Versioning for Mashups - An Exploratory Study

Sandeep Kaur Kuttal, Anita Sarma, Amanda Swearngin, and Gregg Rothermel

Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{skuttal,asarma,aswearng,grother}@cse.unl.edu

Abstract. End users with little software background are developing numerous software applications using devices such as spreadsheets, web mashups, and web macros. Web mashups are particularly popular because they are easy to create and there are large public repositories that store them and allow their reuse. Existing repositories, however, provide no functionality for tracking the development histories of mashups. We believe that versioning capabilities can help end users develop, understand, and reuse mashups. To investigate this belief, we created a versioning extension for Yahoo! Pipes – a popular mashup environment – and conducted an exploratory study of users utilizing the environment. Our results show that versioning information allows users to perform mashup creation tasks more correctly and in less time than users not having that information, while also improving the reusability of pipes.

1 Introduction

It is estimated that in 2012 there will be 90 million end-user programmers in American workplaces [14], using devices such as spreadsheets, web mashups, and web macros to automate common tasks. End users such as these create their own code and reuse others' code to achieve a variety of goals. Often, these goals involve combining content from various websites. This has led to the development of programming environments supporting *web mashups*.

Web mashups integrate multiple data sources or APIs into one interface from various web applications. Input can be from different sources and of different formats. Similarly, outputs of mashups can be formatted in different ways and displayed in various media (SMS alerts, email alerts, etc.). Mashups can solve many different problems and can be implemented relatively quickly. They can be easily reused or distributed to other users in the community for reuse or modification, and mashups can be used as components of other mashups.

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 histories of the mashups that 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 [5]. Versioning systems provide an environment within which previous states of resources (both content and naming) can be easily retrieved. Versioning, therefore, allows engineers to

browse through past and alternative versions of a resource and identify how these different versions differ from each other.

We conjecture that the addition of versioning to mashup programming environments may (1) help end users create mashups more efficiently, (2) make mashups more reusable, (3) help users understand the evolution of mashups, and (4) allow users to utilize these advantages without being aware of the underlying complex functions such as check-in, check-out, and so forth.

To investigate this conjecture we are adding support for versioning to a mashup programming environment. We chose Yahoo! Pipes as a platform for this effort. Yahoo! Pipes is a commercial programming environment that allows users to aggregate and mashup content from around the web. The primary reason for selection of Yahoo! Pipes is its popularity; 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 [12]. The platform also has the advantages of being free, and of utilizing data that can be captured and manipulated by external systems.

In this paper, we report on the first step of our effort to create and study this versioning system. Our approach automatically captures versions without the users' direct involvement, and it allows users to access and navigate these versions through its interface. To explore the potential costs and benefits of using our versioning approach we conducted a "think-aloud" study, in which we observed users attempting to create and understand pipes with and without the aid of versioning information. Our results show that versioning information can help users create pipes more efficiently, and with fewer errors, than when no such information is available.

2 Background and Related Work

Many web mashup programming environments exist, including Yahoo! Pipes [1], JackBe [4], xFruits [7], and IBM Mashup Center [2]. While these environments are more simple than professional programming environments, they are not trivial. There has been recent research aimed at understanding the programming practices and hurdles that mashup programmers face in mashup building. Cao et al. [8, 9] discuss problem solving attempts and barriers that end users face while working with mashup environments, and describe a "design-lens methodology" to view programming. Rosson and Zang [18] investigate the types of information that users encounter and the interactions between information sources that they find useful. The authors also examine data gathering and integration [17] 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.

Yahoo! Pipes [1] is arguably the most popular mashup creation environments, and is being used both by professional and end user programmers. Yahoo! Pipes is a visual programming environment in which modules are placed on a canvas and are connected together using pipes. Figure 1 shows the Yahoo! Pipes interface. Yahoo! Pipes consists of various APIs and modules. The input and output between modules are primarily RSS feed items (e.g., some elements of *item* are date-time, url, location, text and number). RSS feeds consist of item parameters

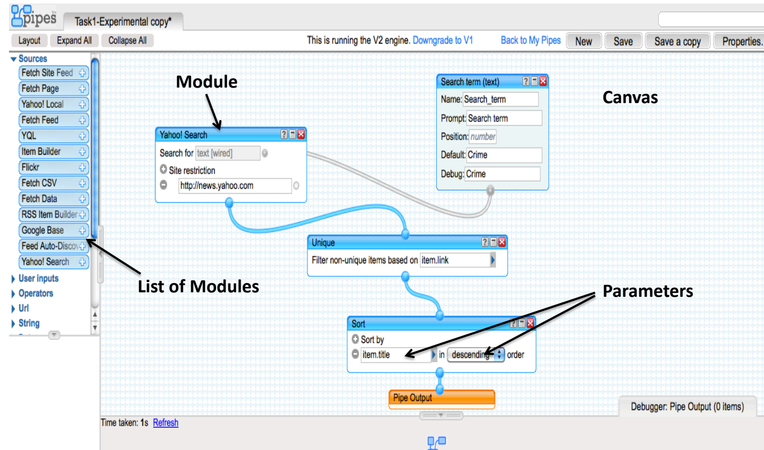

Fig. 1. Yahoo! Pipes environment with a typical pipe

and descriptions. The Yahoo! Pipes modules provide manipulation actions that can be executed on these RSS feed parameters.

Yahoo! Pipes, like other mashup environments, provides a central repository where users can store the mashups they create and use mashups created by others. A user can copy an existing pipe from the repository, modify it to suit one's need, and then publish it to the repository. Such copying of pipes is termed "cloning" and is a common development practice. However, the Yahoo! Pipes server does not allow users to keep versions or histories of these mashups, which makes understanding the development histories of these mashups difficult.

Versioning capabilities are heavily used in commercial software development and are required for a team to be successful. Versioning is largely 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 [15]. Some end user environments such as Google Docs and Google Websites provide basic versioning facilities for enabling basic group editing, but these capabilities are only for text edits. Thus far no versioning tools that we know of exist for mashups.

3 Versioning for Yahoo! Pipes

We have developed versioning capabilities for Yahoo! Pipes that allow users to save and navigate between different versions of pipes. Versions are unobtrusively and automatically generated when users save or clone their pipes. This feature is available for pipes that users create themselves as well as for other pipes that they may reuse. Our Yahoo! Pipes extension allows a user to view, edit, or run an older version of a pipe. The versioning tool is operational for most web browsers (Internet Explorer, Firefox, and Safari).

Figure 2 presents our system architecture. We use a proxy server managing communications between the client (web browser) and the Yahoo! Pipes server. Using Internet Content Adaptation Protocol (ICAP) [3], we intercept the messages sent between the Yahoo! Pipes server and a client. Our proxy wrapper

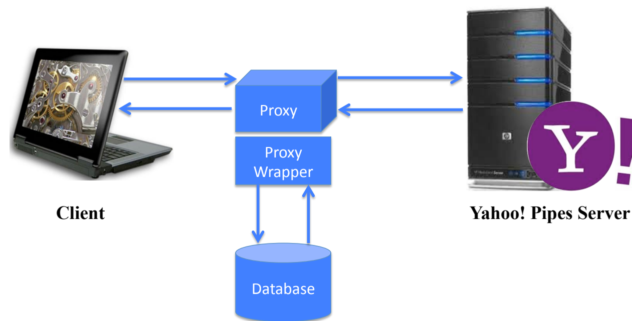

Fig. 2. Architecture of the Versioning System

selects appropriate user events (e.g., save or run) and message contents to create and store versions in a central repository (MySQL database), which serves as the versioning repository. When a user saves his or her pipe, a new version is created for that pipe (e.g., V1, V2, V3,....,Vn). Typically, version control systems allow developers to mark stable or significant versions of the development tree as baselines for easier access and retrieval [11]. When a user *runs* their pipe, we tag that version as a baseline.

The user interface of our extension adds three widgets to the Yahoo! Pipes client interface (see Figure 3). The first two are buttons that allow a user to browse different versions of a pipe via "Undo" or "Redo" operations. "Undo" renders the previous version, while "Redo" renders the next version. Finally, a drop-down list named "History of Pipes" allows a user to select a desired version from the list of available versions for the pipe. This list also displays the modules added or removed per pipe, per version, so that users can view the differences between versions. The foregoing are initial versioning capabilities that we have created. Based on user studies we will determine other appropriate versioning capabilities for end users and ways to intuitively present them.

Note that most versioning systems are text-based and operate at the level of files, whereas we version at the level of modules (Yahoo! Pipes modules). Table 1 presents a comparison of features between our versioning extension to Yahoo! Pipes and typical CM systems.

Our "history list" view provides information about the modules that are added or deleted per version, which is similar to the "diff" feature typically provided by traditional CM tools. We do not yet provide "merge" functionality whereby two or more changes in the same or different modules can be merged. To help end users better understand and navigate across versions we provide the undo and redo features, which matches the revert functionality provided by CM systems. We believe that it will be easier for end users to learn and adopt the undo-redo features to navigate across pipe versions, as these functionalities are common in most text editors. Finally, we mimic tagging of stable versions in CM systems as baselines. That is, when a pipe is *run*, we treat it as a version

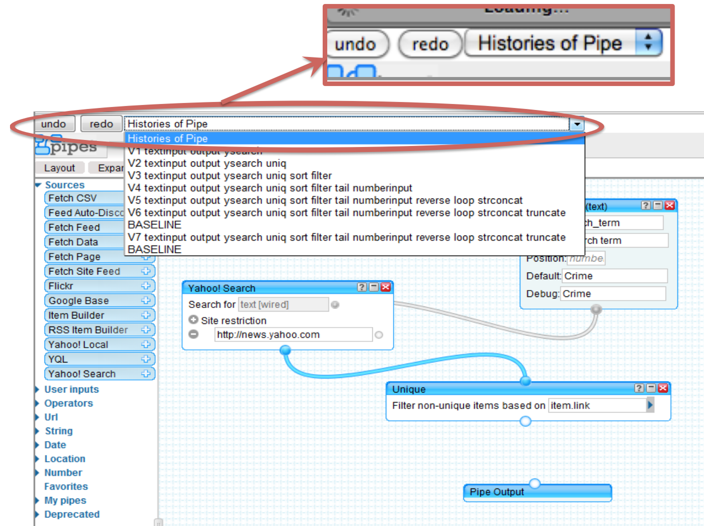

Fig. 3. Yahoo! Pipes extension interface where version V2 of a pipe is displayed. The list view (History of Pipe) shows seven existing versions and a few baseline tags.

Table 1. Comparison of the features implemented in the versioning extension for Yahoo! Pipes with existing configuration management systems

Features implemented	CM Systems	Versioning extension in Yahoo! Pipes
Versioning unit	File level	Module level
Versions created	On commit	On save/cloning of pipe
Browsing	Undo or selecting version	Undo, redo or selecting from list view
Snapshots	Baseline	Run
Diff	File level	Module level
Deltas	Add, Delete and Modify	Add and Delete
Merge	Implemented	Future work
Target population	Professional programmers	End-users

that has been completed and is in a stable form and mark it as a baseline in the history of the pipe.

4 Empirical Study

To investigate the effects of building versioning capabilities into a mashup programming environment we conducted a user study, considering the following research questions:

RQ1: Does versioning allow mashup programmers to efficiently and effectively perform tasks related to mashup creation?

RQ2: Does versioning help mashup programmers understand complex third-party mashups?

4.1 Participants

While ultimately our interest in mashups involves end users, including those who do not have experience programming in more formal languages, for this first, exploratory study, we believed it would be better to choose participants who were readily available and had some programming experience in environments other than Yahoo! Pipes. This would allow us to make sure that participants were able to quickly grasp the programming concepts in Yahoo! Pipes through tutorials that could be provided as part of a study. Another reason for selecting users with basic computer knowledge as participants was to ensure that they would be able to work with sufficiently complex pipes, since benefits of versioning are typically evident only for software artifacts that are non-trivial. Past work has also shown that complete novices to programming found coding in Yahoo! Pipes to be non-trivial and needed external help, even when pipes were three modules in size [18]. We thus focused on students who were computer science or computer engineering majors, or who at least had taken one programming class.

To recruit students 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 any prior experience with Yahoo! Pipes, but all had some programming knowledge (78% knew multiple programming languages).

4.2 Study Setup and Design

The study used a single factor, within-subjects design. (A within study design [16] is one in which the independent variable is tested with each participant.) The independent variable was our versioning extension to Yahoo! Pipes; that is, participants in the control group completed their tasks without our versioning extension and participants in the experimental group used our extension. We opted for a within-subject design for two reasons. First, we wanted to minimize the effects of individual differences among participants, which can be reduced through a within-subjects study design. Second, a within-subject study afforded us more data using a smaller sample size, which was more appropriate for our exploratory study and the use of a think-aloud protocol (see below). Finally, since each participant in the within-subjects study gained experience in performing tasks using the environment with and without our versioning extension, they were better positioned to give feedback about the usefulness and usability of our versioning extension than participants in a between-subject study.

We used think-aloud protocol in our study, where participants were asked to vocalize their thought processes and feelings as they performed their tasks [13]. We used think-aloud protocol because a primary objective of this exploratory study was to gain insights into the users' thought processes – barriers and problems that a user faced 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 user studies 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 brief 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 external web sites where they could find further help with the environment, some of which were within the Yahoo! Pipes environment and some of which were external web pages.

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 [6]). The total times required for completion of the study per participant was approximately 1.5 hours, which included on an average of 60 minutes for task completion.

After all tasks were completed, we administered a 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.

4.3 Tasks

Our analysis of large numbers of Yahoo! Pipes showed that users typically create pipes containing between five and 70 modules, and encompassing a broad spectrum of complexity. It is likely that versioning capabilities will not be useful for simple pipes; however, our belief is that they can be useful in understanding and building upon complex pipes. We therefore created tasks with levels of complexity that we believed were sufficient to investigate this belief.

Specifically, we defined two types of tasks, Task 1 and Task 2, addressing our research questions on whether versioning helps users create pipes of their own, and whether versioning helps users in understanding complex, third-party pipes. Because the study was a within-subjects study, 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. For each of these tasks, we created an appropriately sized Yahoo! Pipes example for participants to utilize.

Task 1 largely required a user to create a relatively small pipe as per a given set of requirements. However, users were given a similar existing pipe (of ten modules) as a template that they needed to first understand. A large portion of the sample pipe could be reused in implementing the pipe required by the task. The existing pipe for Task1.C allowed a Spanish-speaking person to search for reviews of any business (e.g., museum, university), within a geographical location (e.g., San-Francisco), and within a certain distance (e.g., 200 miles), the first

two being provided by the user and the last being hard coded into the pipe. The pipe user could also choose the number of reviews to view. All results returned contained the search term in the title, and were sorted in alphabetical order. Participants were asked to review the pipe until they understood its modules and linkages and then move onto the second part of the task.

The second part of Task1.C required the participant to build a pipe that used a subset (three to four) of the modules from the sample pipe that they had just reviewed, while implementing additional functionality requiring two to three other modules. In particular, the user was asked to create a pipe that allowed a search for a review of any item around any area within a certain specified distance, but required an additional functionality of changing the original titles of the search results to titles 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 specified by the user, filtered to display only those news items with the keyword "crime" 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 user was asked to create a pipe that allowed a French user to search Yahoo! News for a search term while making sure that search result titles were unique and translated into French. Note that the pipe used in this task had an associated versioning history that was accessible to the user.

Task 2 required the user 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. Note that 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.

4.4 Threats to 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 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. Given the lessons learned in this study, we can proceed to create a larger-scale controlled experiment on different classes of users. 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.

Where internal validity is concerned there are several threats to consider. The primary threat to 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. 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 times to understand pipes in the control and experimental tasks, suggesting that the pipes were of similar complexity.

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

4.5 Results and Analysis

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 in turn.

Ha1.1: Less time is required to create a pipe when using versioning than when versioning support is absent.

As participants performed each task relevant to RQ1 (Task1.C and Task1.E), we examined the total times required by participants to complete their tasks, which included both the times to understand the given sample pipe and to implement the required pipe. Figure 4 shows these times as proportions of the total time for Task1.C and Task1.E – times spent on understanding and implementing pipes. The average time spent by all participants in Task1.C was 17.19 minutes (median 14.28) while the average time spent for Task1.E was 7.22 minutes (median 6.68). 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 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.

We wished to further investigate the benefits of versioning support with respect to the subtasks of the pipe creation: namely, understanding and implementing the pipe. We did this by individually analyzing the times used by participants in these tasks (having measured each individually during the study).

Figure 5 shows the time required by participants to *understand* the pipes they were given in Task1.C and Task1.E. The average time taken by users to

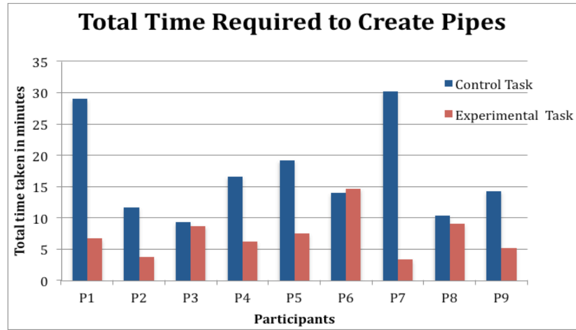

Fig. 4. Total time spent to create a pipe for a given task.

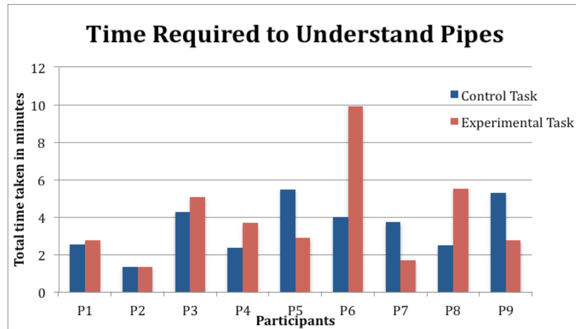

Fig. 5. Time required to understand pipes.

understand the pipe in the control task was 3.49 minutes (median 3.72), while in the experimental task it was 3.94 minutes (median 2.88). As Figure 5 shows, differences in the time required to understand the pipes varied in magnitude and across tasks. This means that versioning support did not appear to provide any significant advantage in the understanding task. We conducted the Wilcoxon Signed-rank test at a 95% confidence interval on the data, and the p-value in this case was 0.363 (with W = 15); thus we cannot assert that there were any differences in understanding the pipes during Task1.C and Task1.E.

A post-hoc analysis of participant behavior showed that they typically did not use the versioning features while engaged in the understanding task even when these were available, so it is difficult to ascribe the lack of differences in understanding time across tasks to the presence or absence of versioning.

We next turned to the *implementation* task. Figure 6 shows the time required to implement pipes for each participant. As the figure shows, for all participants the time required to implement pipes was less in the experimental task than in the control task. The average time required to create pipes in the control task was 13.70 minutes (median 10.36), while the average time required to create pipes in the experimental task was 3.27 minutes (median 3.52). A Wilcoxon Signed-rank test had W=45 with p-value = 0.002. We therefore confirmed that

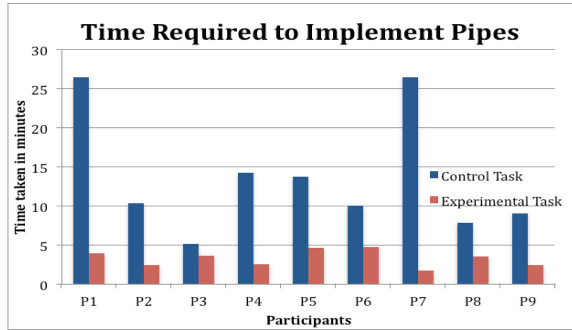

Fig. 6. Time required to implement pipes.

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.

Conceivably, one cause of the foregoing differences may be learning effects; that is, participants may have grown sufficiently familiar with the pipes during the control task to allow them to act more quickly in the experimental task. However, we noted during our observations that most of the time spent in the control task involved users examining the modules of the pipe that they had cloned to determine their applicability for the new pipe. In the experimental task, in contrast, participants were able to select earlier versions with only the modules that were required and thus save time that would otherwise be spent examining and removing the extra modules. The time differences related to the pipe implementation thus do seem to be ascribable to versioning.

Ha1.2: Pipes that are created with versioning support are more correct than those that are created without versioning.

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 and as correct as possible. Since there was no acceptance test, some of the pipes were not completely correct.

To allow us to assess correctness, prior to the experiment the first and third authors created solutions to the tasks; these served as oracles for grading. Following the study, we measured the correctness of pipes by assigning each module in the pipe a certain weight which was then aggregated. More specifically, the weight depended on the number of parameters that the module required as well as the module's existence, with each element being equally weighted. For example, the *Sort* module has two parameters and was a required component for the pipe. We would, therefore, assign 1/3 point if the module was included and 1/3 point to each correct parameter making the total points for the module to be 1. We deducted 5% from the total points given to the pipe if extra modules were included that affected the output. If a required module was completely missing, we deducted the points that would have otherwise been assigned for that module (1 point in the case of *Sort*).

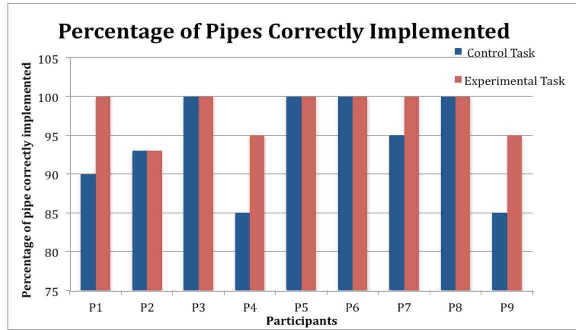

Fig. 7. Correctness of pipes created by participants.

The foregoing scoring procedure was conducted independently by the first and third authors, and then the scorers met to check the results of their grading and ensure that no grading errors had been made.

Figure 7 presents the percentages of correct pipes that were created by participants in the first task. Participants were largely successful in creating correct pipes in both the control and experimental tasks, with an average correctness for the experimental task being 98.1% and an average correctness for the control task being 94.2%. As the figure shows, four participants created completely correct pipes. Participants in the experimental tasks did exhibit a slightly higher measure for correctness of pipes than when created in the control task. We used a Wilcoxon Signed-rank test at a 95% confidence level giving us a W=0 and a p-value of 0.045; hence, we conclude that the differences in correctness between the two groups are statistically significant, confirming hypothesis Ha1.2.

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 our 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 8 summarizes the results of the questionnaire, showing that the participants differed little in terms of the numbers of correct and incorrect answers in both the control and experimental tasks. We evaluated these results through the Wilcoxon Signed-rank test at a 95% confidence interval with W equal to 14.5. Our 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.

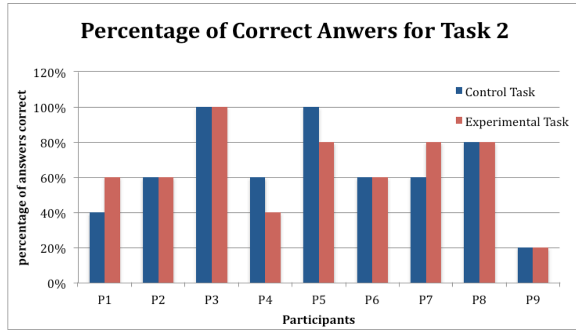

Fig. 8. The percentage of correct answers about pipes given by users in quizzes associated with Task 2.

Despite these results, we note that in the exit survey seven out of nine participants stated that they found versioning helpful for learning what a pipe did. It is possible that the Task 2 quiz result is more reflective of the difficulty in measuring a participants' ability to understand a particular pipe than of the participants use of versioning to do so.

To better understand how versioning helped or could have helped users we further investigated participant behavior in Task 1 since a significant component of the task involved understanding a pipe. We largely depended on experiment notes and the transcripts of the think-aloud sessions to do so. We found that participants, especially those performing control tasks, often desired to return to the original version of the pipe to understand it better. In some other cases participants had deleted more modules than required and wished to bring them back. In both situations a versioning tool could have been helpful.

Further, in Task 1 several participants struggled when they tried to revert to the original pipe to view it and learn how it worked. Participants in Task1.C either cloned their pipe (seven of nine) or constructed it from scratch (two of nine), but in either case they had to refer to the original pipe to look at the modules or its pipe structure. Some participants (P1, P4 and P7) viewed the original pipe by manually opening it in a separate window. With the availability of versioning in Task1.E, however, none of the participants had this trouble, because the versioning system provided access to the original pipe. In Task1.E, in fact, five of nine participants investigated the pipe structure in different versions before deciding which one to use as the base for constructing their pipe. All these participants stated in the exit survey that versioning would help improve the understanding of a pipe.

Finally, we noted that participant P6 in Task1.C also tried to go back to the original pipe, but was unable to locate it from the list of pipes from the Yahoo! page. If this pipe had a versioning tool available then he could have saved the example pipe as a version so he would be able to go back to it whenever he needed to. Note that during the experimental task this participant used versioning to learn about the pipe's structure. He commented that: *"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."

This thought was echoed by another participant, P5, who stated that versioning was helpful when trying to learn about the functionality of a pipe: *"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."*

In summary, we can infer that participants used versioning to understand the functionalities of the pipes provided to them to an extent, but we were not able to see any statistically significant difference in the time it took to understand a pipe in Task 1 or the number of correct answers in Task 2. However, our own observations and exit survey suggest that versioning can enhance the understanding of pipes.

5 Discussion

Quantitative analysis of the data confirmed that versioning capabilities can help mashup programmers complete their tasks efficiently and correctly. We now discuss additional findings from our analysis of the experiment data.

One of our initial conjectures about the potential usefulness of versioning for mashups was that it would enable mashup programmers to better reuse mashups, by making earlier versions available. Since reuse is one of the primary mechanisms by which end-user programmers construct new programs [10], data on this conjecture might further inform the mechanisms by which we provide versioning. Our exploratory study indicates that versioning can facilitate the reuse of pipes and their sub-parts. Our results divided the total time taken to create a pipe into two subsets: (1) the time taken to understand the given example pipe and (2) the time taken to implement the required pipe. We found no statistical benefits of versioning for the first of these subsets, but we found that users were quicker in implementing their pipes when using versioning capabilities. Further, they produced more correct pipes. We observed that the benefits of versioning occurred largely because participants used the "versioning history" functionality and were able to correctly use the modules from the example pipe. Our results show that versioning can indeed facilitate the reuse of pipes and their sub-parts.

This benefit of versioning was suggested in the feedback from our exit surveys where we asked participants about their thoughts on whether versioning improved/can improve the reusability of pipes. Eight of nine participants mentioned that they thought versioning did help improve the reusability of pipes and all eight of them had used versioning in their experimental tasks. In fact, one participant commented on the benefits of understanding the thought process of the creator of the pipe as evidenced through the versioning history, which helps in the understanding of the pipe. Participant P4 commented: *"You can see how each pipe was developed along the way to final product."*

Our qualitative analysis of the transcripts of the think-aloud sessions helped us investigate the different types of problems that participants faced when performing their tasks. Here we list some of the instances of typical problems faced

by users where versioning could have been helpful. For example, one of the problems was in locating the original pipe (when a pipe was created from scratch) or the parent pipe (when a pipe was cloned for the task) for reference during the tasks. Recall that participant P6 was unable to locate the original pipe, which left him frustrated. Similarly, participants who cloned the original pipe often removed too many modules or did not remember how the modules that they had removed functioned in the original pipe. As an example, participant P7 faced this problem and was frustrated when given the assignment of replacing all of the item titles (Task1.C) with a user input. He was not sure whether a *loop* was needed and he ended up removing the loop and adding it back to the canvas at least three times. Currently, there is no mechanism for identifying the parent pipe in the Yahoo! Pipes environment. Moreover, one can only recognize that the current pipe has been cloned, but there is no way of identifying which modules belonged to the original version and which are new additions.

Versioning capabilities would have helped in both these kinds of problems. In the first case, versioning would have allowed the users to easily identify the original (base) version and build from there. In the second case, our versioning features of "undo" and "redo" would have helped participant P7 to easily explore the functionality of the *loop* module.

We found that versioning histories as provided by our list view were frequently used by the participants when they edited pipes in the experimental tasks. The list view enabled the user to directly select a version of the pipe that they desired to modify. As discussed earlier, participants in Task1.E used the list view to select the version of interest, which they then used to build upon. 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."*

We were surprised to note, however, that participants did not really use versioning when they were performing the "understanding" activities or for investigating the effects of changes by going backward ("undo") or forward ("redo"). Our initial assumption was that versioning would be helpful in understanding changes – the development process of another user as visible through the versioning histories, or the participant's own development process through undo and redo. None of the participants used these features. This might be because these were novel but unfamiliar functionalities, which users were uncomfortable in using. We plan to explore different kinds of visual metaphors that make such functionality more intuitive and usable to end users.

6 Conclusion

We have presented a versioning tool for use on mashups, and provided insights into the potential usefulness of that tool by conducting a think-aloud study. The results of the study confirmed that the versioning tool can help mashup programmers create mashups more efficiently and effectively. We also see indications that it can improve reusability of code.

In our future work, we plan to implement a more robust "diff" functionality in our versioning extension for Yahoo! Pipes. Currently, the "diff" of pipes is at

the module level; that is, it identifies modules that have been added or removed. We also plan to fine-tune our analysis to identify modules that have changed, specifically tracking changes to the parameters in the module and connections across modules. We will also consider providing a side-by-side comparison of two versions of a pipe. Such an interface that also highlights the changed modules should allow users to very quickly identify how a pipe has changed. Finally, we will expand the scope of our studies of versioning to include additional groups of users, and in particular, end users.

Acknowledgments

This work was supported in part by the AFOSR through award FA9550-09-1-0129 and by the NSF through award NSF CCF-0747009 to the University of Nebraska - Lincoln.

References

1. http://pipes.yahoo.com/pipes/
2. http://www-01.ibm.com/software/info/mashup-center/
3. http://www.icap-forum.org/
4. http://www.jackbe.com/
5. http://www.open.collab.net/news/press/2007/svn_momentum.html
6. http://www.techsmith.com/morae.asp
7. http://www.xfruits.com/
8. Cao, J., Rector, K., Park, T.H., Fleming, S.D., Burnett, M., Wiedenbeck, S.: A debugging perspective on end-user mashup programming. In: VLHCC. pp. 149–156 (Sep 2010)
9. Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M., Grigoreanu, V.: End-user mashup programming: Through the design lens. In: CHFCS. pp. 1009–1018 (Apr 2010)
10. Cypher, A., Dontcheva, M., Lau, T., Nichols, J.: No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann (2010)
11. Estublier, J., Leblang, D., Hoek, A.v.d., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management. TOSEM 14, 383–430 (Oct 2005)
12. Jones, M., Churchill, E.: Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In: CCT. pp. 51–60 (Jun 2009)
13. Lewis, C.H.: Using the "thinking aloud" method in cognitive interface design. RC 9265, IBM (1982)
14. Scaffidi, C., Shaw, M., Myers, B.: Estimating the numbers of end users and end user programmers. In: VLHCC. pp. 207–214 (Sep 2005)
15. Tichy, W.F.: RCS-A system for version control. In: SPE. pp. 637–654 (1985)
16. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M., Regnell, B., Wessln, A.: Experimentation in Software Engineering: An Introduction. Springer (2000)
17. Zang, N., Rosson, M.: Whats in a mashup? And why? Studying the perceptions of web-active end users. In: VLHCC. pp. 31–38 (Sep 2008)
18. Zang, N., Rosson, M.: Playing with information: How end users think about and integrate dynamic data . In: VLHCC. pp. 85–92 (Sep 2009)