2012

On the Benefits of Providing Versioning Support for End-Users: an Empirical Study

Sandeep Kaur Kuttal
University of Nebraska–Lincoln, skuttal@cse.unl.edu

Anita Sarma
University of Nebraska-Lincoln, asarma@cse.unl.edu

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

On the Benefits of Providing Versioning Support for End-Users: an Empirical Study

Sandeep Kaur Kuttal
Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
skuttal@cse.unl.edu

Anita Sarma
Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
asarma@cse.unl.edu

Gregg Rothermel
Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
grother@cse.unl.edu

September 13, 2012

Abstract

End users with little formal programming background are creating software in many different forms, including spreadsheets, web macros, and web mashups. Web mashups are particularly popular because they are relatively easy to create, and because many programming environments that support their creation are available. These programming environments, however, provide no support for tracking versions or provenance of mashups. We believe that versioning support can help end users create, understand, and debug mashups. To investigate this belief, we have added versioning support to a popular wire-oriented mashup environment, Yahoo! Pipes. Our enhanced environment, which we call "Pipes Plumber", automatically retains versions of pipes, and provides an interface with which pipes programmers can browse histories of pipes and retrieve specific versions. We have conducted two studies of this environment: an initial exploratory study and a larger controlled experiment. Our results support our beliefs that versioning will help pipes programmers create and debug mashups. Subsequent qualitative results provide further insights into the barriers faced by pipes programmers, the support for reuse provided by our approach, and the support for debugging provided, as well as insights into differences between users who have different levels of programming experience.

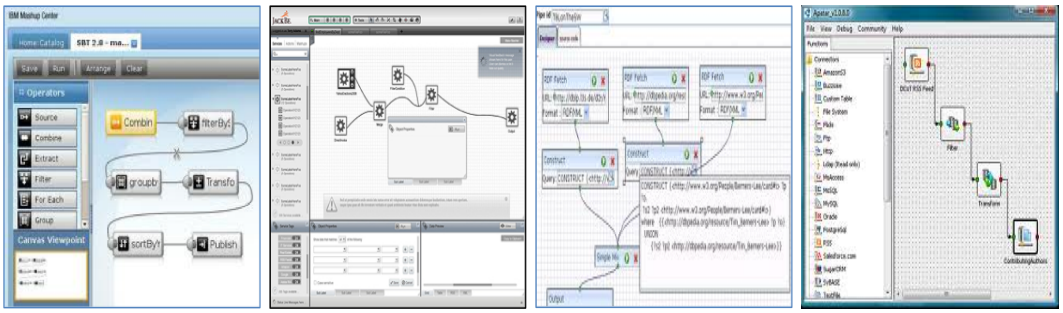
1 Introduction

Web mashups are applications that combine data, functionality and presentation from two or more sources to create new services. Mashups are a crucial emerging technology because of their role in three primary trends; namely, Web 2.0, situational software applications [20], and end-user development [48]. Mashups are popular with both end-user and professional programmers.

Users programming mashups do not need to write scripts or programs; they can instead take advantage of visually oriented programming environments. Examples of these environments include IBM mashup maker[1],

[1]IBM Mashup Maker: http://www.ibm.com/software/info/mashup-center/.

Department of Computer Science & Engineering, University of Nebraska-Lincoln, Technical Report, TR-UNL-CSE-2012-0008



(a) IBM mashup maker (b) Jackbe (c) Deri Pipes (d) Apatar

Figure 1: Wire-oriented mashup environments.

JackBe[2], Deri pipes[3], Apatar[4], xfruit[5], and Yahoo! Pipes[6] (see Figure 1 for screenshots of a subset of these). Among these programming environments, Yahoo! Pipes is one of the most popular. Yahoo! Pipes is a commercial mashup programming environment that allows users to aggregate and "mashup" content from around the web. Since its launch in February 2007, over 90,000 developers have created individual pipes on the Yahoo! pipes platform, and pipes are executed over 5,000,000 times each day [22].

Mashup programming environments provide central repositories to end users where they can store their mashups. However, current environments do not provide facilities by which users can keep track of the versions or provenance of the mashups they create. In the professional software engineering community, versioning is widely acknowledged as beneficial for activities such as code understanding, change traceability, debugging and maintenance [45]. Versioning systems provide an environment within which previous states of resources can be easily retrieved. Versioning allows engineers to browse through past and alternative versions of a resource and determine how these versions differ from one another. Below we provide examples of cases in which versioning support can be helpful for mashup programmers, and especially for end users.

In a domain study of experts, Jones and Scaffidi [23] observed that there is a need for version control to enhance the maintainability and reuse of visual domain specific languages. It has also been observed that in the Yahoo! Pipes repository, 43% of pipes submitted are just variations of previously submitted pipes, indicating that authors may be using the public repository as a private repository to store versions of their pipes containing incremental changes [41].

Backtracking and investigating alternative scenarios are an integral part of exploratory data analysis. Yet most mashup programming environments, including Yahoo! Pipes, cannot represent alternative exploration paths as branching histories, forcing users to rely on memory to compare scenarios. In a study of end-user mashup programmers it was observed that all participants backtracked multiple times while creating

[2]Jackbe: http://www.jackbe.com/.
[3]Deri Pipes: http://pipes.deri.org/.
[4]Apatar: http://apatar.com/.
[5]xfruits
[6]Yahoo! Pipes: http://pipes.yahoo.com/pipes/.

mashups, either because they had alternative ideas or wished to return to some previous successful state [3]. In a later paper further analyzing data from that study, the authors also observed that participants spent 76.3% of their time in debugging alone while developing mashups [2].

Motivated by the studies just described, we are investigating the use of versioning in mashup programming environments. We conjecture that the addition of versioning support to these programming environments will help in several ways, including (1) helping mashup programmers create and reuse mashups, (2) helping mashup programmers understand the evolution of mashups (3) helping mashup programmers backtrack to successful states and explore alternative ideas and (4) helping mashup programmers debug mashups.

To investigate these conjectures, we have added versioning support to a wire oriented mashup environment. We chose Yahoo! Pipes as a platform for this effort. The primary reasons for selection of Yahoo! Pipes include its popularity, the fact that it is available for free, and the fact that its data can be captured and manipulated by external systems. Our extension to Yahoo! Pipes, which we call "Pipes Plumber", keeps version histories for mashups automatically. This allows users to utilize the advantages of versioning without needing to be aware of the underlying functions known to professional programmers such as check-in, check-out, and so forth. To explore the potential cost and benefits of our versioning support, we have conducted two user studies. Our first study [31] involved a version of the Yahoo! Pipes environment augmented with basic configuration management functionality and a simple interface. An exploratory, think-aloud study of nine participants (primarily computer science students) showed that the versioning support helped them create mashups more efficiently.

Subsequently, for a second study, we extended our versioning support by providing additional user interface assistance intended to help users debug faulty mashups. We then conducted a controlled experiment involving participants who do, and who do not, have formal programming training and experience, studying questions related to the creation and debugging of pipes [30]. Our experiment results confirm that both groups of users can create pipes more effectively and efficiently with the aid of versioning support, and they can also debug pipes more effectively.

This article presents the quantitative aspects of both of the foregoing studies, and then augments this with an extensive qualitative analysis of our data. This qualitative analysis provide further insights into the barriers faced by pipes users, the support for reuse and debugging provided by our approach. It also provides insights into differences between end users who have, and who do not have, more formal programming experience.

The remainder of this article is organized as follows. Section 2 provides background information. Section 3 describes our Pipes Plumber extension to Yahoo! Pipes and its interface. Section 4 provides details on the setup and results observed in both studies. Section 5 provides our qualitative analysis of the two studies. Section 6 discusses related work. Section 7 concludes and discusses future work.

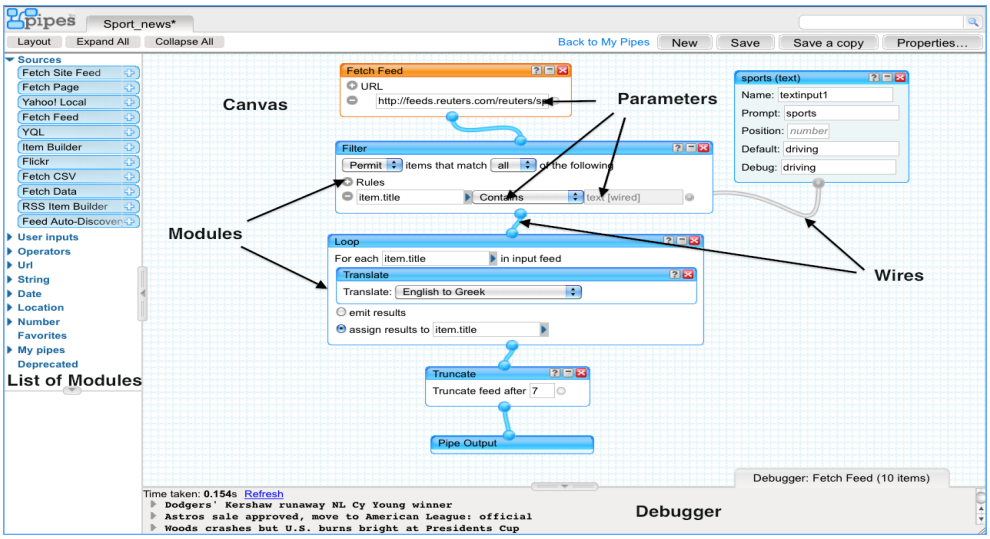


Figure 2: Yahoo! Pipes interface

2 Yahoo! Pipes

Yahoo! Pipes[7] is arguably the most popular mashup creation environment, and is being used both by professional and end-user programmers. Yahoo! Pipes is a web-based visual programming environment introduced by Yahoo! in 2007 with the intent of enabling the users to "rewire the web". As a visual programming environment, Yahoo! Pipes is well suited to representing the solutions to dataflow based processing problems [22]. Yahoo! Pipes "programs" combine simple commands together such that the output of one acts as the input for the other. The Yahoo! Pipes engine also facilitates the wiring of modules together and the transfer of data between them.

Figure 2 shows the interface of the Yahoo! Pipes environment and the various components of that interface. The pipe displayed in the figure takes an RSS feed from Reuter's News as input and then filters the news based on input parameters supplied by the user (by default, sports). It then converts the titles of all the news feeds from English to Greek, and displays the first seven results. The Yahoo! Pipes environment consists of three major components: canvas, library (list of modules) and debugger. The canvas is the central area where a user creates a pipe. The library is located to the left of the pipe editor and consists of various modules that are categorized according to functionality. Users drag modules from the library and place them on the canvas, then connect them to other modules as needed. The debugger helps users check the runtime output of specific modules including the final output module (in Figure 2 the debugger window displays the output from the module `Fetch Feed`).

Pipes programmers create their pipes using the visual interface on the client side. When they save a pipe, it is encoded in JSON format and sent to the Yahoo! Pipes server, which is where all pipes are saved

[7]Yahoo! Pipes: http://pipes.yahoo.com/pipes/.

Table 1: Comparison of Features of Existing CM Systems to Features Available in Pipes Plumber

Features	CM Systems	Our System
Versioning unit	File level	Pipe level
Differencing	Text level	Module level
Deltas	Add, delete, modify	Add and delete
Versions created	On commit	On save/cloning of pipe
Browsing	Undo or select version	Undo, redo or select from list view
Snapshots	Baseline	Run
Merge	Implemented	Future work

and executed. Programmers may also "clone" pipes that are available in the repository, in order to reuse them in new contexts.

Input to a pipe can be HTML, XML, JSON, RDF, RSS feeds, as well as many other formats. Similarly, pipe output can be RSS, JSON, KML and other formats. The input and output between modules are primarily RSS feed items. RSS feeds consist of *item* parameters and descriptions. The Yahoo! Pipes modules provide manipulation actions that can be executed on these RSS feed parameters. In addition to items, Yahoo! Pipes also allows datatypes like *url, location, text, number*, and *date-time* to be defined by users.

3 Pipes Plumber

We now discuss the approach we use to provide versioning capabilities for Yahoo! Pipes, including the features we provide, our system architecture, and the interface used.

3.1 Versioning Features

In the world of professional software development, versioning support is provided by *configuration management* (CM) systems. There are several versioning features available to professional developers in such systems. To achieve versioning support in Yahoo! Pipes we considered each of these features and created appropriate analogous features. Table 1 summarizes the results, which we next elaborate on.

- In most CM systems the *versioning unit* is a file. There is no concept of files in Yahoo! Pipes; the smallest executable unit is a pipe which is checked-in (saved) or checked-out (viewed) by pipes programmers. Thus, we chose to use individual pipes as our versioning unit.

- In CM systems the *differences between two versions* of files are calculated at the text level. Pipes, in contrast, consist of modules and wires. Pipes programmers "program" by adding or deleting modules. Thus, we chose to calculate the differences between pipes at the module level.

- In CM systems, because differencing is done at the text level, *deltas* for a file can be lines of code added, deleted or modified. As mentioned before, we facilitate differencing of pipes at the module level. This

leads to a clear choice of treating modules (added or deleted) as deltas.

- In CM systems a *version* is created by a professional developer on each commit, which he or she deliberately creates on major changes. Our objective is to integrate versioning into the Yahoo! Pipes environment, while requiring little effort on the part of end users to learn about versioning features. End users also purposefully save their pipes to incorporate changes. Thus, we decided to automatically create versions when users save or clone their pipes, so that they do not need extra effort to create versions.

- CM systems allow developers to *browse different versions* of a file by using commands to select some specific version. Such commands may increase the cognitive load for end users; thus we decided to enable browsing using customized widgets. With these widgets, users can perform various activities like undo, selection of a version (as in CM systems) or redo (an additional feature).

- Typically, CM systems allow developers to take *snapshots* of the development tree, by marking significant versions as baselines for retrieval and deployment. In the Yahoo! Pipes environment, a user who believes that his or her pipe is complete will run the pipe to check the results. We use this to facilitate the automatic creation of baselines. When a user runs a pipe, we tag that version as a baseline. In addition to baselines, in our environment, we find other forms of *snapshots* of systems useful, such as the number of results returned and the success or failure of the run. We also tag the user's tested versions of pipes for debugging.

- CM systems provide *merge* facilities to let programmers merge versions; this facilitates collaboration among different professionals. We do not yet support merging, but we plan to do so in the future.

3.2 System Architecture

To implement versioning support, rather than create an add-on for a specific browser we decided to create a proxy wrapper. This proxy wrapper intercepts the JSON code (pipe) that is transmitted between the user's client running in their browser and the Yahoo! Pipes server. This allows "Pipes Plumber" to be operational for most web browsers (including Internet Explorer, Chrome, Firefox and Safari).

Figure 3 presents our system architecture. We use a proxy server (Squid 3.1.4[8]) to manage communications between the client (web browser) and the Yahoo! Pipes server. Using the Internet Content Adaptation Protocol (ICAP[9]), a proxy wrapper intercepts the request and response messages exchanged between a client and the Yahoo! Pipes server. When the user requests the user interface (UI) of Yahoo! Pipes, the response related to the UI is redirected to the proxy wrapper. The proxy wrapper modifies the response messages

[8]Squid: http://www.squid-cache.org/Versions/.
[9]ICAP: http://www.icap-forum.org/.

of the UI by inserting "widgets" related to versioning and debugging into the UI of Yahoo! Pipes before delivering the message to the client.

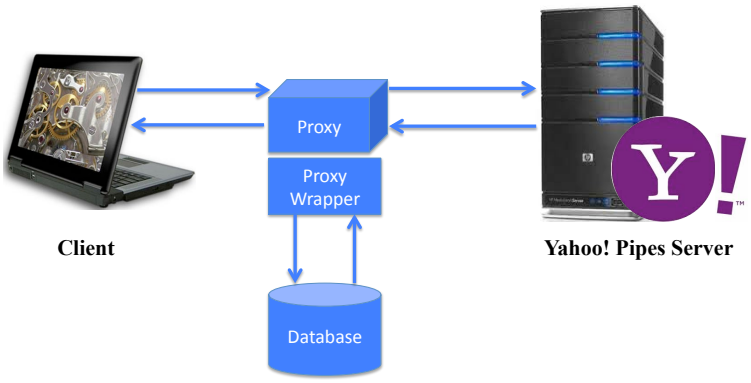


Figure 3: Versioning support architecture

Our proxy wrapper intercepts user events (e.g., save or run) and message contents to create and store versions in a MySQL database, which serves as the versioning repository. When a user saves a pipe, a new version of the pipe is created in the repository. Versions are created in chronological order (e.g., V_1, V_2, V_3,....,V_n) in the sequence of saves. Each version is composed of the set of modules added to the canvas by the user. Each version also keeps track of its parent. When the user requests a particular version of a pipe from the Pipes Plumber interface, the requested version is selected from the repository. Hence, each version of a pipe can be viewed, edited, or run using the Pipes Plumber interface.

To illustrate the process, consider a user-based scenario. Suppose Sally wants to create a pipe using the Yahoo! Pipes environment, but she also wants the benefits of versioning. She connects with our proxy server and sends a request for the Yahoo! Pipes interface to the Yahoo! Pipes server. When the response comes back from the server it comes through the proxy server where the proxy wrapper adds our widgets to the Yahoo! Pipes interface. Hence, Sally views the Pipes Plumber interface and is ready to use our versioning support in the Yahoo! Pipes environment.

Suppose Sally begins to create a pipe to search for movies played in theaters near a specific location. Once her task is done she saves the pipe. As soon as she saves her pipe a save message (POST) is sent from the client machine to the Yahoo! Pipes server. Our proxy wrapper intercepts this message and saves the JSON contents from the message body as a version in the central repository. Hence, the first save of Sally's pipe creates version V_1 in the central repository. Each time Sally adds additional functionalities to the pipe (such as checking reviews or posters of a movie), and later saves or clones her pipe, corresponding versions are saved in the central repository, in chronological order. (Note: the Yahoo! Pipes repository retains only the latest contents of the pipe; earlier versions are all saved by our system.)

Given the foregoing, if Sally wishes to retrieve a previous version V_2 of a pipe, she can achieve this using

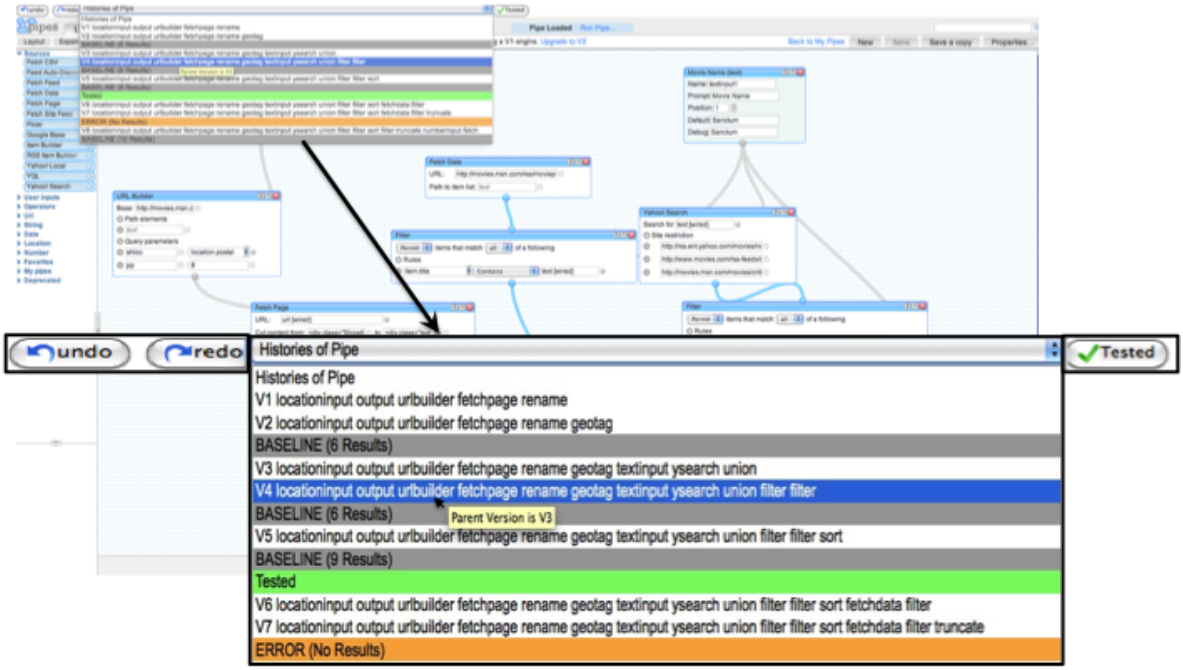


Figure 4: Pipes Plumber interface

the Pipes Plumber interface. She selects version V_2 and this version is drawn from the local repository and sent to the Yahoo! Pipes server, where she can access or execute it.

3.3 Interface

The user interface of Pipes Plumber adds four widgets to the Yahoo! Pipes client interface (see Figure 4). These widgets are 1) "Undo", 2) "Redo", 3) "Tested", and 4) "History of Pipes". The first two widgets,"Undo" and "Redo", are buttons that allow users to browse between consecutive versions of a pipe. "Undo" renders the previous version, while "Redo" renders the next version. The third widget, "Tested", is a button that allows users to tag their pipes as correct.

The fourth widget, a "History of Pipes", displays the modules added or removed per pipe, per version, so that users can view the differences between versions. The "History of Pipes" widget, implemented as a drop-down list, allows a user to select a desired version from the list of available versions of the pipe. At present, the history of pipes list presents versions in chronological order, and this may mask cases in which a version is actually derived from some much older version (e.g., V8 from V3). While we might be able to address this by employing graphical representations of the versioning history, initially we have attempted to address this by displaying further information on version provencance in tooltip boxes that appear when the pipe programmer hovers over a given version (Figure 4). To better describe the local history of mashups in terms of states and actions, we also provide textual descriptions of the contents of the versions.

We conjecture that versions created after a user tests a particular version can more easily be debugged

for errors, because users can see when modules are added to the tested version of the pipe, which may be potential sources of error. The history of pipes list makes use of colors to help users distinguish successful pipe runs, unsuccessful pipe runs, and tested versions of pipes. Versions highlighted in grey (in Figure 4, the menu item labeled "BASELINE") are versions that were successfully executed (the pipe returns non-null results). The numbers of results returned during that particular run are also shown. Orange (item labeled "ERROR (No Results)") signifies that a pipe execution returned no results, a probable indication of an error. Green (items labeled "Tested") indicates that the pipe programmer has confidence that a version is operating correctly, a status that can be applied by using the "Tested" button in our interface.

4 Empirical Studies

We have conducted two user studies. The first, an exploratory study, involved nine participants and focused on pipe creation and understanding tasks in the absence and presence of versioning support. The second study, a controlled experiment, involved 24 participants, and focused on pipe creation and debugging tasks in the absence and presence of versioning support.

4.1 Study I

Our first study addresses the following research questions:

- **RQ1:** How does versioning allow mashup programmers to effectively and efficiently perform tasks related to mashup creation?
- **RQ2:** How does versioning help mashup programmers understand complex third-party mashups?

4.1.1 Pipes Plumber Platform

In this study, basic versioning facilities in the form of "undo", "redo" and "History of Pipes" widgets were added to the interface as described in Section 3.

4.1.2 Participants

To recruit participants we sent an email to a departmental mailing list. As an incentive, participants were included in a raffle for a $25 prize. Nine students responded to our advertisement. All students were male, with seven from computer science or computer engineering and two from other departments. Four of the students were undergraduates and five were graduates. None had prior experience with Yahoo! Pipes, but all had some programming knowledge (78% knew multiple programming languages).

4.1.3 Independent and Dependent Variables

The independent variable in this study involved the presence or absence of versioning information, with the former provided via our enhanced Yahoo! Pipes environment. Dependent variables used for the study were

(1) the correctness of pipes following creation activities, and (2) the time required to create pipes. These variables allow us to measure the effectiveness and efficiency of the participants at performing their tasks.

4.1.4 Study Setup and Design

The study used a single factor, within-subjects design (a design in which the independent variable is tested with each participant [46]). We opted for a within-subjects design for three reasons. First, we wished to minimize the effects of individual differences among participants, which can be reduced through a within-subjects design. Second, a within-subjects design allows us to gather more data using a smaller sample size. Finally, since each participant in our within-subjects study gained experience performing tasks using the environment with and without our versioning support, they were better positioned to provide feedback about the usefulness and usability of our versioning support than would participants in a between-subjects study.

We used think-aloud protocol in our study, asking participants to vocalize their thought processes and feelings as they performed their tasks [32]. We used this protocol because a primary objective of this exploratory study was to gain insights into the users' thought processes, and into the barriers and problems that a user faces when using Yahoo! Pipes and our extension. This approach required us to administer the study to participants on an individual basis with an observer; in this case, the first author.

We performed the study in the Usability Lab of the Computer Science Department at the University of Nebraska-Lincoln. At the beginning of the study, participants were asked to complete a brief background questionnaire, which was followed by a tutorial of approximately ten minutes on Yahoo! Pipes and versioning in general. The tutorial also included a short video of a sample think-aloud study so that users could understand the process. After completion of the tutorial, we asked participants to create a small sample pipe to give them hands-on training in creating a pipe and familiarity with Yahoo! Pipes. We also provided them with a list of documentation links which were within the Yahoo! Pipes web site. Following these preliminaries, participants were asked to complete tasks for the study. Each participant completed a pair of tasks for RQ1 followed by a pair of tasks for RQ2. The first of each pair of tasks was a task without versioning support (control tasks), and the second was a task with versioning support (experimental tasks). We audio recorded each session and logged the users' on-screen interactions using a screen capture system (Morae[10]). The total time required for completion of the study per participant was approximately 1.5 hours, which included an average of 60 minutes for task completion.

After all tasks were completed, we administered an exit survey to obtain feedback and additional thoughts from participants. The survey consisted of both closed and open-ended questions about the tasks, the interface of the versioning extension, and the experimental process.

[10]Morae: http://www.techsmith.com/morae.asp.

4.1.5 Tasks

We designed two tasks to address our research questions: Task1 and Task 2. Since the study was within-subjects, we further subdivided each task into two categories: Control (C) and Experimental (E) tasks. Therefore, in total we defined four tasks with Task1.C and Task1.E addressing RQ1, and Task2.C and Task2.E addressing RQ2, respectively.

Task 1 required a participant to create a pipe as per a given set of requirements. Participants were first given a similar existing pipe (of ten modules) as a template that they needed to understand. Then they were asked to create a pipe reusing some parts of the first pipe and adding some functionality. The existing pipe for Task1.C allowed a Spanish-speaking person to search for reviews of any business within a geographical location and within a certain distance, such that all results returned contained the search term in the title and were sorted in alphabetical order. Once the participant understood the pipe, they were required to create a pipe that allowed a search for a review of any item around any area within a certain specified distance but required the addition of new functionality, namely, the ability to change the original title of the search results to a title of their choice.

Task1.E involved a different pipe of complexity similar to the one used in Task1.C. This pipe was designed for a news enthusiast who wanted to search Yahoo! News for a topic filtered to display only those news items with some keyword in the title. The results of the query were to be unique and contain at least two items in reverse order based on the date of publication. As in Task1.C, the participants were asked to understand the given pipe and then move on to the next step. In the second part of Task1.E, the participant was asked to create a pipe that allowed a French speaking user to search Yahoo! News for a search term while ensuring that search result titles were unique and translated into French. The pipe used in this task had an associated versioning history that was accessible to the user.

Task 2 required the participant to view a given pipe and answer a set of multiple-choice questions about the pipe and its functionality. The pipes used in Task 2 (Task2.C, Task2.E) were larger and more complex than those used in Task 1. Task2.C involved a pipe of 47 modules that displayed a list of unique, "mashed up" contents from five different sites specified by the user. The user could also limit the number of results that were displayed and sort results in descending order of date. Task2.E involved a pipe of 50 modules and was actually a generic filter to merge feeds from different sources, remove duplicates and ensure unique items in those feeds. Further, a user could specify four different feeds, truncate or limit the maximum number of resulting items per feed, and select a maximum number of items that could be displayed. A versioning history for this pipe was available so that participants could investigate how the pipe was built from the ground up to better understand the different functionalities provided by the modules in the pipe.

Measures

To evaluate whether versioning helped pipe programmers in their tasks, we measured the time it took participants to complete each task and the quality of the resulting pipes. The duration of the time needed

by the participants to finish a task was measured in minutes. The quality of pipes was determined by grading the pipes created by each participant to create a correctness score ranging from 0 to 100, where high scores indicate better performance. To reduce possible bias in the grading scheme, the first author and a graduate student not involved in the work created a grading scheme for the pipes. They used this to grade pipes individually, and then they conferred and came to consensus on the grading results. The following paragraphs outline the grading scheme used.

On *Task1*, participants could either reuse the modules from the pipe provided in the preliminary step, or create the pipe by re-implementing the modules. In either case, they were already aware of the functionality provided in the given modules. We thus assigned 40 points to the correct use of the modules or the functionality they provided. The remaining 60 points were awarded if the participant could correctly create the additional modules needed to complete the task. Participants were penalized 10 points if they used incorrect logic (i.e., incorrect wiring between modules) or incorrect URLs. Participants were also penalized five points for each erroneous module added. Finally, for each module that remained in the pipe but had no meaningful impact on the output, two points were deducted.

On *Task2*, we measured only correctness. The task consisted of five quiz questions. Each question was worth 20 points; hence a correct answer resulted in assignment of 20 points and incorrect resulted in assignment of zero points.

4.1.6 Threats to Validity

External Validity. The primary threat to external validity in our study is that all our participants were male and seven out of nine were Computer Science or Computer Engineering majors. Computer Science and Engineering majors are not necessarily representative of other classes of end users who use Yahoo! Pipes. Still, they do represent one class of users and a class that could conveniently be studied initially. A second threat to external validity is that our study considered only two tasks that built on only two types of pipes. Additional studies are needed to examine other tasks and other types of pipes. A third threat arises from the fact that the participants were asked to use pipes that were provided, rather than pipes which they had created for themselves. While the reuse context is common and important, prior familiarity with pipes could lead to different results.

Internal Validity. The primary threat to internal validity for this particular study relates to our choice of a within-subjects design. This study design helped to minimize the effects of individual differences and the use of a small pool of participants, but it might have led to learning effects as the participants moved from initial to later tasks. Another threat is that our tasks for Control and Experimental groups were not counter-balanced; this could have led to a bias in the performance of the experimental tasks. (Note that we counter-balanced the order of tasks as well as the order of treatments in our second study). A further threat could occur if the control and experimental tasks were not comparable in terms of complexity; however,

our data shows that participants required similar amounts of time to understand pipes in the control and experimental tasks, suggesting that the pipes were of similar complexity.

Construct Validity. Threats to construct validity include the possibility that the complexity of our pipes was not high enough to allow measurements of effects. We controlled for this by performing initial pilot studies on non-participants and using their feedback to adjust the pipes and the tasks.

4.1.7 Results

We address our research questions in turn.

Research Question 1

To address our first research question we frame two separate hypotheses, involving the time required to create pipes with and without versioning and the correctness of the pipes thus created. We consider each hypothesis (presenting the alternative hypotheses) in turn.

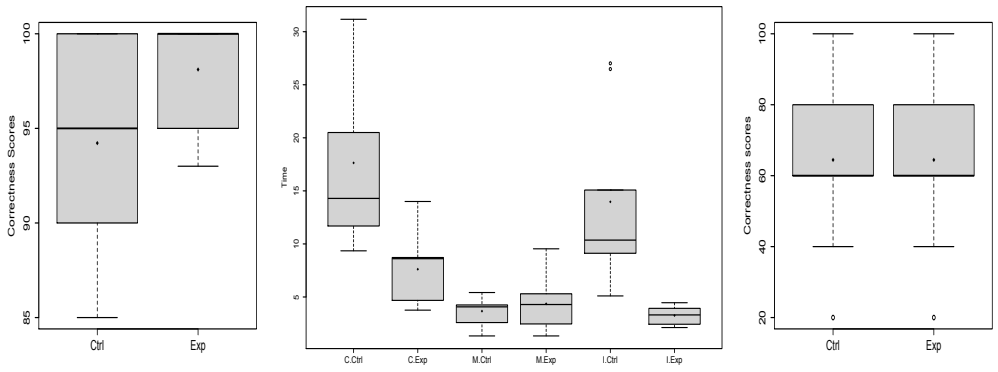
Ha1.1: Pipes created with versioning support are more correct than those created without versioning support.

As our participants attempted Task1.C and Task1.E, we allowed them to proceed to the next task when they believed that they had finished creating the pipe as per the requirements. Since there was no acceptance test, some of the pipes were not completely correct. Figure 5(a) presents the percentages of correct pipes created in the first task. Participants were largely successful in creating correct pipes in both the control and experimental tasks, with median correctness for the control (Ctrl) task being 95% and median correctness for the experimental (Exp) task being 100%. Participants in the experimental tasks did exhibit a slightly higher measure for correctness of pipes than when engaged in the control task. We used the non-parametric Wilcoxon Signed-rank test to compare the groups at α=0.05; giving us a W=0 (df=8) and a p-value of 0.045; hence, we conclude that the differences in correctness between the two groups are statistically significant, confirming hypothesis Ha1.1.

Ha1.2: Pipes created with versioning support require less time to create than those created without versioning support.

As participants performed tasks relevant to RQ1 (Task1.C and Task1.E), we examined the total time required for them to complete (C) their tasks, which included both the time required to understand/comprehend (M) the given sample pipe and the time required to implement (I) the required pipe. Figure 5(b) provides boxplots displaying the distribution of total times taken in these three categories.

We first consider overall creation time. The median time spent by all participants in Task1.C was 14.3 minutes (C.Ctrl) while the median time spent for Task1.E was 6.7 minutes (C.Exp). We used the non parametric Wilcoxon signed-rank test to test the difference in measurements between the two experiment alternatives on each sample. The test yielded W=43 (df=8) with p-value = 0.006 and confirmed the hypothesis, so we conclude that it took participants more time to complete Task1.C than to complete Task1.E.



(a) Correctness (Task1) (b) Times (Task1) (C = Complete, M = Comprehend, I = Implement) (c) Correctness (Task2)

Figure 5: Boxplots for Task1 and Task2: Dots signify means.

Given that creation time differed overall, we wished to determine what part(s) of the task (understanding/comprehension and implementation) contributed to that difference. Thus, we next consider the time required by participants to understand/comprehend the pipes they were given in Task1.C and Task1.E. The median time taken by the participants to understand/comprehend the pipe in the control task was 3.7 minutes (M.Ctrl), while in the experimental task the median was 2.9 minutes (M.Exp). We conducted the Wilcoxon Signed-rank test on the data, and the p-value in this case was 0.363 (with W = 15 and df=8); thus we cannot assert that there were any differences in understanding/comprehending the pipes during Task1.C and Task1.E.

We next turned to the implementation task. Figure 5(b) shows that for all participants, the time required to implement pipes was less in the experimental task than in the control task. The median time required to create pipes in the control task was 10.4 minutes (I.Ctrl), while the median time required to create pipes in the experimental task was 3.5 minutes (I.Exp). We again used a Wilcoxon signed-rank test on the data at at α=0.05; the test yielded W=45 (df=8) with p-value = 0.002. This confirms that the differences in times were statistically significant. It would seem, then, that the observed differences in overall pipe creation time stem from differences in implementation times.

Research Question 2

Task 2 was designed to help us address our second research question regarding the benefits of versioning in enabling the understanding of complex, third-party mashups. We investigate this question through the hypothesis:

Ha2: Versioning will help users learn about the mashups they encounter.

In addition to observing user behavior in Task 2, we administered a multiple choice quiz in which participants were asked to answer questions to help us assess the degree to which they understood the pipes.

Figure 5(c) provides boxplots displaying the distribution of correctness of participant's responses for Task2.C and Task2.E. The median correctness for the control (Ctrl) task was 60% while the median correctness for the experimental (Exp) task was also 60%. We evaluated these results through the Wilcoxon Signed-rank test with W equal to 14.5. A p-value of 0.674 indicates that no significant difference in understanding was observed for tasks with and without versioning support. In our further analysis of participant behavior, we noticed that a majority of users (five of nine) did not refer to the versions when performing Task2.E, which is similar to our observations in Task1.E. Note that we did not measure completion times for this task.

4.2 Study II

Our first study offered support for the claim that versioning can help mashup programmers create mashups more efficiently and effectively. In our second study we wished to further investigate that claim, but with a more diverse population of users including both computer science experts and end users. This will help us determine whether our results generalize, and also allow us to compare the two classes of users. Furthermore, given that our first study did not provide evidence that versioning helped in mashup understanding, we wished to investigate a further dimension of versioning; namely, whether our versioning support can aid users in performing debugging tasks. We thus framed the following research questions:

- **RQ1:** How does versioning help mashup programmers create mashups more effectively and efficiently?
- **RQ2:** How does versioning help mashup programmers debug mashups more effectively and efficiently?
- **RQ3:** How does versioning benefit computer science participants and end users?

4.2.1 Pipes Plumber Platform

To provide support for debugging, for this study we enhanced our versioning interface to include the tags ERROR and Tested in the history of pipes list, and to also display each version's parentage information and the "Tested" widget (as discussed in Section 3).

4.2.2 Participants

We recruited 24 students from the University of Nebraska–Lincoln by sending emails to student mailing lists across departments at the university. Participants were promised $20 for their participation in the study. Participants were selected from those responding to our email on a first-come, first-served basis. Twelve of the participants were from the Computer Science department and had formal training in programming, and the other twelve were from other departments and had no formal training in programming. The distribution of the majors among the the latter students was Electrical Engineering (2), Mechanical Engineering (1), Civil Engineering (4), Transport Engineering (1), Biological Systems (2), Materials Science (1), and Actuarial Science (1). Participants' ages ranged from 19 to 40 years; 22 were male and two were female. Only two of

the participants had prior experience with the Yahoo! Pipe environment, and neither of them had created more than five pipes.

4.2.3 Independent and Dependent Variables

Our independent variable involved the presence or absence of versioning information, with the former provided via Pipes Plumber. To measure effectiveness and efficiency we used two dependent variables tracking (1) the correctness of pipes following creation or debugging activities, and (2) the time required to create or debug a pipe.

4.2.4 Study Setup and Design

Our study setup and design followed that used in Study I. Therefore, in our study each participant completed pipe creation and debugging tasks both with and without versioning support.

We conducted the study in the Usability Lab at the University of Nebraska–Lincoln, where we could observe and record transcripts for each participant. We audio recorded the session and logged the participant's on-screen interactions, again using Morae[11], but we did not employ a think-aloud methodology, nor did the experimenter provide any further input while specific tasks were ongoing. The study protocol was administered to each participant individually, with the first author conducting each session.

The total time required for completion of the study was approximately two hours, which included around 60 minutes for actual task performance. After all tasks were completed, we administered an exit survey to obtain further feedback from the participant.

4.2.5 Tasks

Our study included two types of tasks, each addressing one of the research questions.

Task1 required participants to create pipes, given pipes that they could choose to reuse portions of. *Task2* required participants to debug pipes. To enable a within-subjects study we again created two distinct subtasks for each type of task (reusability and debugging) so that we could examine the participants' experience with and without versioning for that type of task. Task1 was followed by Task2 for all the participants. However, the tasks within Task1 and Task2 were counterbalanced; that is, *Task1.search* was a control task for half of the participants and was an experimental task for the other half of the participants. The sequence in which the treatments were administered was also counterbalanced; that is, half of the participants performed experimental tasks before the control task and the other half performed the control task before the experimental task. Next we describe each subtask in detail.

Task1

Task1 involved two steps. The first step required a participant to understand the functionality of a given pipe. The second step required them to create a pipe that had some functionality in common with the given

[11]Morae: http://www.techsmith.com/morae.asp.

pipe. They could complete this task either by creating the new pipe from scratch or by reusing some of the modules in the given pipe. The two given pipes were of similar complexity and involved similar numbers of modules (13-14 each). The two subtasks were as follows.

In *Task1.search,* participants were given a pipe that lets users search for a review of any item within a given distance of a given location (e.g, reviews of museums within ten miles of Lincoln, Nebraska). The distance from the location is at the discretion of the user and hard-coded in the pipe. The user can choose the number of reviews to be displayed. The results of the search are translated into Korean. The titles of the reviews are unique (i.e., duplicates were filtered out) and they are presented in alphabetical order.

The next step required participants to create a pipe similar to the one used in the first step. The difference was that this step required participants to implement extra functionality; namely, to rename all of the titles in the search result with a title of their choice (e.g, if a participant specifies the title to be "museums" then he or she will see all titles replaced by "museums"). Further, other features such as sorting results, filtering out duplicates, and translating into Korean were not required of this pipe.

In *Task1.blog*, participants were given a pipe that allows a user to search for any topic in a set of blogs hosted on websites such as Blogpulse and Technorati. The pipe then merges the results into a single feed, displaying only unique results and sorting them in ascending order. It also allows the user to truncate the number of results based on user input at run time, and allows users to filter out items in which they are not interested (e.g., filter out crime items from the search results). Finally, the pipe provides a status display regarding the total number of results generated by the query and the truncated results.

In the next step, participants were asked to create a pipe similar to the one used in the first step, but with the additional requirement that the results be translated into Spanish. Similar to the previous task, the new pipe did not require the results to be unique or sorted or filtered on a given keyword. The display status from the previous step also was not required and was considered extraneous.

Task2

In Task2, participants were given pipes into which we had seeded two faults. Each pipe included 15-16 modules. Participants were given detailed requirements about the functionality of the correct pipe along with an example of the output of the correct pipe. Participants were required to correct the seeded faults and ensure the pipe was working as stated in the requirements by matching their results with the sample output. The two subtasks were as follows.

Task2.movie involved a pipe that allows a user to generate a list of local theaters by inputting their zip code. The pipe then collects a list of movies and displays them, together with show times and geolocation, on Yahoo! Maps. The user could also obtain a poster and reviews of a movie.

Task2.eBay involved a pipe that allows a user to search for an item on an auction site such as eBay, or in a list of classified ads on Craigslist (e.g., San Francisco Craigslist). The pipe allows the user to set a price range by inputting minimum and maximum amounts. Search results include the name of the site from

which the item was retrieved. The user has the option to limit (truncate) the number of results displayed.

Measures

To evaluate whether versioning helped pipe programmers in their tasks, we measured the time spent performing tasks and the correctness of the pipes. Time was measured in minutes while correctness pipes was measured with scores ranging from 0 to 100. On *Task1*, we used the same measures used to judge the correctness of Task1 in Study I. On *Task2*, 80 points were assigned if participants successfully identified and corrected the seeded errors in the given pipe. There was no partial credit for just identifying the error. The remaining 20 points were allocated to other errors; specifically, if participants added new errors when implementing the task they were penalized 10 points, and if they added unrelated modules, they were penalized five points apiece (up to a maximum of 10 points total).

4.2.6 Threats to Validity

This study has threats to validity similar to those already discussed in Study I. The primary differences involved the increased size and diversity of the participant pool, and our counterbalancing of tasks and assignment of participant to Control and Experimental groups.

4.2.7 Results and Analysis

We now address our research questions in turn; we then provide additional data on participant feedback.

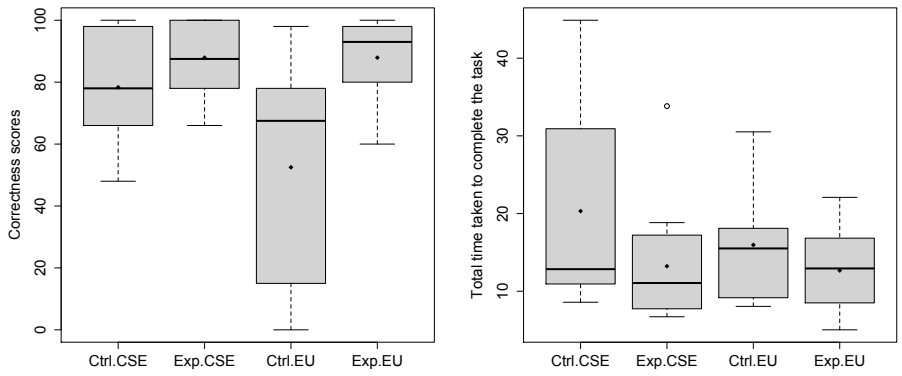
Research Question 1

To address our first research question we frame it as two hypotheses. These hypotheses analyze the effect of the independent variable (presence or absence of versioning) over the correctness of pipes created and the time required to complete the task, respectively.

Ha1.1: Pipes created with versioning support are more correct than those created without versioning support.

Figure 6(a) provides boxplots displaying the distribution of correctness scores for the participants on the two subtasks, per participant group, with CSE denoting Computer Science and Engineering participants, and EU denoting the other ("end-user") participants. For both participant groups, median correctness scores were higher for Experimental groups (when versioning support was used) than for Control groups (no versioning support), and variation in scores was lower, with the results more pronounced for EU participants. The median correctness score for CSE participants was 78.0% when not using versioning and 87.5% when using versioning. The median correctness score for EU participants was 67.5% when not using versioning and 93.0% when using versioning.

We blocked over participant group when performing our statistical analysis. A SPANOVA [8] on the results, performed using the R programming language [42], shows statistically significant differences (at α=0.05) between the treatments (F=12.518, df=23 and p=0.002), indicating that versioning helped pipe



(a) Correctness

(b) Time

Figure 6: Correctness and time results boxplots for Task1: Dots signify means.

programmers create more correct pipes.

Ha1.2: Pipes created with versioning support require less time to create than those created without versioning support.

As mentioned earlier, we recorded the time taken by participants to perform each of the subtasks. During the experiment, two EU participants did not correctly save their pipes after they had finished their implementation, because of which they lost their changes and had to redo their tasks from scratch. We thus exclude those two participants from our sample where time costs are concerned.

Figure 6(b) provides boxplots displaying the distribution of time values (in minutes) spent by the participants in the two subtasks, separated by participant group. Here, for both participant groups, median time costs are lower when versioning support was used, and variation in time is lower, with the variation more pronounced for the CSE participants. The median time cost for CSE participants was 12.8 minutes when not using versioning and 11.1 minutes when using versioning. The median time cost for EU participants was 15.5 minutes when not using versioning and 12.9 minutes when using versioning.

We again blocked over participant group when performing our statistical analysis. A SPANOVA on the results (at α=0.05) shows marginally significant differences between the treatments (F=3.569, df=21 and p=0.073), suggesting that versioning plays some role in helping pipe programmers create pipes more quickly, but with less confidence.

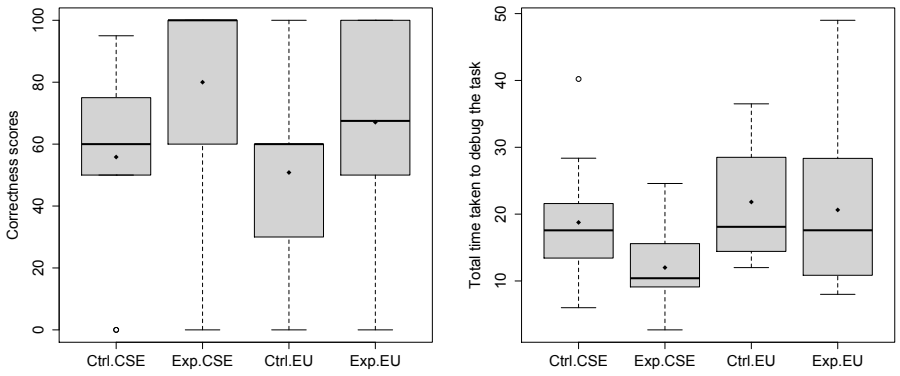
Research Question 2

We frame our second research question as two hypotheses as well, analyzing the effect of the independent variable (presence or absence of versioning) on the abilities of pipe programmers to debug pipes, in terms of

time cost and correctness outcome.

Ha2.1: Versioning support helps pipe programmers debug pipes more correctly than the absence of support.

Figure 7(a) provides boxplots displaying the distribution of correctness scores for the participants in the two subtasks, per participant group. For both participant groups, median correctness scores are higher when versioning support was used than when it was not. The median correctness score for CSE participants was 60.0% without using versioning and 100.0% when using versioning. The median correctness score for EU participants was 60.0% when not using versioning and 67.5% when using versioning.



(a) Correctness (b) Time

Figure 7: Correctness and time results boxplots for Task2. Dots signify means.

We again blocked over participant group when performing a SPANOVA on the results. This analysis revealed statistically significant differences between the treatments (F=10.189, df=23 and p=0.004) indicating that versioning did help pipe programmers in debugging with respect to the correctness of their output.

Ha2.2: Versioning support helps pipe programmers debug pipes more quickly than in the absence of support.

Figure 7(b) provides boxplots giving the distribution of correctness scores for the participants in the two subtasks, per participant group. Here, results differ across participant groups. The median time cost for CSE participants was 17.6 minutes when not using versioning and 10.4 minutes when using versioning. The median time cost for EU participants was 18.1 minutes when not using versioning and 17.6 minutes when using versioning.

In this case, a SPANOVA did not reveal statistically significant differences between the treatments (e.g., F=2.873, df=23 and p= 0.104); we cannot support our hypothesis.

Research Question 3

To address our third research question we framed it as four hypotheses. These hypotheses analyze the

effect of the independent variable (presence and absence of versioning) over the correctness and time variables, for both the pipe creation and debugging tasks, relative to CSE and EU participants.

To address each of the four hypotheses we consider the relevant data along two dimensions. First, we compare results within participant groups. We begin by comparing the results obtained for CSE participants in the Control group to those obtained for CSE participants in the Experimental group to determine whether the participants' behavior was statistically significantly different. In this case we use paired t-tests because the within-subjects design ensures that each participant is in both groups. We then perform the same comparison relative to the EU participants. If one but not the other comparison reveals statistically significant differences, then one group benefits from versioning while the other does not and our hypothesis is supported.

Next, we compare the performances across participant groups. We begin by comparing the results obtained for CSE participants in the Control group to the results obtained for EU participants in the Control group. Then we do the same with CSE and EU participants in the Experimental groups. In this case we use t-tests because participants in each group are different individuals. If one but not the other comparison reveals statistically significant differences, then again, the two participant groups benefit differently from versioning and our hypothesis is supported.

Table 2 presents p-values from the foregoing statistical tests; each row pertains to results for a specific task (Column 1) and dependent variable (Column 2). Columns 4 - 7 present results for each of the four pairs of participant groups that are statistically compared. To facilitate comparison, Column 3 provides p-values measured in relation to all participants (Research Questions 1 and 2). We describe the data in the table further as we consider each of the four hypotheses.

Table 2: Behavior of CSE and EU Participants Without and With Versioning

Tasks	**Dependent Variable**	**Ctrl vs Exp**	**Ctrl.CSE vs Exp.CSE**	**Ctrl.EU vs Exp.EU**	**Ctrl.CSE vs Ctrl.EU**	**Exp.CSE vs Exp.EU**
Task1	**Correctness**	**0.002**	0.108	**0.006**	**0.0349**	1.0
(Creation)	**Time (minutes)**	0.073	0.138	0.369	0.381	0.413
Task2	**Correctness**	**0.004**	**0.023**	0.097	0.702	0.338
(Debug)	**Time (minutes)**	0.104	0.067	0.732	0.409	**0.045**

Ha3.1: CSE and EU participants behave differently in the presence of versioning support in relation to pipe correctness when creating pipes.

The first row of data in Table 2 presents the p-values obtained through statistical tests that address this hypothesis. We begin with comparisons within participant groups. As Column 4 of the table shows, the correctness scores of CSE participants in the Control and Experimental groups were not statistically significantly different (p-value 0.108). In contrast, as Column 5 of the table shows, the correctness scores of EU participants in the control and experimental groups were statistically significantly different (p-value 0.006). This shows that versioning benefited EU but not CSE participants.

Next we consider comparisons across participant groups. As Column 6 of the table shows, the correctness scores of CSE and EU participants in the Control group were (statistically) significantly different (p-value 0.035). In contrast, as Column 7 shows, the correctness scores of CSE and EU participants in the Experimental groups were not statistically different (p-value 1.0). The use of versioning eliminated differences between the two participant groups that exist when versioning is not present.

Both of these comparisons show, therefore, that hypothesis Ha3.1 is supported: CSE and EU participants behave differently in the presence of versioning support in relation to pipe correctness when creating pipes.
Ha3.2: CSE and EU participants behave differently in the presence of versioning support in relation to pipe creation time when creating pipes.

The second row of data in Table 2 presents the p-values obtained through statistical tests performed to address this hypothesis. In this case, none of the comparisons (within or across participant groups) reveals any statistically significant differences. Hence, we cannot support hypothesis Ha3.2; versioning does not cause participant behavior to differ in relation to task completion time during pipe creation.

Ha3.3: CSE and EU participants behave differently in the presence of versioning support in terms of correctness when debugging pipes.

The third row of data in Table 2 presents the p-values obtained through statistical tests performed to address this hypothesis. In this case, we see that (column three) the correctness scores of CSE participants in the Control and Experimental groups are statistically significantly different (p-value 0.023), while in all other comparisons no statistically significant difference exists. Nonetheless, the single difference does indicate that versioning benefited CSE participants but not EU participants, and hypothesis Ha3.3 is supported.
Ha3.4: CSE and EU participants behave differently in the presence of versioning support in relation to time to debug.

The fourth row of data in Table 2 presents the p-values obtained through statistical tests performed to address this hypothesis. In this case, we see that (column six) the task completion times of CSE participants are statistically significantly different from the task completion times of EU participants, when those participants are in the Experimental groups (p-value 0.045). In other comparisons no statistically significant differences exist. Nonetheless, the single difference does indicate that versioning benefited participants from the two groups differently, and hypothesis Ha3.4 is supported.

Participant Feedback

As mentioned in Section 4.1.4, we ended each experiment session by administering a survey to the participants. The survey consisted of both closed and open-ended questions about the tasks, the interface of our versioning support, and the experiment process. In the survey we asked participants to provide ratings on a Likert scale from 1 to 5 (1 indicating "don't believe" and 5 indicating "strongly believe"). The questions asked were related to (1) whether they believed that the versioning helped them understand the evolution of pipes, (2) whether it improved the reusability of the pipes, (3) whether it helped them debug pipes, and

(4) whether it improved the overall interface of Yahoo! Pipes.

Table 3 summarizes the results, showing the mean and standard deviation of the responses by the two different participant groups, and the overall results. The data indicates relatively strong beliefs that versioning helped in all four categories.

Table 3: Participant Feedback about Versioning Support

Variable	Mean (SD)		
	CSE	EU	Total
Helps understand pipe evolution	4.2 (0.8)	4.4 (1.2)	4.3 (1.0)
Improves pipe reusability	4.3 (0.6)	4.2 (1.0)	4.3 (0.8)
Helps in debugging	4.0 (1.3)	4.2 (1.1)	4.1 (1.1)
Overall interface improvement	4.0 (1.0)	4.2 (1.1)	4.1 (1.1)

5 Qualitative Analysis

One of the key goals of our study was to evaluate the usefulness of versioning support in the domain of mashup creation. Since ours is one of the first attempts we are aware of to study versioning support in an end-user domain we wanted to qualitatively investigate the role and impact of versioning in enabling users to reuse and debug in a web mashup domain, namely Yahoo! Pipes. Our qualitative analysis aims to provide insights into the challenges that users face when reusing and debugging pipes and the usefulness of versioning in overcoming these challenges.

We analyzed the results of our experiments by first transcribing the experiment videos and interviews. The transcripts were then coded with three primary code sets: (1) programming barriers that participants faced while performing their tasks (Table 4), (2) reuse activities (Table 7), and (3) debugging strategies (Table 10). We describe these code sets in the following sections. For each code set, two researchers coded small portions of the transcripts independently and compared inter-coder agreement until they reached an 80% agreement covering at least 20% of the transcripts. Once an agreement about the code and its context was achieved the first author coded all the remaining transcripts. We used the Qualyzer[12] tool for coding the transcripts and aggregating the codes.

The results of this qualitative analysis are divided into three sections. Section 5.1 investigates the programming barriers defined by Ko et al. [29] as they apply to Yahoo! Pipes. It then discusses activities that users performed to overcome those barriers, and the differences between End-User (EU) and Computer Science and Engineering (CSE) participants in how they faced and overcame the barriers. Note that this section includes observations across both studies and both tasks. Section 5.2 explores reuse activities performed in Task 2 in both studies (Studies I and II). We then discuss how EU participants differed from CSE participants when performing reuse in Study II. Section 5.3 discusses debugging strategies used by study

[12]Qualyzer: http://qualyzer.bitbucket.org/#home.

Table 4: Programming Barriers Adapted from Ko et al. [29] and Cao et al. [3]

Barriers	Definition	Example
Understanding	Not knowing why the program behaved the way it did	User is unable to understand the type of error and its cause when viewing the output (of a module) in the debugger window
Use	Not knowing how to use a module	User did not know how to set parameters in the `loop` module
Coordination	Not knowing how to connect modules	User tries to connect text input module with count module, which are incompatible because of incompatible output and input types
Selection	Not knowing which module to use for a particular behavior	User incorrectly selects `rename` instead of `string replace` module while appending a prefix to titles of search result
Design	Not knowing how to frame the problem and arrive at a solution	User performs multiple reworks, by removing all previously modules added and starting from scratch

participants (Task 2 in Study II) along with differences observed among EU and CSE participants. Because Study I did not differentiate between EU and CSE participants, participants in this study are referred to as P[x], whereas in Study II, participants are referred to as EU[x] or CSE[x] based on their background.

5.1 Programming Barriers

Prior studies of end users creating mashups have observed that users faced significant challenges [3]. In our study, we wished to determine whether similar challenges existed in Yahoo! Pipes, one of the most popular mashup environments. Further, we wanted to understand whether knowledge of the provenance of a pipe and the ability to revert to an earlier successful version of a pipe would reduce some of the challenges. Another goal was to gain insights into how versioning support was understood and used differently by EU versus CSE participants.

5.1.1 Programming Barriers and Versioning Support

Ko et al. [29] identify six types of programming barriers that end users might face when programming in a new environment. Five of these are observable in mashup domain as previously identified by Cao et al. [3] in their study of Microsoft Popfly (a web mashup domain). Table 4 presents the list of barriers, which includes *understanding, use, coordination, selection, and design*; it also provides definitions and examples. We analyzed instances where participants faced problems using Yahoo! Pipes and classified each instance with the type of barrier (Table 5). We then investigated the occurrences of each barrier to gain insights into the challenges that participants faced and the strategies that they used to overcome the barriers. Our discussion of barriers is ordered based on the frequency with which these barriers were encountered.

There was a greater incidence of the first four barriers than of design barriers. This was an artifact of our task design. Design barriers occur primarily when users try to formulate (design) a solution to a problem. Our tasks did not require users to create a pipe from scratch; instead, our participants were given a pipe that they could reuse in Task 1 (Studies I and II) and a faulty pipe that they needed to debug in Task 2 (Study II). Because we saw few instances of design barriers we do not discuss them further here.

Table 5: Instances of Learning Barriers With and Without Versioning

Barriers	**Instances of Learning Barriers**	
	Ctrl	**Exp**
Understanding	135	84
Use	44	51
Coordination	37	16
Selection	31	26
Design	7	6
Total	**254**	**183**

Understanding barriers occur when the program's externally visible behavior obscures what the program did (or did not do) during execution. Yahoo! Pipes allows users to check the output of an entire pipe by clicking on the "output" module as well as the output of a single module individually. The debugger window found at the bottom of the editor interface (Figure 2) displays the results of the pipe (or the module). However, we found that the feedback provided was not self-explanatory. For example, participant EU12 in the Control group (Task 1) incorrectly used the URL for the `fetch input` module and as a result there was no output for any of the subsequent modules. However, there was no error information provided to notify him that the problem was in the input sources. EU12 ended up checking the output of each module (3-4 times) in the debugger window because of his inability to understand why there was no output from any modules. He ended up checking the output of modules for the entire pipe a total of 54 times, but was not able to identify the error or its location.

A majority of participants had trouble understanding feedback when it was available. In fact, understanding barriers were the largest barrier faced by participants in both the Control and Experimental groups (135/254 and 84/183, see Table 5). Prior work has found understanding barriers to be the most difficult for users to overcome, calling them "insurmountable" [26]. Therefore, it is pertinent to understand why these barriers occur and possible solutions.

We found that understanding barriers were reduced by 37.7% (from 135 instances to 84) in the presence of versioning support. There are two primary reasons for this reduction. First, participants who had versioning support needed to investigate fewer modules when they performed Task 2 in Study II, which required participants to debug two faulty modules. In the Control group, participants had no idea which modules were faulty and ended up having to check each module in the Pipe. Whereas participants in the Experimental group were able to view the history of development and reduce the evaluation space; that is,

they could identify the modules that were non-buggy (already checked and providing correct output) and focus on checking the rest. As a result, they investigated fewer modules and, therefore, faced fewer instances of understanding barriers. For example, although participant EU12, when working in the Experimental group, followed the same approach he had followed when working in the Control group (checking the output of each module that he needed to investigate), he needed to check the debugger window only 12 times instead of 54 times.

A second cause for the reduction in understanding barriers can be attributed to Task 1 in Study II, which required participants to comprehend the example pipe to be able to reuse parts of it in their tasks. On analyzing the participants' behavior, we found that participants in the Experimental group faced fewer barriers because they could trace the evolution of the pipe and understand the functionality of each module and its aggregated effects on the pipe as each module was added (in the past). They could also note the output of the pipe or any selected module in prior (successful) versions. Another indication of how versioning helped in comprehension is visible in the times to completion data in Section 4.1. Note that when we divided Task 1 into "comprehension" and "implementation" phases, participants in both treatment groups required comparable time for the "comprehension" part, but the Experimental group was faster and provided more correct output in the "implementation" phase. We posit that with the help of versioning, participants could comprehend the functionality of modules better and were better positioned to complete their "implementation" task.

Use barriers are caused by a lack of understanding of how to use a particular piece of code or module, and these posed the next biggest challenge for participants (95/437, adding the use barrier instances across treatment groups in Table 5). For example, participant EU7 was frustrated when he tried to use the `loop` module to rename the titles of feed. He spent about 8.20 minutes just trying to experiment with the `loop` module to make it work correctly. Towards this goal, he looked at the example code as well as the text documentation, but was still unsuccessful. We found that many participants had problems with this module, since it required a nested operation. That is, the operation that needs to be performed iteratively (`string replace`) had to be nested within the `loop` module. We note that understanding this usage was difficult for CSE participants too. For example, participant CSE1 spent 5.10 minutes trying to figure out how to insert the `translate` module inside the `loop` module.

Even though use barriers occurred less frequently than understanding barriers, we posit that use barriers are still a challenge for users, when we consider the fact that the "reuse" task (Task 1 in both studies) was the only task that required participants to select and add modules. Additionally, participants needed to add only two new modules. This suggests that recognizing the functionality of modules in Yahoo! Pipes is nontrivial and that the documentation/help provided by Yahoo! Pipes is inadequate to allow users to comprehend the functionality of a module and know how to use it. We found that participants in the Experimental group actually exhibited a slightly greater incidence of this barrier (51 vs. 44 in Table 5). Versioning could not

have helped participants because they faced use barriers when they tried to add new functionality, and the two modules that needed to be added were not part of the original pipe. Therefore the ability to view the history of development or having access to prior successful versions was not useful.

Coordination barriers were the next most prevalent type of barrier (53/437, adding coordination barriers in Table 5). Such barriers are usually caused by a programming language's inherent limitations on how interfaces and individual programming units can be combined to perform complex functionalities [29]. In our case, this problem occurred primarily when participants tried to connect incompatible modules or incorrectly connect modules. For example, participant EU9, when performing Task 1 in the Control group, tried to connect the `truncate` module with the `translate` module. These are incompatible modules because the `truncate` module accepts only RSS "items" as input and output, whereas the `translate` module only accepts "text" for input or output. Although this connection is forbidden, the Yahoo! Pipes interface does not provide appropriate error feedback: the "wire" simply cannot be attached between the two modules. Because of this lack of feedback EU9 tried to perform this action three times before looking for an alternative solution. However, he could not correctly interpret what had gone wrong.

Another example of a coordination barrier occurred when a participant tried to incorrectly connect two compatible modules. For example, participant CSE6 could not understand the dataflow paths across modules in Yahoo! Pipes, where the output of a module needs to be connected to the input of another module. He tried to connect the output of module `loop` with the output of module `string builder` when he was trying to rename the titles (although, he should have inserted the `string builder` module inside the `loop` module). Because of the lack of appropriate feedback, CSE6 did not understand why he could not connect the modules and repeated this operation four times before looking for alternative solutions.

We found coordination barriers to be more than halved in the presence of versioning support (16 vs. 37 instances, see Table 5). We found that coordination barriers were lower in the Experimental group because participants could view examples of how modules are supposed to be connected by viewing prior successful states of the pipe. For example, participant EU4 in Task 2 tried to connect incompatible modules (the `number input` module which takes a "number" as input to the `truncate` module which takes "items" as input). He did not know why he was unable to connect them and commented *"oh! It didn't connect ... this is interesting"*. He then went back to an earlier version of the pipe to view an example of how these modules were connected. After observing the connections in the older version he was able to successfully connect the modules and complete his task correctly.

Selection barriers were the next most prevalent type of barrier faced by participants (57/437, adding selection barriers in Table 5). These barriers are caused when a user is unable to identify the right programming interface and how to achieve the correct behavior for a given program unit. In our situation, participants faced this problem primarily in the reuse task (Task 1, Studies I and II), where they had to add new functionality that required the addition of two new modules. Many participants had difficulty selecting

the right Yahoo! Pipes module to implement the functionality. For example, in Task 1 participants were required to rename the titles of the search results to a title of their choice. To do so, they had to use the operator `string replace` that replaces one string with another; however, many participants selected the wrong module (`rename`). In Yahoo! Pipes the `rename` module is used to define mapping rules of different input formats (e.g., map input parameters to RSS formats). We found that participants made this mistake because: (1) the `rename` module appears before the `string replace` module in the list of available actions, and (2) `string replace` is a computer science term that end users are not familiar with. For example, participant EU6 had this problem when he used the `rename` module and could not identify the correct module to use even after checking the help facility, which includes definitions of modules and examples of their use.

There was a small reduction in selection barriers when participants had versioning support. We found that, this reduction was not attained when participants had to add new functionality. Instead the reduction occurred primarily in situations where participants when creating their pipes began from scratch or incorrectly removed modules that they should have reused. In such cases, participants in the Control group faced selection barriers when they needed to figure out which modules to use, their actual behavior, and correct usage. Users in the Experimental group, in contrast, could refer to prior successful states of the given pipe to correctly use the module and therefore, faced fewer barriers. This is evident in cases where participants in the Control group explicitly asked the observer how they could go back to a prior state. For example, participant EU6 asked the observer *"I can not rename the titles can I go back to the first one I had...?"*

5.1.2 Differences Between EU and CSE Participants

In our study we investigated how EU participants behaved and performed compared to CSE participants. Several studies have shown that end users face greater incidences of programming barriers when compared to professional programmers [29, 37]. We observed similar characteristics (Table 6). For each class of barrier, EU participants typically faced more barriers than their CSE counterparts in each treatment group (i.e., comparing EU and CSE participants for both Control and Experimental groups). This was the case for understanding and use barriers (understanding barriers: 127 vs. 92, use barriers: 59 vs. 36, see Table 6).

Table 6: Learning Barriers With and Without Versioning for CSE and EU Participants

Barriers	**Instances of Learning Barriers**			
	CSE		**EU**	
	Ctrl	**Exp**	**Ctrl**	**Exp**
Understanding	63	29	72	55
Use	17	19	27	32
Coordination	17	11	20	5
Selection	13	9	18	17
Design	1	0	6	6
Total	**111**	**68**	**143**	**115**

Investigations of the other two classes of barriers reveal interesting results. When investigating coordina-

tion barriers, we saw that CSE participants faced slightly higher barriers than their EU counterparts (28 vs. 25, Table 6). On further analysis, we found that while CSE participants in the Experimental group exhibited only a marginal reduction in barriers (17 vs. 11), EU participants exhibited a much larger reduction (20 vs. 5). This was a result of EU participants actively perusing prior (successful) examples of the pipe that they were creating. For example, we discussed earlier how EU4 attempted to connect two incompatible modules, but when he was unsuccessful in doing so he viewed an earlier successful version of the pipe and corrected the problem. We posit that EU participants were more open to seeking help from the versioning support (viewing prior versions), whereas CSE participants had higher self-efficacy and attempted to resolve the problem on their own through trial and error. This implies that versioning support can help end users learn by providing examples of correct structure and use.

In the case of selection barriers, we found that CSE participants benefited from versioning, but EU participants did not (13 vs. 9 for CSE, 18 vs. 17 for EU). As noted earlier, a primary reason for the reduction in selection barriers with versioning support occurred in cases where participants began creating their pipes from scratch or incorrectly removed "correct" modules from the given pipe, which was not a predominant strategy among EU participants. Another reason EU participants performed poorly compared to CSE participants is the naming conventions of modules, which were closer to computer science conventions. EU participants were unfamiliar with such conventions and had difficulty understanding modules even when they saw an example (e.g., `string replace` vs. `rename`).

5.1.3 Overcoming Barriers

Help mechanisms provided through the programming environment or seeking help from colleagues are the most common ways to overcome programming barriers [27]. Prior studies have indicated that Yahoo! Pipes users find it difficult to solicit help through discussion forums and many of their questions go unanswered. Therefore, we believe that an understanding of the evolution of a pipe will provide help and guidance to users. Here we report on our observations on how participants used the two primary resources available to them in the study: (1) help provided by the Yahoo! Pipes environment, and (2) versioning support.

Yahoo! Pipes help: Participants in our study were allowed to use help available through Yahoo! Pipes. We classified the help available into two categories: internal and external. Internal help is help that is available within the environment, which includes: selecting a module from the list of modules and reading its help documentation by clicking on the "?" icon in the module, or hovering over the module (in a list of modules) to obtain tooltip information. External help is support available outside the Yahoo! Pipes environment and includes example code or documentation on pipes available from Yahoo!.

We found that participants utilized both external and internal help. It was interesting to note that both EU and CSE participants preferred internal help (65%) to external help: participants resorted to external help only when internal help did not suffice. Further, we found tooltips to be the most favored form of help,

as they allowed participants to look at the functionality of a module at a glance. When using external help, we found a difference between EU and CSE participant behavior. While CSE participants predominantly investigated example source code, EU participants preferred to read the text documentation. This could be attributed to the fact that CSE participants were more familiar with programming and therefore could quickly read and understand the example code, whereas EU participants being less familiar (and comfortable) with coding needed the text documentation to understand the example code.

Despite the use of extensive help by our participants, we note that this usage was not very effective in overcoming barriers. For example, in Study II, of the 19 participants who used help, only two were able to overcome the barriers. We found no correlation between the "instances of help used" and the correctness scores of pipes, which supports our observation.

Versioning support: We found that versioning support was helpful in overcoming barriers. Ko et al. [29] note that the programming barriers faced by users arise as a result of Norman's "gulf of execution" (the difference between users' intentions and actions available through the system) and "gulf of evaluation" (the effort to determine whether a desired goal has been achieved). Coordination and use barriers pose gulf of execution problems, understanding barriers pose gulf of evaluation problems, and selection barriers pose both gulf of execution and evaluation problems.

Here we investigate how versioning support helped bridge the different gulfs and thereby alleviated barriers. Norman recommends bridging gulfs of execution by establishing visible constraints on the actions that are possible. Following this reasoning coordination barriers can be reduced when implicit rules of connecting modules have explicit representations. We believe that within versioning support, especially, the ability to view prior versions allowed participants to see how particular modules were connected, making the connection rules more explicit. For example, participant P6 commented: *"It is easy to look at a partial part of the pipe in the versioning rather than looking at the entire pipe. It is easy to go back to the previous versions."*

Similarly, use barriers can be alleviated if users can find examples of usage of difficult-to-use modules. For example, many participants had difficulty determining how to correctly use the `loop` module, which could have been alleviated if participants could view an example of a correct implementation of the `loop` module. For example, when participant CSE1 had to translate iteratively, he did not know how to use the `loop` module. Instead, he found an earlier version of the pipe that contained an example of a `loop` module (`string builder` inside the `loop` module). After inspecting the code and usage of the `loop` module in the earlier version of the pipe, he successfully embedded the `translate` module into the `loop` module for his task. In a similar fashion, recommending syntactically related pipes from the repository can serve as examples to help users learn how to use a particular module.

To overcome gulfs of evaluation, Norman recommends that the current system state be accessible and understandable to users. While Yahoo! Pipes provides the ability to determine the output of a pipe at each

Table 7: Reuse Activities Adapted from Rosson et al. [38]

Reuse Activities	Definition	Example
Finding a Usage Context	Finding the right example pipe to reuse from	(none observed)
Evaluating a Usage Context	Assessing the appropriateness of the modules for reuse	A participant selected (cloned) an appropriate version from the history of pipes list as a starting point for creating his pipe
Debugging a Usage Context	Modifying the modules for reuse to fit task context	User began by executing the pipe to understand its functionality and modified the pipe to add additional functionality

module (clicking on the module lists the output in the debugger window), we found it to be insufficient. In fact, understanding barriers that were a result of gulf of evaluation problems posed the biggest challenge for participants. While versioning support cannot directly help bridge the gulf of evaluation, the ability to view the provenance of the pipe and the output of each successful version helped to an extent. For example, participant P5 commented: *"When looking at the overall finished pipe it was quite intimidating. Using the versioning tool it was much easier to follow along the order of the finished pipe to gain a better understanding"*. Another participant (EU10) noted: *"I liked being able to see in a step-by-step manner in which it (the pipe) was created (which) made it easy to find the error"*.

Finally, selection barriers that pose both gulfs can be alleviated if users can view how complex functionalities can be implemented in Yahoo! Pipe modules. We did not include such a scenario in our study; participants faced the selection barriers when they had to implement new functionality by implementing two new modules. Still, we believe that versioning support can help users by mining different examples of implementations of such functionalities from the repository history.

5.2 Reuse

Reuse has been found to be a primary mechanism by which end users create new programs [5]. In fact, a study of the Yahoo! Pipes repository that surveyed 32,887 pipes found that a majority of pipes (17,874 – 54.35%) were cloned, of which a large portion (43%) were highly similar to each other [41]. This suggests that finding an example pipe and reusing parts of the pipe is a fairly common occurrence in the Yahoo! Pipes community. However, reuse is no easy task, even for professional developers [38]. Therefore, we wanted to investigate how end users perform reuse in Yahoo! Pipes and whether versioning support can help. Task 1 in both studies (Study I and II) required participants to reuse parts of a given pipe. Here we report on our observations of participants' reuse actions. We frame our observations on reuse (as originally defined) by Rosson et al. [38] in their study of reuse in SmallTalk.

5.2.1 Reuse and Versioning Support

Rosson et al. [38] identify three primary activities that users perform when they attempt reuse: *finding a usage context, evaluating a usage context, and debugging a usage context.* Table 7 presents the three reuse activities, their definition within the context of our study, and an example of each.

Finding a usage context: Finding the correct usage context is the first step in reuse and relates to users locating the right example with which to begin their reuse task. However, since in Yahoo! Pipes the majority of users create their pipes by cloning an existing pipe, our primary goal was to understand how users internalized the functionalities of the example pipe to create their own. In our study we provided participants with an example pipe since we were not investigating how they identify the right example.

Evaluating a usage context: The second step in debugging activities includes evaluating the appropriateness of the example to the task at hand. This activity includes executing the sample code and assessing the similarity of its functionality to the functionality required to complete the task. Similar to the study by Rosson et. al. [38], our task provided participants with a working (example) pipe that included components (modules) to be reused, so that participants could easily execute the pipe to understand the functionality of the modules. Since the example pipe contained only a subset of modules that could be reused, participants needed to evaluate the context in which these modules were used and determine their appropriateness.

Participants used three options to evaluate usage context: (1) modify the example pipe by removing or adding modules, (2) create a clone of the pipe and then modify the clone, and (3) start a pipe from scratch. The majority of participants began by modifying the given pipe directly, while a few participants performed the latter two strategies.

We found versioning support to be useful in helping participants evaluate the usage context of a module. For example, to evaluate the need for a particular module, participants in the Control group removed modules that they thought were irrelevant. However, many times they performed this step incorrectly. Because of this, participants needed to remember which modules they had removed and how to correctly connect those modules to the rest of the pipe. We observed that several participants struggled and were frustrated when they made erroneous decisions and wanted to revert to the original pipe to view it. In these cases, they viewed the original pipe by opening a new instance of the Yahoo! Pipes environment and viewing the example pipe and correcting their error. One participant, in fact, created a screen shot of the original pipe and referred to it when constructing the pipe. However, in the case of the Experimental group, participants could simply revert to a previous version (before they had removed the modules) or the original pipe through the Pipes Plumber history of pipes list. For example, participant P3 commented: *"Having versioning helps to see small building blocks makes it easy to see the entire picture. In the future, modifying/editing having all versions will be helpful."*

Another instance in which versioning support was found to be useful involved cases in which participants viewed the provenance of the pipe to determine the right point from which to begin their reuse work. That

is, participants viewed the evolution history, found the version that had the majority of the functionality that was needed, executed that version to evaluate the appropriateness of the modules, and then cloned that version of the pipe for reuse. For example participant P4 noted: *"You can see how each pipe was developed along the way to final product."* However, a side effect of providing information on the provenance of the pipe was that participants selected and executed every version in the history before they settled on the version from which to begin their reuse; this caused participants in the Experimental group to take extra time to complete their task.

Debugging a usage context: The final step in reuse includes users tweaking the reusable components to fit the context of their current task. In their study, Rosson et al. [38] noted that upon finding an example component (in the Smalltalk library), programmers immediately moved into code development to try out that example. Their primary goal was to analyze the example component to understand how it would work for the new context and write new code by using the example component as a model. Once the component was plugged into the new context they would analyze the workings of the reused component and any new code through testing. This method of development is termed "debugging into existence". We observed a similar trend with participants performing incremental development (one module at a time) and interleaving exploration of the modules in the given pipe with code development (bringing in a reusable module, modifying it, then testing it). The result was a highly contextualized, incremental analysis of the example application, and tightly integrated analysis and code development phases.

Participants in our study primarily analyzed Yahoo! Pipe modules through testing, which involved running the pipe and checking the output in the debugger window. But, they followed three distinct strategies when performing code development: (1) exploring alternative ideas, (2) backtracking their changes, and (3) investigating prior successful states. We analyzed the study results to identify instances of each strategy (see Table 8).

Table 8: Instances of Explore Mechanisms Seen During Pipe Creation, With and Without Versioning

Explore	**Percentage of Instances of Explore used**	
	Ctrl	**Exp**
Alternative Ideas	80	38
Backtracking	74	36
Successful State	0	44
Total	**154**	**118**

Exploring Alternative Ideas: This strategy involves participants exploring different options to create a new functionality or remove an error. An example of this strategy was observed when participant CSE3 in Study II attempted to rename the news titles to his desired titles using the `string replace` module. When that did not provide the right output he explored using the `rename` module instead, so he removed the `string replace` module and connected the `loop` module with the `rename` module. However, that resulted

in an error (incorrect use of `loop` module). He then tried different methods (spending 7.5 minutes) for connecting the two modules.

Participants in the Control group exhibited more instances of exploring alternative ideas (80 vs. 38, see Table 8). This was primarily because participants in the Control group ended up reinvestigating some of the strategies that they had already attempted since they did not retain any explicit account of the strategies (the information resided in their heads). In contrast, participants in the Experimental group could identify the strategies that they had already investigated (modules added or removed) through the history of pipes list.

Backtracking: We labeled activities as backtracking when participants explicitly reverted their changes because the line of development that they were pursuing was not fruitful. Participants backtracked primarily because of selection and use barriers. In our earlier example, when participant CSE3 realized that the `rename` module was incorrect, he backtracked through his changes to investigate the use of the `string replace` module again. He realized that the problem was not in the module but how it was connected. He commented: *"Ummm actually it was working with the string replace, now trying to figure out how to get that to work [in a loop]."*

We found participants in the Control group to have greater instances of backtracking (74 vs. 36, see Table 8). We found that participants in both groups made erroneous changes or changes that led to different results than what was expected. In such situations, they would revert their changes. The primary reason for less backtracking among the Experimental group was that participants better understood the functionality of modules in the given pipe and hence were able to correctly reuse the needed modules. They backtracked primarily when they were adding new modules (additional functionality). In contrast, participants in the Control group backtracked through their changes for additional modules as well as modules that were being reused.

Investigating Past Successful States: Since we recorded the version history of the given pipe as well as changes made by participants, they could execute any previous successful state in the development history. We identified instances when participants in the Experimental group opened and executed a version of the example pipe (not including their changes) to understand how the development history of a pipe can aid in reuse. We note such an example when CSE2 (in Study II), while exploring how to use the `string replace` module in a `loop` module, remembered that the `loop` module was used in the sample pipe. He then saved his current version, identified the prior version that he wanted to view, and opened that version in the editor. He then tested that version to understand the workings of the particular module and after he was satisfied, he simply reverted to his latest version to resume his work. Where as a participant in the Control group who wanted to view the modules in the original pipe would have to open another instance of the Yahoo! Pipes environment. If they needed to understand the effects of a particular module, they had to manually isolate the workings of the modules of interest and then restore the original pipe – an error prone activity.

5.2.2 Differences Between EU and CSE Participants

Both EU and CSE participants behaved similarly by interleaving the two primary reuse activities: evaluating and debugging the usage context. When we investigate the different reuse strategies followed by both participant groups, participants in the Experimental group performed better (higher correctness scores) while exploring fewer alternative ideas or backtracking. However, we found the difference between the treatment groups to be higher among CSE participants compared to EU participants. For example, CSE participants in the Control group explored alternative ideas four times more than those in the Experimental group (51 vs. 13, see Table 9). Similarly, CSE participants in the Control group backtracked three times more than their Experimental group counterparts (50 vs. 17). In contrast, the differences between EU participants in the treatment groups were marginal (e.g., alternative ideas: 29 vs. 25; backtracking: 24 vs. 19). This suggests that the CSE participants were better at understanding the example pipe and needed to refer back to the original pipe fewer times than their EU counterparts. These results indicate that EU participants needed to repetitively refer to an example pipe to perform their task. Providing a means for referring to and executing prior versions while not losing their changes as can therefore benefit EU participants.

Table 9: Instances of Explore Mechanisms Used to Overcome Barriers by CSE and EU Participants

Explore	**Instances of Explore Used**			
	CSE		**EU**	
	Ctrl	**Exp**	**Ctrl**	**Exp**
Alternative Ideas	51	13	29	25
Backtracking	50	17	24	19
Successful State	0	23	0	21
Total	**101**	**53**	**53**	**65**

5.2.3 Enhancing Reuse through Versioning

Versioning support allowed participants to better understand the given pipe and the context in which modules of that pipe were used, which helped them complete their reuse task with less backtracking or exploration of alternative ideas. However, versioning support was also useful when participants were adding functionality. We found that participants who had versioning support were more adventurous and explored more modules than those who did not, because they knew they could revert to an earlier state. For example, participant EU7, who was in the Control group after previously being in the Experimental group, created a set of changes while under the impression that the changes were being saved. After a while EU7 wanted to revert to an earlier stage of his pipe (program). However, since he was in the Control group this was not possible through the interface, which led him to ask the observer: *"I don't have option for going back to an earlier version? so how should I know that it [what he has recently created] has error".* There were other participants who explicitly requested versioning help while performing their reuse tasks. For example, participant EU8 asked: *"where is the undo window?"* when he wanted to revert to his earlier changes.

Table 10: Debugging Strategies Adapted from Grigoreanu et al. [14]and Cao et al. [2]

Debugging Strategy	Definition	Example
Feedback Following	Investigating system generated feedback in debugger window	User uses `translate` module to translate English feeds to Greek. He selects the `translate` module and observes the output in the debugger window.
Code Inspection	Inspecting modules in a pipe and pipe logic	User sequentially inspects each module of a pipe while understanding or debugging her pipe.
Testing	Testing a module by trying different input values	User supplies different "movie names" as input while running his pipe to test the correctness of the pipe
DataFlow	Users explicitly tracing data dependencies through a pipe	User had trouble understanding the top-down dataflow of the pipe as he tries to connect the input of the "sort" module with the input of the "union" module .
Proceed as in Prior Experience	Referring to a code snippet encountered in prior task/example pipe	User while performing task2 encounters `loop` module which he encountered earlier (in task1). He first consults usage of `loop` module in task1 before completing task2.

5.3 Debugging

Debugging is an integral part of programming. Studies have shown that professional developers [38] as well as students [12] spend significant portions of their time debugging. End users are no different. A study by Cao et al. [3] observed that end users creating mashups spent a significant portion of their time (76.3%) in debugging. These results indicate that programmers (professionals, students, end users) "debug into existence" their programs; that is, they create their programs though a process of successive refinement [38]. Since debugging is such an essential activity, one of our goals was to observe the challenges that participants face when debugging in Yahoo! Pipes and how versioning support helps in debugging. We frame our observations by using the classification framework of debugging strategies first proposed by Grigoreanu et al. [14], which was later used by Cao et al. [2] in their mashup study.

5.3.1 Debugging and Versioning Support

Table 10 lists the different debugging strategies that participants used in our study and the definitions and examples of each strategy contextualized for our study. Table 11 displays the numbers of instances of each strategy observed in our study, with and without the use of versioning. We discuss the strategies that we observed in the order of their occurrence. In our ensuing discussion, we specify how versioning influenced debugging strategies.

Feedback following is defined as run time observation of program behavior. In Yahoo! Pipes, users can observe program behavior by viewing the output of an individual module or the entire pipe in the debugger

Table 11: Debugging Strategies With and Without Versioning

Debugging	Instances of Debugging Strategies	
	Ctrl	Exp
Feedback Following	463	218
Code Inspection	87	97
Testing	85	52
DataFlow	13	5
Proceed as Prior Eg.	22	6
Total	**680**	**378**

window. However, we found that the feedback provided was insufficient. For example, when an input module is not correctly connected to the pipe, all modules *fail silently*;, that is, there is no feedback provided in the debugger window. We found that EU participants had difficulty identifying a problem when there was no feedback to follow. For instance, participant EU12 did not connect the input correctly at the beginning of the pipe and thus none of the modules or the pipe had any feedback in the debugger window. He investigated each module in the pipe repeatedly (a total of 54 investigations for the entire pipe) to try to identify the problem. However, he was unsuccessful because of the lack of feedback. While versioning support cannot directly help users overcome this problem, it can be helpful in situations where the user can refer back to a correct example. For instance, if a participant had incorrectly removed the input connection she could revert back to an earlier successful state or view an example of a connection of the input module in a similar pipe to identify where she had made a mistake.

In cases where Yahoo! Pipes provides error feedback some of these were found to be too technical for EU participants, who had difficulty understanding the meaning of the error messages. For example, an EU participant asked the observer for help resolving an error in the `fetch data` module because he could not understand the error message in the debugger window that stated: *"failed to parse input"*. He commented to the observer *"I am confused ... why the error is there? Can you help me?"* He spent 6.6 minutes looking at the page source of the website so as to resolve that error, but was unsuccessful. He then moved on to the next task.

Finally, Yahoo! Pipes provides contextual feedback when a user is connecting modules through wires, but this feedback turned out to be non-intuitive. For example, Yahoo! Pipes disallows the connection of two modules with incompatible data types. There is built-in support to help users identify the compatible modules with which the current module can be connected. When a user tries to connect the output of a module with a wire, all compatible modules in the canvas flash. However, many participants (both EU and CSE) missed this feedback and its meaning, and ended up attempting to connect incompatible modules multiple times.

Code Inspection in Yahoo! Pipes involved participants scrutinizing the mashup logic, such as the functionality of modules, the parameter settings of modules, and the connections (wires) between modules. Participants traced the logic of each module by following the input and output of modules and their connec-

tions. Since modules in Yahoo! Pipe serve as black box entities, participants attempted to understand the logic within a module by using help (tool tips or clicking the "?" icon) or checking the module's output. We found code inspection to be performed equally frequently across the treatment groups, the only difference being that participants in the Experimental group used Pipes Plumber's history of pipes list to identify a correct/tested version and perform code inspection on those versions.

Testing of individual modules as well as the entire pipe was a fairly common practice given the black box nature of Yahoo! Pipe modules. We found that participants followed an incremental approach where they added one module at a time to the canvas and after adding the module executed the pipe to test it. The most common testing techniques involved investigating the output of a module by trying out a series of different inputs, and testing the validity of URLs externally through a browser.

A critical step in debugging is isolating the parts of the program that could be faulty and then testing it to identify the specific fault. In our studies, participants tried different approaches for doing this, but the most common involved isolating a module and testing it in isolation to identify the fault. A majority of participants used this approach in *Task2.movie* (aggregating movie reviews), which involved information from three different feeds combined together through a `union` module. Participants in the Control group, when investigating whether the fault existed in this aggregation (`union` module) step, first disconnected all the input feeds and then tested each feed (module) until they were convinced that the error was not in any of the feeds. Participants in the Experimental group were able to reduce the amount of testing by identifying the modules that had already been tested from the history of pipes list provided by Pipes Plumber. However, while Pipes Plumber helped participants identify the faulty module they did not always know how to debug it, which accounts for the large number of testing steps for EU participants. For example, participant EU11 was able to correctly identify the faulty module through the history of pipes list, but unable to correct the fault. She remarked: *"there is an error in truncate [module]. I do not know how to fix it".*

Data-flow dependencies: Yahoo! Pipes is primarily a data-driven environment and the spatial layout of the pipe typically mimics the dataflow, with modules laid out vertically and the output of a module (at a higher level) serving as an input to a module at a lower level. While such a layout is not necessary for compilation purposes, the modules and their connections are visualized in such a pattern to facilitate understanding. Thus, most participants investigated the functionality of pipes by following data dependencies. Despite the spatial layout, however, some participants in our study had trouble understanding the data flow. For example, participant CSE3 tried to connect the modules in bottom-up fashion, by connecting the input of `text input` module to the input of the `gbase` module. After this he tried to connect the output of the two modules together.

Proceed as in prior experience is a strategy in which participants refer to their prior experiences in building a similar application to help them in their current task. In our study the versioning history served as an external experience repository. Some participants viewed their earlier tasks or the given (earlier) sample

pipes to complete their current task. For example, when participant EU4 in his second task was trying to connect two incompatible modules and unable to do so, he remembered that he had used similar modules in his earlier task and opened the earlier pipes that he had completed. After checking the structure and connection logic in one of the examples he was able to correctly connect the modules in his current pipe.

5.3.2 Differences Between EU and CSE Participants

EU and CSE participants were consistent in the kinds of debugging strategies that they used and the benefits they received from versioning support, with the Experimental group needing fewer strategies to get to the correct outcome (EU participants: 378 vs. 224 and CSE participants: 318 vs. 115). While both groups needed fewer debugging strategies when using versioning support, CSE participants generally benefitted more. For example, when adding the total debugging strategies across treatment groups, CSE participants used debugging strategies 433 times, whereas EU participants used them 602 times. This difference can be attributed to the fact that CSE participants were more proficient in debugging than their EU counterparts.

Table 12: Debugging Strategies for CSE and EU Participants

Debugging	**Instances of Debugging Strategies**			
	CSE		**EU**	
	Ctrl	**Exp**	**Ctrl**	**Exp**
Feedback Following	240	78	223	140
Code Inspection	37	18	50	49
Testing	29	10	56	42
DataFlow	1	3	12	4
Proceed as Prior Eg.	11	3	11	3
Total	**318**	**115**	**378**	**224**

An interesting difference between the two participant groups is that in the Control group, CSE participants had a greater incidence of "feedback following" than EU participants (240 vs. 223, see Table 12), but fewer instances in the Experimental group (78 vs. 140, see Table 12). This difference arose largely because CSE participants primarily used "feedback following" to arrive at the correct output, whereas EU participants also relied heavily on help mechanisms. In the Experimental group, CSE participants were able to isolate the problem modules and debug them to arrive at the correct output with far fewer instances of "feedback following" than EU participants. We found that this was because EU participants needed to debug more to arrive at the correct output even after isolating the problem modules.

Few participants had difficulty understanding the dataflow concepts in the environment. This suggests that the visual "wire" like connections are a good metaphor for depicting dataflow dependencies. However, some EU participants had trouble understanding the dataflow in the given pipes (16 instances for EU participants compared to 4 by CSE participants, see Table 12). Versioning support was not helpful when participants had issues understanding this concept. We posit that unlike coordination barriers (where users do not know the right way to connect two modules), which can be alleviated by looking at examples of how

specific modules are connected, dataflow problems are more dynamic and depend on the input and cannot be alleviated by simply viewing a static picture of a pipe. Rather, a dynamic view of the data processing and flow need to be made explicit.

5.3.3 Enhancing Debugging through Versioning

The greatest help provided by versioning was allowing participants to reduce the testing space. That is, participants in the Experimental group could easily identify the parts of the pipes that had already been tested and were therefore correct, so they only had to debug parts of the pipe. While versioning cannot directly help in correctly debugging a pipe, it can provide examples of correct usage of modules that can serve as a reference. Participants can test each module as it was added in its development to identify problem spots and focus their debugging efforts on those spots.

5.4 Implications

Our qualitative analysis prompts us to suggest several implications for ways in which mashup programming environments can be improved; these involve the provision of better information, testing and debugging support, and recommendation systems.

5.4.1 Providing Better Information

We found that a large number of the programming barriers faced by users were caused because of poor feedback and help functionality currently provided by Yahoo! Pipes. For example, the largest barriers (understanding) arose because users could not understand the runtime feedback or error messages. Providing explicit feedback in a language that is accessible to end users can alleviate such programming barriers. Improved documentation will help with understanding, use, and selection barriers. Similarly, built-in examples may help users understand the general usage of modules and thereby reduce both use and coordination barriers. Improved tool tip help functionality could provide quicker access to such documentation, as we found participants to overwhelmingly prefer help within the environment to help found outside.

Better feedback in the form of visual cues that build upon intuitive color and visual concepts could also help users to be more effective in creating wire-oriented mashups. For example, we observed that users tend to follow the color interpretations that we use in our daily life (e.g., green depicts go, red means error/danger), which was not consistently used in Yahoo! Pipes. Currently, Yahoo! pipes highlights the module that is under inspection (or debugged) in an orange color, which led some of the participants to have difficulty in determining whether the orange should be seen as an error or a warning sign. For instance, one of the EU participants asked *"Why is the module orange? Does it mean I have error?"* Using more appropriate colors could remove such sources of confusion.

Finally, our versioning interface could be enhanced to provide more intuitive views of versioning history and differences. Current source code revision tools provide editors that allow the comparison of two source

files side by side [17]. We could implement a similar editor where differences between versions of pipes could be shown by providing a side-by-side views of the pipe and highlighting the differences. Further, currently we show a linear history of the evolution of pipes, which can be improved to show provenance of the particular parent for each version in the pipe history.

5.4.2 Testing and Debugging

There is a need for better testing techniques and better debugging tools to help pipes programmers create dependable mashups. Grammel and Storey [13] have stressed the need for such techniques. Our observations underscored this need. In our studies, end users had difficulty locating the sources of errors, and engaged in testing activities in an ad-hoc manner. Support for more rigorous methodologies for ensuring dependability would help users attain more dependable mashups. Support for better error reporting would help with both assessing and correcting problems that occur. In addition, other testing and debugging techniques created to assist end-user programmers such as whyline [28] and WYSIWYT [11] may be of further help. Finally, once pipes have been constructed and made available to the community, the community itself may be of help. Social recommender system like *"HelpMeOut"*, that aids debugging error messages by suggesting solutions that peers have applied in the past are known to be helpful [16].

5.4.3 Recommendations

A large percentage of programs in Yahoo! Pipes are clones (or pipes that have been reused) [41] and learning by examples has been shown to be an effective technique for end users [33]. Therefore, recommendation systems that identify syntactic and semantically similar pipes can help end users in their development efforts. Keeping histories and versions in the repositories will help in creating more sophisticated recommendation systems. Information on usage patterns and evolution of mashups can be used to recommend more fine-grained mashup components than can currently be identified by users. Further, since version histories will be capable of capturing the successful and unsuccessful states of mashups, more stable and correct versions can be recommended. Finally, social recommendation systems that can connect novices with the creators of the example pipes can allow the user to learn from the community.

We also believe that recommendation systems that can guide users in their attempts to use modules in Yahoo! Pipes will help alleviate various learning barriers. By considering modules in terms of their structures and parameters such recommendation systems should be able to suggest not only entire pipes, but also types of parameters to be used in a module. In summary, recommendation systems should be able to suggest types of parameters, example pipes, and people to connect with to the user.

6 Related Work

There has been quite a bit of recent research related to mashups. Zang and Rosson [49] investigate the types of information that users encounter and the interactions between information sources that they find useful in relation to mashups. The authors also examine data gathering and integration [50] and discuss results of a study of web users focusing on their perceptions of what mashups could do for them and how they might be created. They also note that, when asked about the mashup creation process, end users could not even describe them in terms of the three basic steps of collecting, transforming, and displaying data.

There has been recent research aimed at understanding the programming practices and hurdles that mashup programmers face in mashup building. Cao et al. [3] discuss problem solving attempts and barriers that end users face while working with mashup environments, and describe a "design-lens methodology" to view programming. Cao et al. later [2] also study a debugging perspective on end-user mashup programming.

Researchers have also begun to empirically study the Yahoo! Pipes environment itself, in order to understand the programming practices and issues faced by end-users and their communities. Jones and Churchill [22] describe various issues faced by end users while developing web mashups using the Yahoo! Pipes environment. They observe the conversations of the users in discussion forums in order to understand the practices followed, problem solving tactics employed, and collaborative debugging engaged in by these online communities of end users.

Dinmore and Boylls [7] empirically studied end-user programming behaviors in the Yahoo! Pipes environment. They observe that most users sample only a small fraction of the available design space, and simple models describe their composition behaviors. They also find that users attempt to minimize the degrees of freedom associated with a composition as it is built and used.

Stolee et al. [41] analyzed a large set of pipes to understand the trends and behaviors of end users and their communities. They observe that end users employ the repository in different ways than professionals, do not effectively reuse existing programs, and are not aware of the community. In another study [40] they classified various "smells" (programming errors) found in Yahoo! Pipes and devised refactoring techniques that can be used to remove those smells from the pipes.

Versioning has also been the subject of substantial prior work. Versioning capabilities are heavily used in commercial software development and are required for a team to be successful. Versioning is used by professional developers to keep track of changes (theirs and others), share or benchmark the latest versions of their code, or revert their changes [45]. Tools such as diffIE have been used to keep track of changes in web pages [43, 44]. Our versioning capabilities also allow end users to keep track of changes and revert their changes. In addition to this, we allow users to benchmark significant events, as an aid in debugging mashups.

History mechanisms such as undo or "time travel" enable revisitation of earlier versions in a variety of

applications [1, 6, 9, 35]. History tools can play an important part in visualization processes, supporting iterative analysis by enabling users to review, retrieve, and revisit visualization states. Graph visualizations can help to present, manage and export histories [10, 18, 19, 24, 25].

There has been some work in revision control on visual interfaces specifically meant for capturing design histories. This includes work related to revision histories and determining differences between UML diagrams [4, 36], CASE diagrams [34], and statecharts [39]. Even for facilitating design some interaction design tools exist to track and visualize changes [15]. Our work differs from these in two aspects: (1) these revision control mechanisms are used for designing software or visual media (WYSIWYG document editors, movie producers and video game developers) whereas ours targets mashups, and (2) these mechanisms are built for designers whereas our target population is end users.

Some end user environments such as Google Docs and Google Websites provide basic versioning facilities to enable group editing, but these capabilities are only for text edits. These environments also allow versions to be created on each save. Our versioning capabilities differ from these as we provide a list-view of the pipes, which helps end users view the abstract constructs of pipes.

7 Conclusion

We have presented our Pipes Plumber extension to Yahoo! Pipes, which adds versioning support for mashup programmers using that environment. Our empirical results studying the use of that environment in mashup creation and debugging tasks provide evidence that our versioning support can help mashup programmers create and debug mashups, and this includes both individuals who have formal programming experience and those who do not. Our qualitative analysis reveals additional insights into the ways in which versioning addresses barriers faced by mashup programmers, reuse problems, and problems in debugging.

We intend to extend our environment to provide further visualization support to mashup programmers. In particular, we will consider methods for presenting version histories in forms such as family-tree-like structures, which may be appealing metaphors for end users. As discussed in Section 6, various visual representations for viewing the histories of graphs or documents have been suggested in prior work (e.g, [10, 18, 19, 24, 25, 47]. There has also been work in the web domain on methods for keeping histories of the webpages visited and visual ways for depicting these histories. [21]. This prior work may provide useful mechanisms for use in mashup programming environments. We also intend to conduct additional studies of our versioning approach, in particular using larger pipe artifacts, and considering longer term scenarios in which participants construct pipes and versions over extended periods of time.

Acknowledgments

This work was supported in part by the AFOSR through award FA9550-10-1-0406 to the University of Nebraska-Lincoln. We thank Branden Barber and Amanda Swearngin for helping transcribe recording

sessions of the participants and helping with the grading task. We thank Margaret Burnett for her feedback. We also thank our study participants.

References

[1] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *Transactions on Computer Human Interaction*, 1:269–294, 1994.

[2] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A debugging perspective on end-user mashup programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 149–156, 2010.

[3] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. End-user mashup programming: Through the design lens. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1009–1018, 2010.

[4] P. Chen, M. Critchlow, A. Garg, C. Van Der Westhuizen, and A. Van Der Hoek. *Software Product-Family Engineering*. Springer Verlag, 2003.

[5] A. Cypher, M. Dontcheva, T. Lau, and J. Nichols. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010.

[6] Mark Derthick and Steven F. Roth. Enhancing data exploration with a branching history of user operations. In *Knowledge Based Systems*, pages 65–74, 2001.

[7] Matthew D. Dinmore and C. Curtis Boylls. Empirically-observed end-user programming behaviors in Yahoo! Pipes. In *Psychology of Programming Interest Group*, 2010.

[8] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for Research, 3rd Edition*. Wiley, 2004.

[9] W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. A temporal model for multi-level undo and redo. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 31–40, 2000.

[10] May Eric, Eric Z. Ayers, and John T. Stasko. Using graphic history in browsing the world wide web. In *International World Wide Web Conference*, pages 11–14, 1995.

[11] M. Fisher, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Transactions on Software Engineering and Methodology*, 15:150–194, April 2006.

[12] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander. Debugging from the student perspective. *Transactions on Education*, 53:390–396, Aug. 2010.

[13] Lars Grammel and Margaret-Anne Storey. An end user perspective on mashup makers. Technical Report DCS-324-IR, Department of Computer Science, University of Victoria, 2008.

[14] Valentina Grigoreanu, James Brundage, Eric Bahna, Margaret M. Burnett, Paul Elrif, and Jeffrey Snover. Males' and females' script debugging strategies. In *Proceedings of the International Symposium on End-User Development*, pages 205–224, 2009.

[15] Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. d.note: revising user interfaces through change tracking, annotations, and alternatives. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 493–502, 2010.

[16] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1019–1028, 2010.

[17] Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21:264–268, April 1978.

[18] Jeffrey Heer, Jock D. Mackinlay, Chris Stolte, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *Transactions on Visualization and Computer Graphics*, 14:1189–1196, November 2008.

[19] Ron R. Hightower, Laura T. Ring, Jonathan I. Helfman, Benjamin B. Bederson, and James D. Hollan. Padprints: graphical multiscale web histories. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 58–65, 1998.

[20] Angus F. M. Huang, Shin Bo Huang, Evan Y. F. Lee, and Stephen J. H. Yang. Improving end-user programming with situational mashups in Web 2.0 environments. In *IEEE International Symposium on Service-Oriented System Engineering*, pages 62–67, 2008.

[21] Adam Jatowt, Yukiko Kawai, Hiroaki Ohshima, and Katsumi Tanaka. What can history tell us?: Towards different models of interaction with document histories. In *Proceedings of the Hypertext and Hypermedia*, pages 5–14, 2008.

[22] M.C. Jones and E.F. Churchill. Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In *Proceedings of the International Conference on Communities and Technologies*, pages 51–60, June 2009.

[23] Michael Jones and Christopher Scaffidi. Obstacles and opportunities with using visual and domain-specific languages in scientific programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 9–16, 2011.

[24] Shaun Kaasten and Saul Greenberg. Integrating back, history and bookmarks in web browsers. In *Extended Abstracts on Human Factors in Computing Systems*, pages 379–380, 2001.

[25] Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. Where do web sites come from?: capturing and interacting with design history. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1–8, 2002.

[26] A.J. Ko, B.A. Myers, and H.H. Aung. Six learning barriers in end-user programming systems . In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, 2004.

[27] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the International Conference on Software Engineering*, pages 344–353, 2007.

[28] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 151–158, 2004.

[29] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, 2004.

[30] Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel. History repeats itself more easily when you log it: Versioning for mashups. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 69–72, 2011.

[31] Sandeep Kaur Kuttal, Anita Sarma, Amanda Swearngin, and Gregg Rothermel. Versioning for mashups - an exploratory study. In *Proceedings of the International Symposium on End-User Development*, pages 25–41, 2011.

[32] C. H. Lewis. Using the "thinking aloud" method in cognitive interface design. RC 9265, IBM, 1982.

[33] Henry Lieberman, Fabio Paterno, and Volker Wulf. *End User Development*, volume 9. Springer Netherlands, 2006.

[34] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the International Conference on Automated Software Engineering*, pages 204–213, 2005.

[35] C. Meng, M. Yasue, A. Imamiya, and X. Mao. Visualizing histories for selective undo and redo. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 459–464, 1998.

[36] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 227–236, 2003.

[37] John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54:237–264, February 2001.

[38] Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *Transactions on Computer Human Interaction*, 3:219–253, September 1996.

[39] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 335–340, 2009.

[40] Kathryn T. Stolee and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the International Conference on Software Engineering*, pages 81–90, 2011.

[41] Katie Stolee, Sebastian Elbaum, and Anita Sarma. End-user programmers and their communities: An artifact-based analysis. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 147–156, 2011.

[42] Paul Teetor. *R Cookbook*. O'Reilly, first edition, 2011.

[43] Jaime Teevan, Susan T. Dumais, and Daniel J. Liebling. A longitudinal study of how highlighting web content change affects people's web interactions. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1353–1356, 2010.

[44] Jaime Teevan, Susan T. Dumais, Daniel J. Liebling, and Richard L. Hughes. Changing how people view changes on the web. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 237–246, 2009.

[45] W. F. Tichy. RCS-A system for version control. *Software – Practice & Experience*, 15:637–654, July 1985.

[46] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Springer, 2000.

[47] Allison Woodruff, Andrew Faulring, Ruth Rosenholtz, Julie Morrsion, and Peter Pirolli. Using thumbnails to search the web. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 198–205, 2001.

[48] Kwok Bun Yue. Experience on mashup development with end user programming environment. *Journal of Information Systems Education*, 21:111–119, April 2010.

[49] N. Zang and M. Rosson. What's in a mashup? And why? Studying the perceptions of web-active end users. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 31–38, 2008.

[50] N. Zang and M. Rosson. Playing with information: How end users think about and integrate dynamic data. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 85–92, 2009.