On the maintenance of UI-integrated mashup applications

Maxim Shevertalov and Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{max, spiros}@drexel.edu

Abstract—This paper describes the maintenance of long-lived mashup applications that are integrated at the user interface (UI) layer. It presents techniques that help mashup developers to maintain applications by identifying when and how the original applications’ UIs change. It describes a novel mashup editing environment that can be used to create, share, and edit mashups. This paper also presents an experiment that demonstrates our approach’s ability to track UI changes as an application evolves and a demonstration of the effort expanded by developers to maintain mashups as the applications used by the mashups evolve.

I. INTRODUCTION

Mashups are applications that integrate features of other software applications [1]. For example, the Mapskrieg mashup (www.mapskrieg.com) combines Google Maps (maps.google.com) and Craigslist’s (www.craigslist.com) apartment listings to present users with an annotated map containing property locations, pricing, and contact information.

A key advantage of creating mashup applications, instead of creating applications from scratch, is the ability to reuse familiar features. For example, rather than creating a brand new mapping application, the developer can reuse existing and familiar mapping applications such as Google Maps or Yahoo Maps (maps.yahoo.com). By doing so, not only does the developer no longer need to understand the intricacies of creating a mapping application, but he is also able to include a user interface familiar to many of the mashup’s users.

This paper focuses on the maintenance issues that arise when integrating applications into mashups at the user interface (UI) layer, as opposed to the process or data layers. Techniques are described to help developers maintain mashups by identifying when and how the original applications change. The technique is validated using an experiment that demonstrates our approach’s ability to track UI changes as an application evolves and a demonstration of the effort expended by developers to maintain mashups as the applications used by the mashups evolve.

The goal of this work is to focus on mashups that are created by integrating features at the UI layer. We have created a web-based mashup environment named MashDesk that enables developers to specify integration points by identifying UI elements and chains of actions that are necessary for the operation of the mashup. For example, the feature of adding a point to a map object can be invoked in the provided UI by identifying which fields are involved in the process, how the data should be formatted, and the chain of actions that needs to be triggered in order for the mapping application to accept the input. Because MashDesk is a web-based environment, the focus is on web technologies such as HTML [20], CSS [2], and Javascript [9].

There are many instances where an integration involving the UI layer is preferable to using APIs. An obvious example is when an API does not exist or does not offer all of the functionality of the application. However, even if a good API exists, enabling integration to occur at the UI layer enables less experienced developers or, even regular users with little or no programming experience, to create applications by allowing them to compose applications using familiar UI constructs. This is an important goal of the MashDesk environment.

One of the goals of MashDesk is to allow developers to share their applications. These applications could be standalone or mashups that depend on other applications to be present in the environment. MashDesk does not distinguish between applications and mashups, but instead treats them all the same.

Notice that, because mashups are treated as any other application in MashDesk, a mashup can be composed from the features of other mashups. Thus, any mashup may have several layers of dependencies that can change as the applications used by the mashup evolve. In addition, unlike an API, which explicitly exports functionality, the developer has little incentive to keep their UI constant or inform others when it changes. This presents a unique maintenance issue, especially when dealing with applications integrated at the UI layer.

There are many aspects of a UI that the original developer may change. These changes effect the resulting mashup with varying degrees of severity. Developers may alter only the look-and-feel of the original application. These changes do not effect the mashup in any significant way because the underlying application has not changed.

Application developers may, and typically do, go beyond cosmetic changes as their applications evolve. Developers may choose to move or rename UI widgets. These changes effect
how the application is presented to the user, but do not alter its functionality. In these cases, the mashup has to update its references to any of the UI widgets that have changed. To do so, the user must first identify the new locations of the UI widgets used in the mashup. MashDesk employs a novel technique to assist the mashup developer in this task. It is discussed in Section IV.

Application developers may go a step further and drastically alter their original applications. Meaning that they change the process and usability of the application. Application features may be removed or modified such that the user is required to perform different actions to access them. Also, new features may be added. It is important for the composed mashup to be modular enough so that only those features that have changed need to be addressed by the mashup developer.

This paper focuses on the maintenance concerns that are unique to mashup application integrated at the UI layer. The rest of this paper is organized as follows; Section II provides background information about mashup applications; Section III provides a general overview of the MashDesk environment; Section IV describes UI widget tracking and presents an experiment tracking select widgets through several years of websites’ evolution; Section V presents a demonstration in which a mashup application has to evolve due to a change in the UI of an application it uses; finally Section VI outlines concluding remarks and future work.

II. BACKGROUND

The concept of mashups has been popularized in the web applications domain because many well-known web based systems, such as Google Maps (maps.google.com) and Flickr (www.flickr.com), have released their APIs to the public. This enables developers to leverage existing web technologies and create new applications. By restricting the development process to feature composition, mashups enable developers to rapidly create custom applications aimed at niche audiences [10], [12].

It is convenient to separate the types of mashups into the following three categories: data, user interface (UI), and process. Data mashups combine two or more data sets to create a new data set. UI mashups combine familiar UI elements to create new applications. Process mashups combine two or more processes into a single execution. It is also worth noting that most mashups fall into more than one of these categories or can be implemented in different ways. For example, consider a data mashup that combines an apartment listing data source with a crimes report data source. That is only one of the ways such an application can be created. Another is by creating a process mashup that, given a data set of apartment listings, makes calls to an application that returns crime reports for a specified area.

Several tools and approaches that foster mashup creation have been developed recently. These range from writing “glue” code to paradigms such as visual programming [3], [17] and programming by demonstration [6].

Zang et. al’s survey [28] focuses on the experience of producing mashups. They found that the biggest problem mashup developers face is the reliability of the APIs they use. Participants in the survey described APIs as having authentication, performance, and dependency problems. Wong et. al’s survey [27] focuses on the types of mashup applications being developed. They found that the majority of applications supported data visualization. In fact, 40% of the applications presented geolocation data on a map.

Visual programming techniques [3], [17] were created to help users with little or no technical expertise to develop software. They are typically centered around the concepts of graphs where application features are nodes and edges represent data and control flow. An alternative approach uses lists to represent a sequence of actions and containers to represent behaviors such as loops [21], [22].

Tools such as Microsoft’s Popfly [27], and Yahoo’s Pipes [8], are commonly used to create mashups. These tools provide their users with several modules representing popular web applications such as Google Maps and Flickr. The mashup is represented as a directed graph where nodes represent the included modules and edges represent data and control flow between these modules.

Another approach to mashup creation is Intel’s Mash Maker [7]. This tool was created as a plug-in for the Firefox [16] web browser. It extracts information from rendered web pages and provides users with suggestions of how that information can be used. For example, a user can specify how a street address on a web page is presented and instructs Mash Maker to use that address as input into the search box on GoogleMaps, thus annotating the web page with map information. It is a similar concept to the visual programming tools. However, it does a better job of hiding the programming environment.

While these tools simplify the mashup creation process, they still require a high degree of familiarity with the provided components. Even though a user may know which functionality of what applications he wishes to compose, it is not immediately apparent how that functionality will be presented in Popfly or Pipes. The functionality may have been decomposed into several different actions, or may not have been provided at all.

A different approach to mashup creation, and one that expands on the Mash Maker techniques, is to incorporate ideas from the programming by demonstration paradigm [6]. Programming by demonstration is an attempt to enable less experienced users to create their own applications and algorithms by demonstrating behavior. To employ programming by demonstration, users are asked to demonstrate how a program would behave in certain scenarios. After a few demonstrations, the programming by demonstration engine is able to generalize the user’s actions into a program or algorithm [14], [18].

Karma [25], [26] and Vegemite [15] are two tools that employ programming by demonstration to create data mashups. Users demonstrate to these tools how specific data management tasks are performed and the tools generalize them. For example, during data cleanup, the user is asked to demonstrate
Fig. 1. Illustrating the process of creating mashups. (a) The developer forms a connection between two applications. (b) The developer connects the page variable exported by the first application to a table of $x$ and $y$ values imported by the second. (c) The developer demonstrates how to extract the $x$ and $y$ values from the page variable. By clicking on two cells for each column. Shown here is the demonstration for the $y$’s column. Dark green cells are those selected by the user, while light green cells are included automatically. (d) The generated mashup.

how a single field in a table should be manipulated and, based on that demonstration, Karma cleans up the rest of the data.

The MashDesk environment expands on these ideas further. It supports different developer types by providing different ways that mashups can be created. Developers can use regular programming, visual programming, and programming by demonstration. By demonstrating how applications interact, mashup developers can integrate applications at the UI layer. That is the focus of this paper and the following section will provide an overview of the process.

III. MASHUP EDITOR OVERVIEW

This section provides an overview of the MashDesk environment. The MashDesk environment arose from our previous work in composing networked application [23], [24]. Because this paper is focused on the maintenance concerns of mashups integrated at the UI layer, a comprehensive description of the MashDesk environment is out of the scope. Instead, this section provides a general overview of MashDesk with a focus on mashups that are integrated at the UI layer.

The MashDesk environment allows developers to share their applications with other users. These application can be standalone or mashups. While both types are presented to the MashDesk user in a uniform manner, special care in managing dependencies is taken when deploying mashup applications.

When creating an application, the developer has an option of specifying an API. This API consists of data that the application can export or import and actions that can trigger an import or export event.

Mashup developers can create mashups directly within the environment. Figure 1 illustrates one of the mashup creation processes. It begins with the developer entering “mashup mode”. He proceeds to draw connections between the applications to be combined. Once drawn, the links can be edited. If the original application’s developers provided an API, mashup developers can use it by connecting the output from one application to the input of another. Developers can also specify which events on the source or sink applications will trigger a data transfer.

While the API is the preferred way to construct mashups, it is not always available. Therefore, every application
exports the page variable which gives the mashup access to all of the UI elements contained within that application. Given the page variable, the mashup developer can demonstrate how to extract data from it. This process is similar to that of Karma and Vegimite, both described in Section II. Each element in the page is identified by a unique XPath query [4], [5]. Once an XPath query for an element has been determined, it can be generalized by replacing identifiers with wild-cards. For example, the XPath query /table[@id='main']/tr[1]/td[4] references the fourth cell in the first row of a table with an id value of “main”. This query can be generalized with /table[@id='main']/tr[*]/td[4] to reference the entire fourth column, or with /table[@id='main']/tr[1]/td[*] to reference the entire first row, or with /table[@id='main']/tr[1]/td[*] to access all of the cells in the table.

The user demonstrates selection by clicking on elements in the page thus setting them as either positive or negative exemplars. With each positive exemplar, MashDesk creates a new XPath query that at least returns all of the positive exemplars. For example, if the developer first clicks on a single table cell, MashDesk will use the XPath query that references only that table cell. Then, when the developer clicks on another cell in the same row, MashDesk will generalize those queries, by replacing clauses that differentiate them with wild-cards, thus creating an XPath query that selects the entire row.

While the positive selection process is similar to the processes employed by Karma and Vegimite, MashDesk’s handling of negative exemplars is an important extension. To allow for negative exemplars, MashDesk introduces the concept of description labels. When a new positive exemplar is selected and the generalized XPath query is recalculated, several description labels are attached to each element returned by the query. When a new negative exemplar is selected, the description labels are used to remove elements from the selection automatically.

Description labels are generated by specialized functions in MashDesk. Each function takes an element as input and returns an array of description labels. The returned array may be empty. MashDesk calls each description-label-generating function one at a time and concatenates all of the results. Currently several description-label-generating functions have been implemented. They return labels based on the element’s type, the element’s content’s type, and the content of related elements.

One of the description-label-generating functions that is worth mentioning is named xpath_labels(). It returns labels based on the contents of elements related to the passed in element by generalizing the passed in element’s XPath query. The labels are returned in the format “xpath:<number of XPath query replacements> : <contents of element>”. For example, given that “/table/tr[3]/td[2]” is the XPath query for the negative exemplar cell in Figure 2, it can be generalized with a single change into “/table/tr[*]/td[2]” and “/table/tr[3]/td[*]”. The first query selects the second column, while the second selects the third row. Using the second query, the xpath_labels() function will return the following three description labels: “xpath:1:B”, “xpath:1:2,” and “xpath:1:2.2”. The first of which is used to eliminate cells (1, 4) and (1, 7).

The data extraction techniques described above are useful in extracting data from a single page. However, not all applications have all of the required data on one page. Many applications require the user to interact with UI elements to display the required information. Take the case of applications returning results based on a search string. The page that contains the list of results may not contain all of the information required by the mashup developer. One may have to click on individual search results to access the needed information. This is the case with the geocaching applications used in the demonstration in Section V.

To deal with such a scenario, the MashDesk mashup environment employs programming by demonstration. Mashup developers are asked to demonstrate how the data is generated. They begin by navigating to the starting page of the application. Next they perform the same steps they would as a user trying to extract information. Once a page with the necessary information has been reached, developers will enter data extraction mode and identify the elements to be extracted. Once the data has been identified, they may continue to other pages or hit the back button to signify an iteration in a loop of data extractions.

For example, a developer may wish to gather information such as name, email address, and phone number of people that meet a certain search criteria. The developer may also have access to a website that allows him to search for people based on that criteria, but the search results page only contains full names. The application’s user has to click on each name to find an email and phone number associated with it. To handle this case, mashup developer will have to demonstrate the process of data extraction to MashDesk. This process is illustrated in greater detail in the demonstration in Section V.

So far this section has only mentioned the creation of

![Fig. 2. An example of data extracted from the an HTML table. Dark green elements are positive exemplars. Dark red elements are negative exemplars. Light green elements are included by MashDesk. Light red elements are positive exemplars. Dark red elements are negative exemplars.](image)
output from an application. However, to create mashups one must also instruct applications to process input. For example, suppose an application that draws a plot when presented with a series of points, entered one point at a time. Using such an application in a mashup will typically require that the application’s developers provide an API capable of accepting a series of points. If such an API is not provided it will present a challenge to the mashup developer. The MashDesk environment solves this problem by enabling the mashup developer to demonstrate how a series of points can be used by the plot drawing application.

The task of demonstrating how data should be processed by an application is similar to that of demonstrating how the data is collected. The mashup developer uses the application’s interface to demonstrate the task of processing each data point individually. As the developer does so, the MashDesk environment detects input variables and loops in the process. Input variables correspond to values entered into various form fields and loops are discovered by analyzing the chain of events and finding repeating sequences.

As the mashup developer demonstrates the process, the MashDesk environment generates pseudo code for a program that the developer can check. Once the developer judges that the pseudo code captures his intent, he concludes the demonstration and the MashDesk environment generates the appropriate API interface.

In addition to being used for integration, UI widgets are also used to construct new interfaces for the created mashups. Once the integration process is finished, the mashup developer has the option of creating a new UI for his application. The new interface is mostly composed of widgets extracted from the applications being combined. This helps the mashup developer to hide the integration points and only present those UI widgets that are necessary. A good example of this is the case of combining an application that generates data in the form of \( x \) and \( y \) coordinates with an application that creates a plot given a data set that contain \( x \) and \( y \) coordinates. The final mashup only needs the UI widgets that are involved in data generation and displaying the actual plot. The widget that displays the generated data and the widgets used to plot points are unnecessary and can be discarded in the presentation of the mashup. Figure 3 illustrates this process.

The developer begins the process of creating a new UI for his mashup by entering “UI edit” mode in the MashDesk environment. Once in edit mode, the developer can drag UI widgets from any of the applications participating in the mashup onto a new application frame. Notice that in Figure 3b, the UI widgets present in the new canvas on the right, are no longer present in the original frames. He is able to control many properties of the widgets added to the frame, including their overall style, their position in the frame, and their behavior during a resize event. Once all of the widgets are organized and the developer exits the UI edit mode, all UI widgets that are not added to the new interface are hidden from view and become inaccessible to the mashup’s user.

As demonstrated by this section, UI elements are integral
Fig. 4. A sample of HTML code and the corresponding tree representation.

to the mashups composed in the MashDesk environment. They play a role in the integration of applications and can be made accessible to application users in a new mashup interface. Given their prominent role, tracking the evolution of UI widgets is important to the maintenance of mashups.

IV. UI Widget Identification and Tracking

As pointed out previously, the ability to identify and track UI widgets is integral to the maintenance of mashups integrated at the UI layer. This section describes that process and presents an experiment demonstrating its effectiveness.

When the mashup developer selects a widget, several properties used to identify it are stored. The most important property is the XPath query that references the widget. In addition to the XPath query, MashDesk stores the object’s type, the data it contains, and local identifiers. While the first two are self-explanatory, the third property needs further explanation.

An identifier is a text string contained within an HTML element. It could be a label identifying an input area, or a piece of text present on the page. What makes the identifier local is the distance between the selected element and the one containing the identification string. One can think of the HTML structure as a tree and, in this case, the distance between two elements is the tree distance between them in the HTML. For example, given the code segment in Figure 4, the input element with an id value of name has two local identifiers. The string “Name:” with a distance of 2 and the string “Age:” with a distance of 4. The path from the selected input to the first string requires two hops, one from the input to the parent div element and the second from the parent element to the div element containing the string “Name:”. In the MashDesk environment an identifier is considered local if its tree-distance is less than or equal to 15.

Once a widget is selected and all of the required information is gathered, MashDesk can track the presence of that widget throughout an evolution of an application. When a new version of the application is loaded, MashDesk performs the following steps in its attempts to rediscover the selected widgets.

First, it attempts to use the stored object’s XPath query. If the query returns an object with the same properties as the stored object, it is labeled and treated as the selected object. If the XPath query does not return an object, or the returned object does not match all of the stored properties, MashDesk proceeds to search the application for a widget that best matches the selected widget.

The search is limited to objects that have the same type as the selected widget. Once all of the candidate widgets are found, MashDesk discovers all of the local identifiers for each widget. Given those sets, they are compared to the original and a similarity score for each widget is calculated.

To calculate the similarity score between two widgets, their local identifiers are compared. Two identifiers match if their strings match. The score is calculated by summing the absolute value of the differences between distances of matching identifiers. Because locality is capped at a distance of 15, the difference cannot be greater than 14. Therefore, any identifiers that do not match, either from the candidate’s set or the original’s set, contribute a value of 15 to the final score. Identifiers with the lowest score are kept as candidate matches. Widgets with no matching identifiers are dropped from consideration. The following is the formula to calculate the similarity score, assuming that at least one match between the two sets of local identifiers:

$$S = 15(|I_1| - |D|) + 15(|I_2| - |D|) + \sum_{i=0}^{D_i} D_i$$

where $I_1$ is the set of all local identifiers associated with widget 1, $I_2$ is the set of all local identifiers associated with widget 2, and $D$ is the set containing the absolute value of the differences between the tree-distances of matching identifiers.

If, after the identifier comparison step, only a single candidate widget remains, it is presented to the mashup developer as a replacement for the selected widget. If the mashup developer confirms the selection, that widget replaces the original in the mashup. However, if more than one widget is selected, the algorithm performs an additional step in its attempt to narrow the search to a single result.

In the case of multiple widget candidates with identical similarity scores, MashDesk selects the best candidate by calculating the difference between each of candidate widget’s XPath queries and the original widget’s XPath query using a standard differentiation technique [13]. The widget whose query is closest to the original is chosen as the candidate replacement. If multiple candidate widgets have the same lowest difference score, they are all presented to the mashup developer as potential replacements.

To evaluate this algorithm we developed an experiment using the Internet Archives Wayback Machine [11]. The Wayback Machine has been archiving websites since the mid 1990’s and has over 150 billion web pages stored in its database. We downloaded several different versions of some of the most popular websites and attempted to track UI widgets as those pages evolved. The results are presented in Figure 5.

For each of the sites in the experiment, all of the pages made available by the Wayback Machine’s website were downloaded. It is worth noting that not all of the downloaded pages were useful. Several pages displayed an error message, instead of the website, and many of the non-error pages were
Success

(changes)

ERRORS

591/591 1/0 0/0 25/19
0/0 2/0 0/0 8/1
0/0 15/10 21/15 12/12
99.66% 98.41% 98.99% 96.82%
66.67% 86.15% 95.74% 75.95%
2 4 2 8

Fig. 5. Table summarizing the results of the tracking experiments.

The number of times each result was selected is shown in the first five rows in Figure 5. The following quantity is the number of those results that were correct. For example, the value “6/4” in the Match-1 row for the Amazon experiment means that the Match-1 result came up 6 times during the experiment. Out of those 6 times, 4 times the Match-1 result correctly pointed to the widget being tracked.

If a website had undergone significant changes and the widgets were not tracked, the No Match result was counted as the correct response. A significant change involves a major redesign of the website. For example, in the case of the MapQuest experiment, the site went through several drastic redesigns. At different iterations inputs were added and removed. At one point, the site moved away from a layout based on tables, to a more modern, hierarchical structure. In fact, mapquest.com did not have any input fields to track in the first few iterations of the site, forcing us to track links to different parts of the site as features.

Two version of the success rate are reported in Figure 5. The first is the percentage of the widgets successfully identified during the entire experiment, including the pages that were not altered. The second is the percentage of the widgets successfully identified only in the cases where the original XPath query failed to identify a widget, meaning pages that had undergone a change.

When a widget was not tracked via the original XPath query or a tracking mistake was made, the experimenter reselected the correct widgets to track, just as the mashup developer would select replacement widgets. During most experiments, 3 UI widgets were selected to be tracked. The only exception is that during some of the CNN experiments only two widgets were tracked. It is worth noting that the results in first six rows of Figure 5 are provided at the widget granularity. The final row, labeled ERRORS, is at a page granularity, representing the number of pages in which at least one widget was mistakenly identified.

These results represent several years worth of data. The Amazon experiment includes data from 2003 until 2008, the CNN experiment includes data from 1997 to 2009, the Google experiment includes data from 2000 until 2008, and the MapQuest experiment includes data from 1997 to 2009.

Each experiment involved 200 pages and given that amount of data it is encouraging that the tracking algorithm failed for only 16 out of 800 pages. Amazon and Google provided the most stable pages. While MapQuest and CNN went through several major revisions.

During the Amazon experiment, the category select widget, the search input widget, and the search button widget were tracked. These were located at the bottom of the page, and even though the main page was altered several times, those widgets were always present. The only mistakes occurred early in the experiment when the HTML element that contained all of the content was reworked. In both cases the search button at the bottom of the page was mistaken for the search button at the top of the page.

The Google front page was more volatile than the Amazon page. At one point, it contained tabs and links to various search options. During the Google experiment, the search input widget, the search button, and the I’m feeling lucky button were tracked. Notice that the Google experiment led to several Match-1 results. This is due to the fact that Google uses hidden input fields that were not always present on the archived page. These fields shared the same parent as the widgets being tracked, therefore changing the XPath query required to access the tracked widgets. This and the CNN experiments were also the only ones in which the algorithm incorrectly detected a change in the website when no visible difference was observed in the UI.

During the CNN experiment we attempted to track links to specific news categories. However, there was a period during which every aspect of the front page changed daily. The only discernible constants to track were the links to the European and Asian editions. During the experiment, the tracking algorithm was able to successfully track categories
through a major update. At one point, the categories layout changed from a list on the left of the page to a list at the top. However, because neither the categories nor their local properties had changed, MashDesk was able to track them through that redesign.

The MapQuest experiment contained the most volatile page. In fact, the form allowing users to enter addresses did not appear on the page until June, 2000. It was the 26th page in the experiment. The page used in the experiment went through 13 major redesigns. Out of those, 4 were detected successfully and 5 were not. In the remaining 4 cases, the widgets remained tracked despite the change. In 3 out of the 5 unsuccessful redesign detection cases, the algorithm selected several incorrect options for each of the widgets being tracked.

Overall this experiment proved to be a success. In all cases, when a widget was tracked via the original XPath query it was never mistaken. Using additional, local identifiers, proved useful in tracking widgets through minor, visually unnoticeable, changes. The algorithm was also able to successfully track UI widgets through several major redesigns. The only issue with the algorithm is that it sometimes had difficulty detecting major redesigns. However, arguably, that is not a major flaw. When a major redesign is not detected, it will still be observed because the mashup using the redesigned application will begin to exhibit faulty behavior.

V. DEMONSTRATION

So far all of the maintenance features of the MashDesk environment have been described separately. The following demonstration illustrates how they are used together to update a mashup when the UI of one of its applications evolves. In this case the evolution will be simulated by replacing one of the applications with another, similar, application.

The mashup presented here combines a geocaching-search web application with an application name Mapper. Geocaching is an activity where participants use GPS to hide and seek containers, called “geocaches”, throughout the world [19]. Mapper accepts latitude and longitude coordinates as input and plots them on a GoogleMaps map. The goal of this demonstration is to illustrate the process of updating a mashup when the UI of one of the applications it is based on changes.

Mapper is an application native to the MashDesk environment. The two geocaching applications used in the demonstration are not. These applications are wrappers around the OpenCaching.com and GeoCaching.com websites. Neither one had inputs or outputs defined prior to the demonstration. To integrate them into the mashup, the developer has to go through the process described in Section III and identify the data to extract and actions that trigger an export event.

Neither of the caching applications provide the latitude and longitude coordinates as a part of the search results page, the user has to click on each link to get that information. Also, the coordinates are accessed in slightly different ways for each website. When the user clicks on a search result in OpenCaching.com, that result is fetched and displayed directly on the search results page. When the user clicks on a search result in GeoCaching.com, that result loads a new page, replacing the search results page.

The demonstration begins with an existing mashup application that combines OpenCaching.com and Mapper. Figure 6a illustrates that application. It has a search input field and a search button taken from OpenCaching.com as well as a GoogleMap’s map taken from the Mapper application. The next step is to set GeoCaching.com as the next version of OpenCaching.com. This step is performed internally in the MashDesk environment.

At the next instantiation of the mashup, the MashDesk environment warns the mashup developer that the OpenCaching.com application has changed and that the mashup needs to evolve as well. MashDesk was able to detect this change because the tracking algorithm presented in Section IV was not able to access the predefined UI widgets with the saved XPath queries.

When the mashup developer enters edit mode, as illustrated in Figure 6b, MashDesk breaks the mashup into separate applications and highlights the missing UI widgets. The developer is asked to identify missing UI widgets in the GeoCaching.com application. He first drags the search input widget from the GeoCaching.com page into the red rectangle that represents the location of the search input box. Next he drags the search button from the GeoCaching.com page into the red rectangle that represents the location of the search button. Once all of the elements have been replaced, the developer is asked to execute a test case.

The execution of a test case begins with the developer entering a search string into the search input widget and hitting the search button. MashDesk attempts to execute the test cases using the program developed for OpenCaching.com. Failing to do so, MashDesk requests the developer to again demonstrate how the website should be used. Because this will serve as a replacement for the previous application, the MashDesk environment does not need to redefine the data that the GeoCaching.com application has to export. Instead it presents the developer with the search results page and an export table with labeled headers. MashDesk continues by asking the developer to demonstrate how to fill it. This is illustrated in Figure 6c.

The developer begins his demonstration by clicking on the first search result’s link, which loads the corresponding page. Next, the developer enters “data extraction” mode and demonstrates how to extract and clean the latitude and longitude coordinate. Once the extraction process is demonstrated for the first search result, the developer clicks on the back button. At this point, the export table contains a single data row.

Once the developer returns to the search results page, he clicks on the second search result’s link. Doing so prompts MashDesk to generalize the XPath query referring to the first search result’s link into an XPath query that captures all of the search result links on the page. This is illustrated to the developer by highlighting all of the search result links on the page by surrounding them with a yellow background. The developer is asked to confirm this selection. Doing so causes
Fig. 6. Illustrating a maintenance operation that replaces OpenCaching.com with GeoCaching.com. (a) The original application. (b) MashDesk identifies missing widgets by drawing red boxes in their place. The green arrows illustrate the developer’s actions to drag replacement widgets into correct spaces. (c) The results of a test search entered by the developer. MashDesk has created an extraction table to store data as the developer demonstrate how to extract it. (d) Once MashDesk gathers enough information, it proceeds to populate the extraction table using the process demonstrated by the developer. (e) The final mashup

MashDesk to begin visiting these links, extracting the latitude and longitude coordinates from each page, and adding them to the export table. This is illustrated in Figure 6d. Notice that the results in the table and the search results are not in the same order. This is due to the fact that MashDesk uses asynchronous HTTP requests to follow each search result’s link on the page, thus causing them not to be returned in the order they were requested. Once the table is filled, the developer is asked to confirm the results. Once the results are confirmed and the demonstration over, the developer is presented with the new mashup that can be stored and used in the same manner as the previous one. The new mashup is illustrated in Figure 6e.

This demonstration illustrates how mashup maintenance operations are performed inside the MashDesk mashup environment. The developer is only asked to demonstrate the features that have changed. If, for example, the process of extracting data from GeoCaching.com’s search results was similar enough to the original process from OpenCaching.com, the developer would not have been asked to repeat it.

VI. CONCLUSIONS AND FUTURE WORK

This paper deals with maintenance issues in mashups that have been composed by integrating applications at the UI layer. Employing such an integration means that UI widgets will be used to access application features, as opposed to an application’s API. Integrating applications at the UI layer has several advantages. First, the developer is not limited to...
using the provided API. Second, the developer can reuse UI experiences he is already familiar with, instead of creating new ones. And finally, UI layer integration empowers non-expert mashup developers to create mashups by only dealing with constructs they are familiar with.

While there are clear benefits to such mashups, the maintenance challenge in using the UI in integration stems from the fact that not only does the UI change in significant ways, but even small alterations can cause problems for the mashup. This was demonstrated in the UI widget tracking experiment in Section IV. During that test, many small and large changes to the UI layer were observed.

To address the issue of UI evolution, the MashDesk environment employs an algorithm to track UI changes as an application evolves. Section IV explains that algorithm and demonstrates its effectiveness in tracking the evolution of four websites over a period of several years. In addition to tracking UI changes, Section III describes how mashups are created in a modular fashion. This modularity enables developers to update their mashups as applications evolve. The process of performing such an update is presented in the demonstration in Section V.

While many of the challenges are addressed in this paper, there is still work left to do. One area in which we wish to focus is in increasing automation. Currently, when a change is detected, the developer is asked to confirm a potential solution. If test cases based on the initial mashup composition can be stored, a regression test suite can be applied to all potential solutions and the most suitable one can be picked automatically.

Another area of potential automation is in demonstrating the actions required to exercises application features. In the demonstration described in Section V, when OpenCaching.com was replaced with GeoCaching.com, the developer had to repeat the demonstration showing MashDesk how to extract the data and formulate the export table. One may be able to automate that process. As the process starts, much of the information describing the export table is already known, therefore it is a matter of discovering the correct links to press and the correct data fields to process. Given the number of possible combinations this seems to be a candidate problem to be solved via genetic programming.

REFERENCES