTML or a DOM. These are difficult to learn and use.

End User Manually Selects, Copies, Pastes Data
Some web enhancements, such as Ubiquity [23], require each website datum to be manually selected or copied and pasted. This manual approach can quickly become unwieldy for larger data sets.

Attaching enhancements to arbitrary websites is a difficult problem. reform is a step towards a generalized solution that these existing systems could use to simplify development and generalize to more websites.

END USER UI ATTACHMENT WITH REFORM

reform's purpose is to enable the 1.4 billion Internet end users to attach enhancements with a web extractor. Traditional extraction approaches pose two major challenges to end users. First there is the *pattern expression* problem: how can an end user, without learning a special language or understanding HTML or DOM representations, specify an extraction pattern that is expressive enough to represent the wide variety of DOM structures that can appear on different websites, and navigate the many inconsistencies and special cases that occur in layouts? Second, there is the *data mapping* problem: how can an end user plan and design a data schema that is compatible with the desired enhancements, extract the data to that schema, and connect the schema to the enhancements? Both tasks can require up-front planning, abstract reasoning and can be difficult to debug. These challenges are obstacles to widespread end user web enhancement. We now outline reform's approach.

Pattern expression: reform users specify patterns by example, and a machine learning system infers an expressive matching pattern behind the scenes. User interaction involves nothing more than highlighting elements of the webpage that should be connected with the enhancement and removing highlights from incorrectly inferred elements. Our machine learning algorithm synthesizes hundreds of features that explain the examples provided and computes weights to choose the best amongst them.

Data mapping: Traditionally, extraction systems output to an intermediate data representation with a general schema, which allows extracted data to be reused for multiple purposes. This modularity is sensible if extraction is a difficult and costly task. reform, in contrast, eliminates intermediate steps and ties extraction directly to a specific, constrained enhancement goal. The programmer defines an enhancement schema, and the end user only highlights the webpage elements that fit the schema's fields; a simpler task. In our formative design process, we tested end users on hypothetical schema debugging tasks with different designs for textual and visual diagrammatic schema representations, but users uniformly found them abstract and had difficulty identifying errors in schema definitions, especially with the more complicated hierarchical schemata with nested lists of lists. Schema definition appears to be a burden to end users. Furthermore, directing extraction towards a specific enhancement goal has additional advantages:

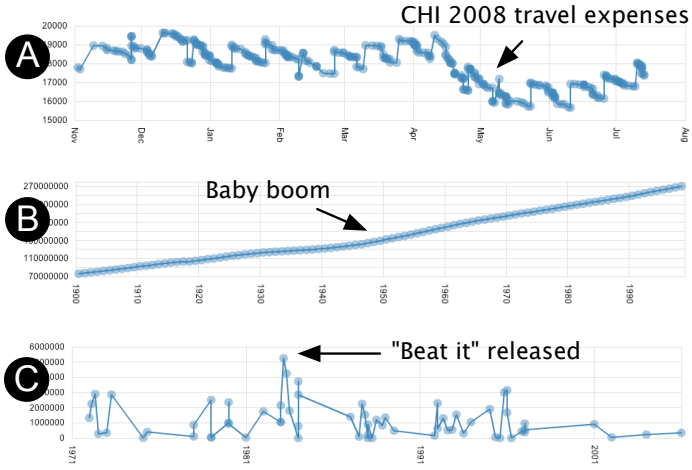

Figure 3: (A) Nine months of personal bank account history (comfirstcu.org) (b) US population growth from npg.org (c) The sales of Michael Jackson released singles from Wikipedia

- We can guide and prompt the user through extraction, using the concrete terminology defined by the enhancement. If the enhancement needs a "purchase price" from the webpage, the extractor will ask the user to click "purchase prices". After each click, the system is able to update the enhancement's display with the new extracted data, providing the user with incremental feedback towards the enhancement goal.
- Predefining the schema can also guide and constrain the machine learning. For instance, if it knows the user is selecting a time, it can consider only DOM nodes that parse as a time. This allows accurate inference with fewer examples.

We will explain reform's interface and extraction algorithm with the following running example.

Visualizing Financial Patterns with a Bank Timeline

Most bank websites provide users with a multi-page list of purchases and transactions: an online "bank statement." Although this presentation may fulfill contractual obligations, it is a difficult format for understanding spending trends and spotting fraud or errors. Here we illustrate reform by describing how an end user applies a reform-powered timeline to their banking site, shown in Figure 4. In the next section, we will give more details about reform's algorithm.

This timeline is a Firefox extension using the reform library. The extension is a general purpose timeline: it is not

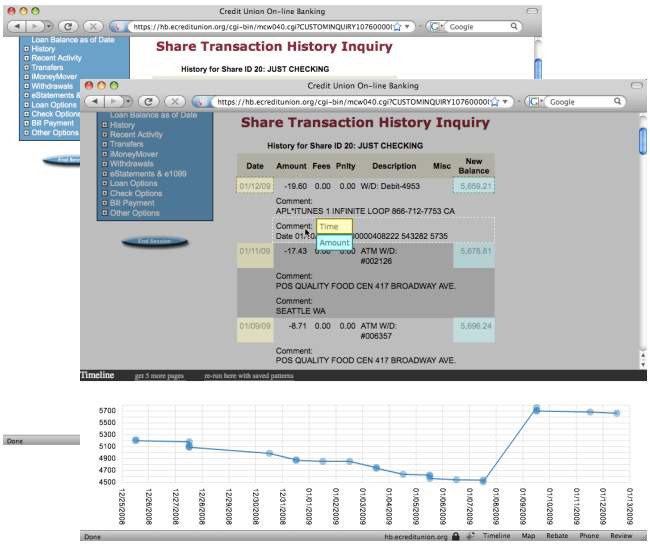

Figure 4: To plot her spending history, the user clicks the *Timeline* button, then clicks an example *time* and *amount*.

tied to the particular banking site, and only interacts with reform. To be used, it needs tuples containing a "time" and an "amount". reform includes a number of such datatypes, such as numbers, prices, names, and location addresses.

While the user is at her online banking page, she opens the timeline widget from a button at the bottom of her browser. reform checks its database of sites to see if the bank already has a timeline extraction pattern. If it does not, reform starts the interactive selection by example mode.

Prompting User for Datatypes

As the user moves the cursor around the screen, the DOM node under the current cursor location is outlined, and a floating menu appears next to it, prompting the user to specify if the node is a "time" or "amount" (Figure 5). This menu is always present, following the mouse and the node underneath the cursor.

Learning Extraction Pattern by Example

Since the user is plotting transaction dates and balances, she first moves the cursor to a transaction date and clicks "time". This marks the DOM node as a positive example for time, and gives it a dark yellow border. reform processes this example, highlights what it thinks are other times on the page, and forwards them to the timeline widget, which graphs them. The user now moves the cursor to a transaction balance amount and clicks "amount", giving it a dark blue border. reform processes the example "amount", highlights the other amounts, and passes the result to the timeline, which graphs the completed result (Figure 4). With two clicks, the user has taught the timeline to graph her financial history.

Negative Examples Disambiguate Layouts

This bank task requires only one example per data type. However, some pages with more ambiguity require additional learning. In fact, this banking page contains an inconsistency: for some transactions, the actual purchase date (*e.g.* "09/18/08") is recorded in a comment field at the bottom of the transaction, because the date column instead contains an internal posting date (*e.g.* "09/19/08").

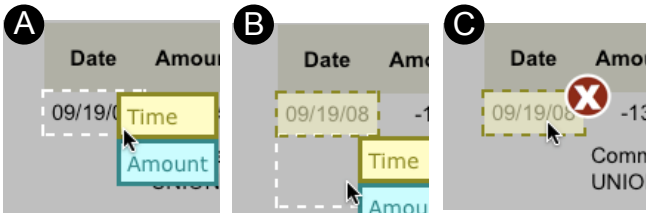

Figure 5: Specifying examples. (A) A menu for specifying examples follows the cursor, jumping to the node under the mouse. (B) After the user clicks "Time", the system marks the node as a positive example. Positive examples are bordered with the color of the field. Inferred nodes are highlighted but have no border. (C) When the user moves the cursor over a selected node, an X is displayed. The user can click on the X to provide a negative example, turning the border red (Figure 6).

reform can be taught this idiosyncratic information layout. When the user hovers over the undesired date "09/19/08", a red X appears (Figure 5C). The user clicks the X to mark the date as a negative example, and instead marks the correct comment date "09/18/08" as a time. Negative examples receive a red border. reform now correctly selects the date from all transactions with comments, but it ignores transactions without comments, because it no longer sees a usable "time" in them. To finish the task, the user moves the cursor to a missing date in a transaction without comments and marks it "time." These three additional examples successfully teach reform a rule that prefers the comment dates to the posted dates for transactions with comments, but uses the posted date when there is not a date in the comment (Figure 6).

MACHINE LEARNING EXTRACTION ALGORITHM

Given a set of user-specified positive and negative examples and a schema specified by the enhancement, the goal of reform's algorithm is to return a set of tuples that both match the pattern implied by the positive and negative examples and conform to the schema.

Let us clarify some terms before continuing. Each schema consists of a set of *fields*, such as {time, amount}. Each field has a *data type,* such as "date" or "number," and a *user-visible name* such as "flight start time" or "account balance." A *tuple* consists of a set of concrete DOM nodes, one for each field in the schema.

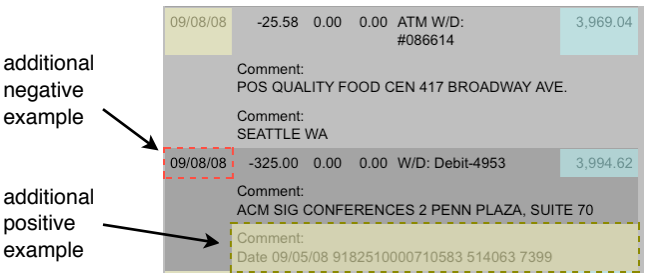

Figure 6: Adapting to heterogeneous layouts. The system was trained to prefer dates in "comments" fields over dates in the left hand column for transactions with dates in comments.

The algorithm includes two phases, which are analogous to the phases of lexing and parsing in compilers.

Phase 1: Lexing. The positive and negative examples provided by the user for each field are used to train a support vector machine [15]. This trained SVM then labels every node in the DOM with the degree that it "matches" the field, given the user's examples. Like compiler lexing, this phase analyzes the type of each node in isolation, without considering its relationship to the nodes around it.

Phase 2: Parsing. Given these isolated match strengths on each node, the second phase extracts a coherent table of tuples. It segments the webpage into tuple boundaries, associating nodes with high SVM scores together so that, for instance, a timeline can graph a time node and amount node as a single datapoint. Like compiler parsing, this phase infers a structural relationship amongst individual nodes.

Phase 1: Lexing a Match Strength for Every Node

Traditional end user extractors commonly use DOM path templates, such as XPaths, to identify extractable nodes. Our machine learning approach generalizes DOM path templates. Rather than derive one path per field, we derive many paths, along with attributes and values of the nodes at the ends of those paths. Our SVM then determines which paths are important by examining the positive and negative examples, and assigning each (path, attribute, value) triplet a variable weight. Thus, when there is only one example, all the paths have equal weights, and our extractor behaves similarly to an XPath, readily extracting regular structure. However, when the page has complex varying structure, inconsistencies, or noise, an XPath can fail, whereas our system takes additional examples from the user and adapts to the inconsistencies by variably weighting the plurality of alternative paths.

Compute Feature Vocabulary

To train a SVM we must first represent each node as a feature vector. We synthesize a new format for feature vectors, called a *feature vocabulary*, after every new example, to capture the features that characterize the new set of examples. Each feature represents a triplet (path, attribute, value), representing a concept such as "this node's parent's second child has an x coordinate of 33px", or ([parent, second-child], x coordinate, 33px). A node's feature vector will then be computed as an array of booleans, each true if and only if the node at the end of *path* from the original node has an *attribute* with the specified *value*.

We capture the following attributes at the end of each path:

- Spatial: x and y coordinates, width and height
- Matched datatypes: whether the node contains a date, time, number, price, or address, as recognized by reform
- The first and last 3 words of text contained in the node
- The DOM attributes id and class
- The node's index in its parent's list of children

We experimented with a few ways of generating and representing paths, and settled on simply counting hops in a post-order depth-first-search from the starting node, up to a distance of 10 nodes with left-first ordering and 10 with right, creating a total of 21 paths, including the empty path

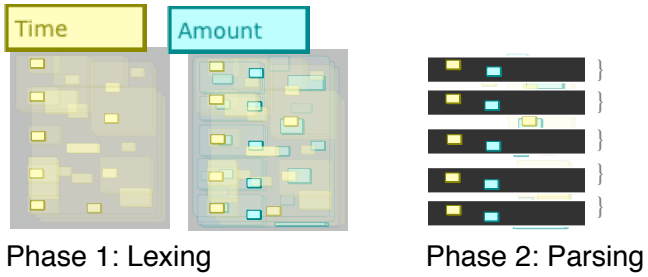

Figure 7: In the first phase of extraction, reform assigns each node in the webpage an individual match strength. In the second, it finds a coherent set of tuples in horizontal strips.

representing the node itself. We store paths according to the number of hops left or right, such as "4 left" or "2 right."

We create a feature vocabulary for each field with the following algorithm. For each example node, traverse each of the 21 paths and record each attribute's value at the end of that path. Then union all (path, attribute, value) combinations together, assigning an index to each. This becomes the feature vocabulary. Computing a feature vector for a node is then a matter of repeatedly traversing DFS *paths* to neighboring nodes, and comparing the *value* of each neighbor's *property* to the value in the vocabulary. If they match, the vector is set to true at that feature's index.

Training a SVM from the Examples

We compute feature vectors for each positive and negative example, and with them train the SVM. There is, however, one trick. Since each task begins with only positive examples, and the user may in fact never provide a negative one, we fabricate a *fake* negative example with every feature set to false. Since our feature vocabulary was created from positive examples in such a situation, and every feature was generated to be "like" an example, a vector with every feature set to false effectively represents "not like" the positive examples to the SVM. This simple trick works well.

Predict Match Strength of Every DOM Node

We then create feature vectors for each node in the web page, and predict their match score by measuring their distance from the SVM margin. At this stage, we also incorporate knowledge of the enhancement's schema, by automatically setting any node's score to zero that does not contain the correct datatype. For instance, if we are training a SVM to recognize "purchase price" on a webpage, we ignore all nodes that cannot be parsed as a price. We then normalize all scores to sit between zero and one. At the end of this process, every node in the tree has been labeled with its distance to the margin for every field in the schema. In the timeline scenario, every node would be labeled with its similarity to both "time" and "amount" (Figure 7).

Phase 2: Parsing an Optimal Table of Tuples

Now that we know how similar each individual node is to each field in isolation, we need to extract a coherent set of tuples, containing nodes with large match strengths. This is straightforward if the system only needs to find a single tuple per page, as XPath templates (*e.g.* Web Summaries [6]) and geometric coordinates naturally match a single

node per page. Systems that instead compute a variable match strength for each node from features (*e.g.* Chickenfoot [2], CoScriptor [17], reform) can return the single node with the largest match strength.

However, reform also supports finding *all* matching tuples on a page, which requires segmenting the page and relating nodes into a table of values—a more difficult problem. Existing end user extractors with support for multiple tuples per page, such as Solvent [21], Sifter [10] and Karma [22] use an XPath approach to segment the page into tuples, in effect replacing part of a node's XPath with a wildcard. For instance, they might substitute "third child of the second DIV" with "each child of the second DIV". Unfortunately, this requires a tree with structure such that a set of children precisely align with tuple boundaries. Some DOM trees are different, such as the online bank statement as visualized in Figure 8. Furthermore, XPath wildcards do not provide a straightforward approach for incorporating negative examples from the user to focus on different features and disambiguate extraction. The XPath approach is limited by relying on a particular tree structure to identify tuple boundaries, instead of examining the breadth of features available and allowing both positive and negative examples to be input by the user.

reform parses tuple boundaries independently of the underlying webpage structure and can work from any lexing phase that labels tree nodes with match strengths. We compute the segmentation with an optimization routine that looks at the way a page is displayed, rather than represented, and tries to slice the page into horizontal display strips that maximize a utility function over the best-matching fields in each strip. These best-matching fields within strips become tuples (Figure 7).

This algorithm makes the assumption that each tuple lies completely within non-overlapping vertical (*e.g.* y-start to y-end) regions. The next section describes layouts in which this assumption does not hold. Its goal, then, is to choose a sequence of y-coordinate segmenting locations at which to slice the page into tuples of maximum score. We calculate the score of a slice by finding the best-matching node for each field within the slice and summing their values. We then add the scores of all chosen slices to calculate the net utility of a page segmentation.

However, this flat sum over maximums encourages the creation of small tuples with low-scoring fields, since it is of higher value to sum two tuples with match scores *e.g.* (0.9, .01) and (.01, .0.9) = 0.91 + 0.91 = 1.82 rather than a single tuple covering both, with scores (.9, .9) = 1.8. To encourage clustering high-valued fields together into a single tuple, we additionally run each segment score through a gentle convex function. We find that cubing each segment score works well.

Now our task is to search over all possible segmentations to choose the one with the largest utility. Since the brute-force approach explores an exponential search space, we employ a dynamic programming approach to make it efficient. We also do not consider segments taller than 700 pixels. Finally, we filter out segments with scores below one half.

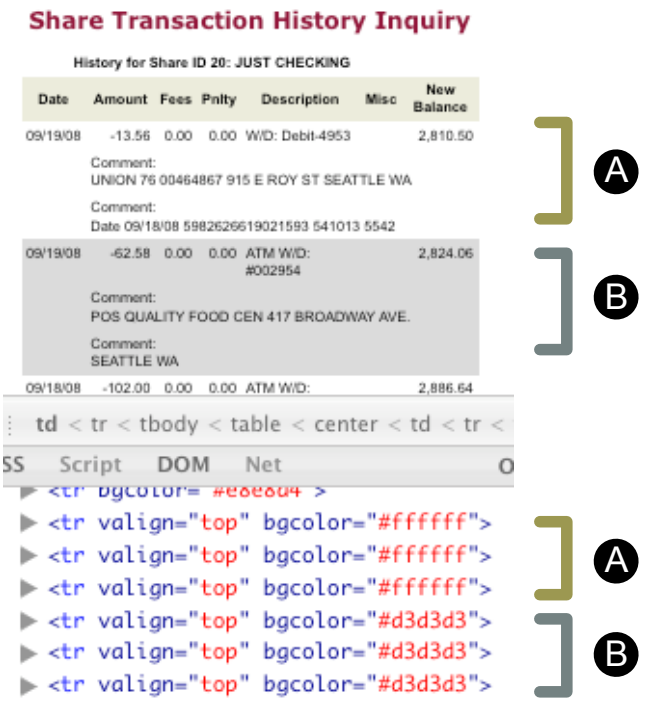

Figure 8: The bank account internals as displayed in Firebug. Notice that each tuple spans multiple <tr> nodes of varying background color, without a single enclosing parent. XPaths segment each element of a list along an enclosing parent, which fails when a tuple is composed of multiple children. They also have trouble when there are extraneous children or when tuples are at different depths in the tree.

Extracting Typed Data from Free Text

Recognizing entities, such as times, dates, numbers, and addresses in text is a known problem with many existing solutions. We use hand-coded regular expressions.

Enhancement Programmer API

Enhancements use reform by defining a schema and calling reform.start_selection(schema, callback, interactive). A schema consists of an enhancement's name, such as 'Timeline,' an array of data types for each field of the tuples, such as 'number,' 'timedate,' 'address,' 'link,' or '*,' and a matching array of user-visible names for each field. Each time the user provides an example, the system infers new tuples and passes them to the callback. If interactive is set to true, reform will force a new interactive selection session. Otherwise, it will first try to find an existing pattern for the website in its database.

Implementation & Performance Details

reform is implemented with a Javascript and XUL user interface in Firefox, which serializes the DOM to a Python web server to perform the feature generation, string recognition, and tuple fitting. This web server delegates the raw machine learning task to a native binary, communicating over files. Extraction patterns are automatically stored on the web server, storing one pattern per enhancement per website domain name. reform does not yet share patterns across users. The enhancements use Prefuse Flare [13], Google Maps, and the iPhone iUI library.

The prototype is almost fast enough for production use. Processing the first example generally takes a second or two. Since each example has a new set of (path, property, value) tuples, and thus features, the size of feature vectors grows proportionally to the number of examples provided by the user. After 5 or 6 examples, updates can take 3-10 seconds to process. The bulk of this time is spent in inter-process communication, writing feature vectors to disk for the SVM; wasted time that would be eliminated if reform were rewritten to run in a single process.

Limitations in the Algorithms & Future Work

reform cannot yet work on all website layouts. However, by architecting the extraction process into separate lexing and a parsing phases, we can extend the algorithm piece-by-piece to overcome its limitations. For instance, we can solve problems in parsing without affecting lexing. In this section we describe the current algorithmic limitations.

Limitations in Phase 1: Lexing Individual Fields

We assume each field is separated by a node boundary. Sometimes fields are separated by text patterns, such as with "$100 - Microwave". Bigham describes one approach we could adopt, in which additional node boundaries are inferred before learning [3]. Regions could also be interactively selected, and learned, as a phase that occurs before phase 1. Finally, sometimes a user wants a single field to be composed of multiple disconnected regions, such as when selecting nodes to include in an iPhone web summary.

Limitations in Phase 2: Fitting a Layout of Tuples

Our algorithm can find a best *single* tuple for a page, or find a *list* of tuples. However, we only support vertical lists, since the parser uses horizontal strips to segment tuples. Web data can also be laid out in a horizontal list, a two-dimensional grid (*e.g.,* some product pages on Amazon and some photo pages on Flickr), or a more complicated nested combination of horizontal lists of vertical lists and grids, which we call newspaper layout because it is common on the front pages of newspapers. reform cannot learn these layouts. However, it would be straightforward to extend our existing algorithm to horizontal layouts by running it horizontally instead of vertically, and grid layouts by recursing.

We also fail to support nested schemas, *e.g.,* containing lists of tuples of lists. For instance one might want to scrape a list of photos on Flickr, where each photo has a list of tags. One could represent such a schema as (photo, (tag)).

A dual problem is extracting lists with headers. For example, a database of calendar events might have the schema (date, time, description), but display the events by date separated by headers. This could be represented with the nested schema (date, (time, description)). By shifting between schema representations, the same parsing algorithm could handle both headered lists and nested schemata.

Extracting Information from Unstructured Text

We consider natural language extraction from unstructured text, such as learning facts from a chapter of Shakespeare or a forum post, to be a much different problem than semi-structured data extraction, and do not support it. Other solutions could be integrated with our system in the future.

EVALUATION

We built five enhancements and ran two small studies to evaluate reform's architecture, algorithms, and user interface. The enhancements exercised the space of possibilities with our architecture and API, and uncovered areas for improvement. Our algorithm study elicited cross-site generalizability: on how many sites does it work, and how frequent is each failure mode? Our user study tested our interaction goal: can novice end users successfully attach enhancements to websites?

Five UI Enhancements

To validate our architecture we built five demonstration interface enhancements. They exercise a space of enhancements one can implement with reform. We chose the suite to cover the following breadth of web UI enhancement categories: (1) mashups; (2) static->dynamic AJAX upgrades; (3) visualizations; (4) additional UI features; (5) mobile web facades; and (6) aggregations. The enhancements in this section that cover these categories, in turn, are: (1) remap, rebate; (2) resume; (3) revisit, remap; (4) rebate; (5) reduce; (6) rebate.

Remap: A Universal Map

Maps are consistently the most popular, well-known, and frequently-created mashups. Of all the mashups on programmableweb.com, 38% are tagged "map." In Zang's survey of mashup developers, he found that 77% had built mapping mashups and 96% had used the Google Maps API [27]. We created a single mapping enhancement that end users can apply to any website.

Suppose, for instance, that an end user is browsing nucleartourist.com and finds a list of nuclear reactor locations in the United States. The page lists the precise address of the reactors, but the user would reach a much richer understanding of the bulk of data if the website provided a map. The user can click on the *remap* button at the bottom of his browser, the page fades to grey and the blue and yellow selector menu prompts him for "Address" and "Details", as can be seen in Figure 2. He selects the address, and the system plots it on a map along with any other addresses it finds on the page and highlights those addresses in the web page. He then highlights one of the reactor names, and selects "Details". This teaches remap that the name should appear in a caption bubble when clicking on the reactor's icon. After two more examples, the system has learned a pattern for extracting every plant name and address from this page and can plot them on the map.

In another example, Figure 9 shows how we can modify a government website of sex offenders in Buda, TX to include an easily accessible map.

Rebate: Aggregating Websites to Save Money

rebate is a two-phase application. First, end users extract feeds of coupon & rebate offers and other special deals. rebate can then display these deals whenever the user is buying something matching the deal. There are many websites that aggregate different types of coupons and deals. rebate users can aggregate deals from retail and deal sites into a single database and stay informed while shopping.

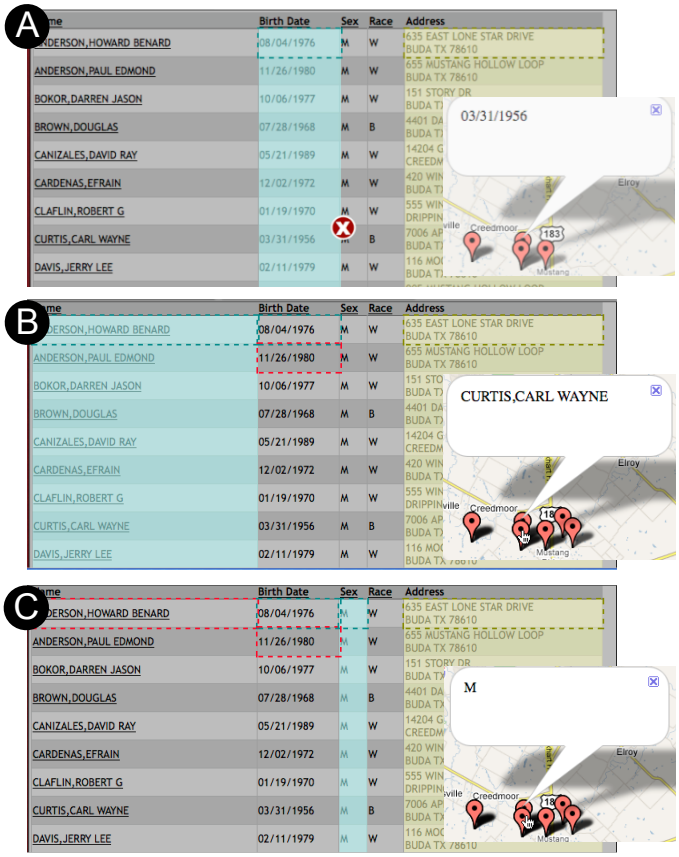

Figure 9: Plotting Buda, TX sex offenders on remap**. By progressively changing the positive and negative examples, this user can view different data from the table in the detail bubble of each pin on the map.**

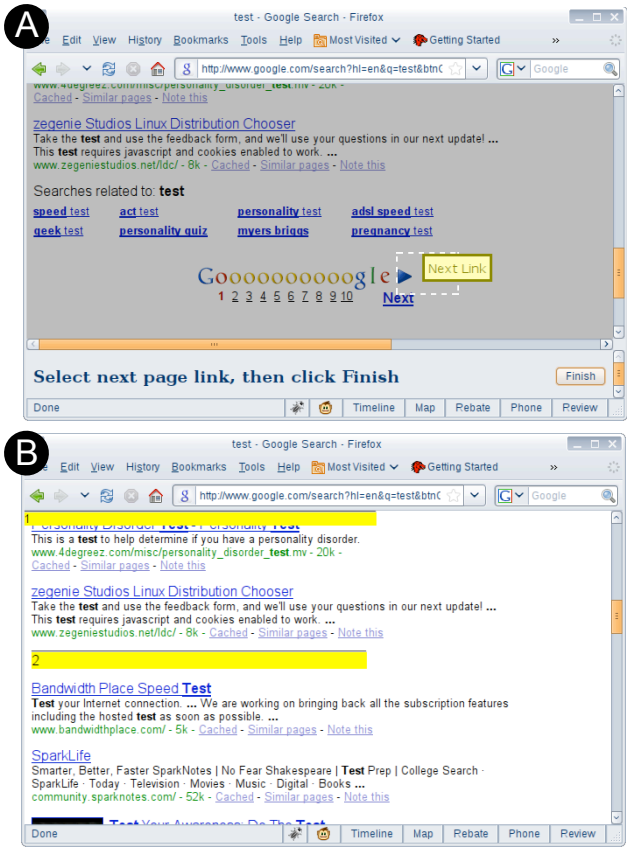

Figure 10: The auto-paging resume **enhancement. (A) The user specifies a page region and a "next page" link. (B) The system then provides an infinite scrolling page with yellow headers, inspired by the iPhone's infinite scroll pane. When a header scrolls to the top of the pane it sticks.**

Reduce: Mobile Website Generator

We increasingly browse the web on cell phones, yet only a small proportion of websites offer cellphone versions. The reduce enhancement makes an iPhone-specific version of any website. It prompts the user to teach it a sequence of (title, summary, link) tuples, and uses this content to fill in a template with the iPhone look and feel.

Resume: Improving the Ubiquitous "Next Page" Link

Our resume enhancement is an example of using reform to change a fundamental web interaction widget. resume replaces "next page" links, ubiquitous on the web, with AJAX that automatically fetches the next page and stitches it into the current one. Users scroll from one page to the next, with a "Page 2" header in the middle, rather than clicking and waiting for a page refresh (see Figure 10).

Revisit: A General Timeline Visualization Enhancement

We described the revisit enhancement in a previous section. It allows end-users to attach a timeline to any data that accumulates quantities over time such as bank statements or sales statements.

These enhancements demonstrate how reform can enable visualizations, facades, mashups, aggregators, and new interactive widgets to be written once and applied to many websites via end user web extraction.

Algorithm Generalizability

To evaluate how well our algorithms generalize across different websites, we tested reform on a set of thirty websites. To avoid bias in selecting test pages, we used Mechanical Turk [1] to recruit anonymous Internet users and paid them 20-50 cents to collect pages from the Internet for us. These users had not used reform previously. They were told only that we were testing timeline visualization software and needed webpages containing times or dates and amounts. We displayed the bank account as an example page, along with a screenshot of a reform timeline for the page, to make the task more concrete. We applied the timeline enhancement to all of the pages they found and counted the number of examples necessary to visualize the data. If reform failed to extract the proper pattern, we noted the reason that it failed. We classified each failure as whether it was caused by an unsupported layout format (nested, horizontal, grid, newspaper), by an unsupported node boundary format, or by other unexpected problems.

Generalization Results

Of the thirty sites, reform successfully extracted timeline data for twenty, or 67% of them. Of the failures, two had horizontal data layouts, four had grid layouts, four had datums separated by whitespace instead of node boundaries, and one had data displayed with images instead of parse-

able text. The successful pages required an average of 2.5 examples, or 1.3 per field since the timeline schema has two fields. These results show that if we extend reform's second phase parser to handle new layouts and the first phase lexer to support non-node whitespace datum boundaries, we could extract timelines for most pages on the web. We have not run generalization studies for the other enhancements.

Task	URL	User #						
		1	2	3	4	5	6	7
Map	http://www.yellowpages.com/?search=record+store	4	4	4	10	5	3	5
	http://www.nucleartourist.com/us/address.htm	2	3	5	5	4	6	8
	USPS branches (site no longer available)	2	2	4	14	2	10	2
Timeline	Community First Credit Union	2	2	2	6	2	8	2
	http://digg.com/	5	6	4	5	11	12	2
	http://www.npg.org/facts/us_historical_pops.htm	2	2	2	2	9	6	2
iPhone	http://news.google.com/?ned=us&topic=el	7	11	4	2	4	18	3
	http://digg.com/	6	6	4	5	3	6	3

Figure 11: Number of examples (clicks) for users to complete each task. Shaded box means the user failed to complete the task. All others were completed successfully.

User Study

Making expressive extraction algorithms accessible to novice users is a difficult problem. To evaluate our approach, we recruited novice end users and asked them to attach our enhancements to a variety of websites with reform. By the time of the study, we had built the remap, revisit, and reduce enhancements. For each enhancement, we asked an assistant unfamiliar with the extraction algorithm to select two test websites to use in our tasks. We then selected an additional website for each enhancement to create a test set of 9 websites for the 3 enhancements. We ensured that for each enhancement the pages ranged in difficulty, so that we could observe corrective behavior from users. We only used pages with layouts that could be processed with reform. We canceled the last of the three iPhone tasks, because a bug made it impossible to complete, resulting in a total of 8 enhancement tasks.

We recruited seven end users to attempt our enhancement tasks and offered them five dollars to complete a five to ten minute study. Three subjects were female and four were male. We screened subjects to ensure that they were Internet users, but did not understand HTML.

The study began with the facilitator demonstrating the use of remap on a Washington State website with addresses for driving license office locations. This introduction lasted approximately one minute. Then the subject was tasked with using remap, revisit, and reduce on each of the websites for a total of 8 tasks. The subjects could not ask questions during the tasks until they gave up, at which point the facilitator recorded the task as a failure. If a subject simply did not notice an errant missed or extra inference, the facilitator helped the subject and tallied the oversight but did not mark the task as a failure. The study was conducted on a 2Ghz 13" Macbook laptop computer using a trackpad. Websites were used live from the Internet, and thus could vary across users.

User Study Results

Most of the tasks (86%) were completed successfully, and the failures were isolated to only two subjects who had much more difficulty than the others. The average number of examples needed to successfully complete a task was 4.4 clicks, with a standard deviation of 2.6 clicks. More detailed results are shown in Figure 11.

There was large variance across users, most of whom had never seen a programming by example interface prior to this study. Two users understood the reform concepts immediately and completed the tasks readily, skipping instructions. Three completed the tasks at a moderate pace. Two had difficulties that appeared to stem from basic misunderstandings. User #6 did not realize that the system learned from his examples and inferred selections for him, and instead assumed he needed to manually select every datum on the page as one would do for a normal Google map. When he tried to select an inferred node, it displayed a red X to allow a negative example, which confused him, and he sometimes clicked the X and sometimes selected a nearby node instead. He mentioned he did not understand what the X meant. However, he seemed to understand how the system worked by the time he completed the final task. User #4 also did not understand what the red X meant during her first tasks, thinking it was trying to tell her that something was wrong, instead of affording the ability to correct mistakes. Her understanding also appeared to improve slowly over time. In addition to the X, both users also sometimes seemed not to understand what the blue and yellow highlights meant. We suspect these graphics could be clarified. Nonetheless, we found it encouraging that users had few other problems when they understood these basic concepts.

The number of examples required varied across enhancements and websites. Digg and Google News, for instance, required additional learning to overcome internal variations when different stories had different imagery and layouts. reduce required more examples than other enhancements partially because the summarization schema does not contain data type constraints such as "date." Training also requires more examples if users provide slightly different examples: for instance, every date might be enclosed within two nodes, with one slightly larger than the other, and a user might click on these nodes differently in different tuples. We were surprised to observe, however, that even if users gave unknowingly erroneous training, such as marking a date as amount, the SVM was flexible enough to recover the correct pattern after additional examples: a single bad data point would eventually be overruled. We also noticed that users would sometimes restart extraction to get a clean slate if they accidentally gave multiple incorrect examples and subsequently saw strange inferences from reform. The data in Figure 11 aggregates the number of examples given before and after restarts. We did not record task times, but estimate tasks took anywhere between 20

seconds and two minutes once users understood and were comfortable with the interface.

Many users asked if they could download the program. Multiple users said they would like to use the visualizations in education, for use by their students, children, or selves. This study verifies that reform's expressive by-example extraction is accessible to novice end users, and their comments suggest that some may have the motivation to use it.

CONCLUSION

We present reform, a prototype tool with which novice end users can attach web user interface enhancements to new websites. reform presents a new architecture for enhancement and interactive technology for web extraction. End users in a small study were able to successfully use the system. We believe reform can be extended to support a much broader class of web pages with straightforward modifications to its extraction algorithms.

ACKNOWLEDGEMENTS

We thank Nathan Morris for help running the user studies, Ravin Balakrishnan for early feedback, and the anonymous reviewers for their comments that improved this paper. This work was supported under a National Science Foundation fellowship and a Microsoft Live Labs internship.

REFERENCES

1. Amazon Mechanical Turk. (http://www.mturk.com)
2. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customiztion of rendered web pages. *In Proc UIST 2005,* ACM Press(2005), 163–172.
3. Jeffrey P. Bigham, Anna C. Cavender, Ryan S. Kaminsky, Craig M. Prince and Tyler S. Robison. Transcendence: enabling a personal view of the deep web. *In Proc IUI 2008*. ACM Press (2008), 169–178.
4. Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, Pat Hanrahan, Jeff Klingner, Alon Halevy and Luna Dong. Vispedia: interactive visual exploration of wikipedia data via search-based integration. *In IEEE Transactions on Visualizations and Computer Graphics 14*, 6. (2008), 1213–1220.
5. Dapper. (http://dapper.net)
6. Mira Dontcheva, Steven M. Drucker, Geraldine Wade, David Salesin, Michael F. Cohen. Summarizing personal web browsing sessions. *In Proc UIST 2006.* ACM Press (2006), 115–124.
7. Robert J. Ennals and David Gay. User-friendly functional programming for web mashups. In *Proc ICFP 2007*. ACM Press (2007), 223–234.
8. Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog and Sergio Flesca. The Lixto data extraction project: back and forth between theory and practice. *In Proc PODS 2004*. ACM Press (2004), 1–12.
9. Greasemonkey (https://addons.mozilla.org/en-US/firefox/addon/748)
10. David Huynh, Robert Miller, and David Karger. Enabling web browsers to augment web sites' filtering and sorting functionality. *In Proc UIST 2006*. ACM Press (2006), 125–134.
11. Björn Hartmann, Scott Doorley and Scott R. Klemmer. Hacking, mashing, gluing: understanding opportunistic design. *In IEEE Pervasive Computing 7,* 3 (2008), 46–54.
12. Björn Hartmann, Leslie Wu, Kevin Collins and Scott R. Klemmer. Programming by a sample: rapidly creating web applications with d.mix. *In Proc UIST 2007.* ACM Press (2007), 241–250.
13. Jeffrey Heer, Stuart K. Card and James A. Landay. prefuse: a toolkit for interactive information visualization. *In Proc CHI 2008*. ACM Press (2008), 421–430.
14. Utku Irmak. Interactive wrapper generation with minimal user effort. *In Proc WWW 2006.* ACM Press (2006), 553–563.
15. Thorston Joachims, Making large-scale SVM learning practical. *Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf and C. Burges and A. Smola (ed.), MIT-Press (1999).
16. Alberto H.F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva and Juliana S. Teixeira. A brief survey of web data extraction tools. *In ACM SIGMOD Record 31,* 2 (2002), 84–93.
17. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber and Eser Kandogan, Koala: capture, share, automate, personalize business processes on the web. *In Proc CHI 2007*. ACM Press (2007), 943-946.
18. Netcraft Web Server Survey. (http://news.netcraft.com/archives/web_server_survey.html)
19. Jeffrey Nichols, Zhigang Hua, John Barton. Highlight: a system for creating and deploying mobile web applications. *In Proc UIST 2008*. ACM Press (2008), 249–258.
20. Solvent. (http://simile.mit.edu/wiki/Solvent)
21. Rattapoom Tuchinda, Pedro Szekely and Craig A. Knoblock. Bulding mashups by example. *In Proc IUI 2008*. ACM Press (2008), 139–148.
22. Ubiquity (http://labs.mozilla.com/projects/ubiquity/)
23. United States Bureau of Labor Statistics. (http://www.bls.gov/oco/)
24. Jeffrey Wong and Jason I. Hong. Making mashups with Marmite: towards end-user programming for the web. *In Proc CHI 2007*. ACM Press (2007), 1435–1444.
25. Yahoo! Pipes. (http://pipes.yahoo.com)
26. Nan Zang, Mary Beth Rosson and Vincent Nasser. Mashups: Who? What? Why? *Ext Abstracts CHI 2008*. ACM Press (2008), 3171–3176.