History Repeats Itself More Easily When You Log It: Versioning for Mashups

Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel
University of Nebraska–Lincoln
{skuttal, asarma, grother}@cse.unl.edu

***Abstract*—Web mashup environments provide a way for users to combine data from web applications and services to create new content. Currently, these environments do not provide support for tracking the development histories of mashups. We have thus added configuration management support to the Yahoo! Pipes mashup environment. We describe this support, and provide results of an experiment studying the ability of programmers to create and debug mashups in its presence. Our results show that versioning support can help both groups of users do both tasks better.**

I. Introduction

Web mashups allow end-user and professional programmers to obtain data from various web sites and services, and perform various operations on and combine that data to achieve various goals. Mashup environments such as Yahoo! Pipes [9] and IBM Mashup Maker [5] help them perform these tasks. While mashup programming is considered to be potentially useful, creating mashups still poses difficulties [10], [11], because the application programming interfaces, data structures and programming models involved can quickly become unmanageable [6]. Debugging mashups is also challenging: in one study, end users spent 76.3% of their time on this task alone [1], [2].

Mashup programming environments provide repositories that support the reuse and development of mashups. Current repositories, however, do not provide information on how mashups have been developed over time, as users clone, incrementally create, and enhance them. Professional developers rely on configuration management systems to keep prior versions of systems and store information about changes [4]. Versioning information indicates the process by which systems evolve, helping developers better understand them. It also helps developers track cases in which faults have been introduced and correct them.

Motivated by these observations, we have been investigating whether versioning features can be used by mashup programmers. In prior work [7], we described an approach for supporting versioning in the Yahoo! Pipes environment. Our approach involved adding basic configuration management to the environment with a simple interface. An exploratory study showed that our versioning support helped them create mashups more efficiently.

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

II. Yahoo! Pipes

Yahoo! Pipes [9] is a semi-automatic mashup environment. 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 1 shows the interface of the Yahoo! Pipes environment with various components of the interface. Since its launch in 2007, nearly 90,000 developers have created mashups using this environment, and an estimated 5,000,000 pipe runs occur each day [6].

III. Versioning for Yahoo! Pipes

We have provided versioning support within the Yahoo! Pipes environment [7]. Whenever a pipe programmer saves or clones a pipe, our system automatically generates a new version for the pipe, with version numbers created in chronological order (V1, V2,....,Vn). When a pipe programmer wishes to retrieve a prior version our system allows them to do so; they can then view, edit, and run a version.

To help pipe programmers work with versions we provided an initial interface including "undo" and "redo" but-

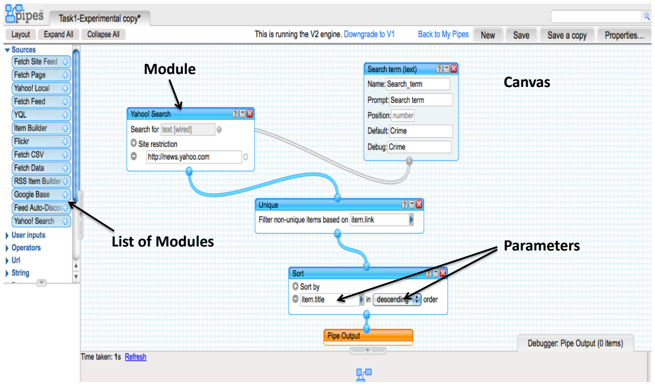
This is running the V2 engine. Downgrade to V1
Search for text [wired]
Site restriction http://news.yahoo.com
Name: Search_term
Prompt: Search term
Position: number
Default: Crime
Debug: Crime
Filter non-unique items based on item.link
Sort by item.title in descending order
Time taken: 1s Refresh
Debugger: Pipe Output (0 items)

Figure 1. Yahoo! Pipes interface

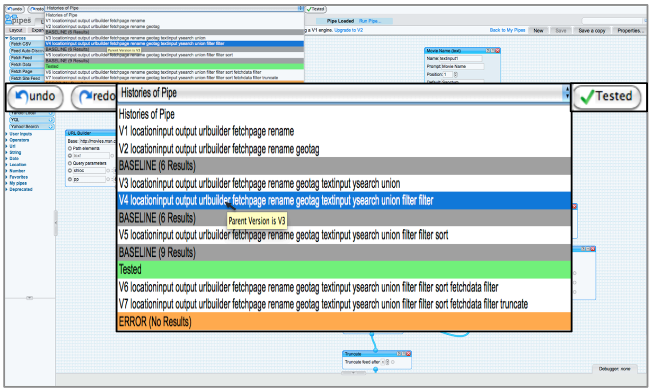
undo
redo
Histories of Pipe
Tested
Histories of Pipe
V1 locationinput output urlbuilder fetchpage rename
V2 locationinput output urlbuilder fetchpage rename geotag
BASELINE (6 Results)
V3 locationinput output urlbuilder fetchpage rename geotag textinput ysearch union
V4 locationinput output urlbuilder fetchpage rename geotag textinput ysearch union filter filter
BASELINE (6 Results)
Parent Version is V3
V5 locationinput output urlbuilder fetchpage rename geotag textinput ysearch union filter filter sort
BASELINE (9 Results)
Tested
V6 locationinput output urlbuilder fetchpage rename geotag textinput ysearch union filter filter sort fetchdata filter
V7 locationinput output urlbuilder fetchpage rename geotag textinput ysearch union filter filter sort fetchdata filter truncate
ERROR (No Results)

Figure 2. Pipes Plumber interface

tons that allow them to progress forward and backward through a sequence of versions, and a list view showing the linear development tree for versions and modules, in the sequence they were added and removed within each version. This lets programmers view older versions, and also helps them see which modules have been added before and after other modules. When a pipe is run we treat it as a "baseline" to indicate that it is a version in stable form. Implementation details can be found in [7].

We have further enhanced our versioning support, which we now refer to as "Pipes Plumber". Figure 2 shows the Pipes Plumber interface, which provides widgets for undo, redo, and history display as in our prior version (in the screenshot the latter is expanded to show "Histories of Pipe" information.)

Pipes Plumber expands on this support in three ways. First, the pipe history information lists 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, currently we display information on version parentage via tooltips that appear when the pipe programmer hovers over a given version (see Figure 2).

As a second enhancement, which we believe can be utilized in debugging as well as in assessing the status of versions of other pipes, we utilize color coding in the history information to provide insights into the run status of versions. Versions in grey are those that were successfully executed (in Figure 2, the menu item labeled "BASELINE"), which signifies potential correctness of the pipe since it returns non-null results. Orange (item labeled "ERROR (No Results)") signifies that a pipe execution returned no results, a probable indication of an error. Finally, green (items labeled "Tested") indicates that the pipe programmer has confidence that a given version is operating correctly, a status that can be applied to the version through the third enhancement to the interface, a "Tested" button.

IV. Empirical Study

We evaluated whether "Pipes Plumber" can help mashup programmers create and debug mashups. Our research questions were as follows:

- **RQ1:** Does versioning help mashup programmers create pipes more effectively and efficiently?
- **RQ2:** Does versioning help mashup programmers debug pipes more effectively and efficiently?

A. *Participants*

We recruited 24 students from the University of Nebraska who were compensated $20 for their participation. Twelve of the participants were from the Computer Science department, and the rest were from other departments (Science and Engineering) and had no formal training or experience in programming. Participants' ages ranged from 19 to 40 years; 22 were male and two were female.

B. *Study Setup and Design*

Our study employed a single-factor, within-subjects design, in which participants conducted both pipe creation and debugging tasks. The independent variable involved the presence or absence of versioning information. 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.

In each session, we administered a background questionnaire and a ten minute tutorial on Yahoo! Pipes. We then asked the participant to create a small sample pipe. Next, we asked the participant to complete specific tasks required for the study. We audio recorded sessions and logged participant's on-screen interactions. The total time required for completion of the study was approximately two hours, which included around 60 minutes for actual task performance. At the end we administered an exit survey.

C. *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.[1]

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 pipe. In *Task1.search*, participants were given a pipe that let Korean users search for a review of any item within a given distance from a given location with a date on which the review was published. The next step required the participant to create a pipe similar to the above; this required participants to modify the pipe by implementing extra functionality

[1] For detailed descriptions of the tasks see http://www.cse.unl.edu/∼skuttal/Tasksonline.pdf.

and removing some functionality. *Task1.blog* was similar in scope to *Task1.search* and involved a similar pipe.

In Task2, participants were given pipes containing two seeded faults. Participants were required to correct the faults and ensure the pipe worked as stated in the requirements by comparing their results to the sample output. *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. Participants could also get a poster and reviews of a movie. *Task2.eBay* involved a pipe that allows a user to search for an item within a price range on an auction site (e.g., eBay, Craigslist). Search results were limited by user and include the name of the site from which the item was retrieved.

D. Measures

To evaluate the use of versioning, we measured the times required to complete tasks and the quality of the resulting pipes. The quality of pipes was determined by grading the resultant pipes through a correctness score ranging from 0 to 100. The first author and an undergraduate student not involved in the work created a grading scheme for the pipes. They used this to grade pipes individually, and then they came to consensus on the grading results.

On *Task1* 40 points were attributed to the correct use of the modules or the functionality provided to participants. The remaining 60 points were awarded if the participant could correctly create the additional modules needed to complete the task. On *Task2*, 80 points were assigned if participants successfully identified and corrected the seeded errors in the given pipe and remaining 20 points were allocated to other errors introduced by user.

E. Threats to Validity

Our choice of a within-subject design could lead to learning effects, since participants perform two tasks (one control, one treatment). We counterbalanced our tasks (experimental and control) to reduce this threat. In our post-hoc analysis, we did not find evidence of bias or consistent differences correlated with task order. There is a potential for learning effects in the debugging tasks, but this effect would be consistent across the treatment groups and would not affect our results. A second validity threat can arise if our pairs of subtasks (two per task type) are not of equal complexity. To limit this threat, we also interchanged the subtasks that were given to the control and experimental groups.

Our participants were all university students, and our "end-user" population was primarily engineering students. All but two participants were male. Participants were asked to use pipes that were provided to them, rather than pipes which they had created for themselves. While the reuse context is important, prior familiarity with pipes could lead to different results.

Table I
PIPE CREATION TASK: RESULTS SUMMARY

Dependent variable	P-value	Treatment variable	CSE (median)	EU (median)
Correctness	0.002	Control	78.0%	67.5%
		Experiment	87.5%	93.0%
Time (minutes)	0.073	Control	12.8	15.5
		Experiment	11.1	12.9

F. Results and Analysis

Tables I and II summarize our experiment results.

Research Question 1. We frame our first research question 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 are more correct.

For both participant groups (CSE denoting Computer Science and Engineering students, and EU denoting the other ("end-user") participants), median correctness scores were higher in the experimental task (when versioning support was used) than in the control task (no versioning support), and variation in scores was lower, with the results more pronounced for EU participants. The median correctness scores for CSE and EU participants were 78.0% and 67.5% in the Control group and 87.5% and 93.0% in the Experimental group, respectively.

An analysis of interaction [3] showed that there were interaction effects between the EU and CSE participant groups. We therefore blocked over participant group when performing our statistical analysis. A SPANOVA [3] on the results, performed using the R programming language [8], shows highly statistically significant differences between the treatments (F=12.518 and p=0.002), indicating that versioning helped pipe programmers create more correct pipes.

Ha1.2: Pipes can be created more quickly with versioning.

For both participant groups, median time costs are lower with versioning support, and variation in time is lower, with the variation more pronounced for the CSE participants. The median time cost for CSE and EU participants was 12.8 and 15.5 minutes in the Control group and 11.1 and 12.9 minutes in the Experimental group, respectively.[2]

We block over participant group when performing our statistical analysis because of interaction effects. A SPANOVA on the results shows marginally significant differences between the treatments (F=3.569 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)

[2] Two EU participants did not save their pipes after they had finished their implementations, lost their changes, and had to recreate them. We exclude those two participants from our sample where time costs are concerned.

Table II
PIPE DEBUGGING TASK: RESULTS SUMMARY

Dependent variable	P-value	Treatment variable	CSE (median)	EU (median)
Correctness	0.004	Control	60.0%	60.0%
		Experiment	100.0%	67.5%
Time (minutes)	0.104	Control	17.6	18.1
		Experiment	10.4	17.6

on the abilities of pipe programmers to debug pipes, in terms of time cost and correctness outcome.

Ha2.1: Versioning helps pipe programmers debug pipes more correctly.

For both participant groups, median correctness scores are higher with versioning support. Median correctness score for CSE and EU participants was 60.0% in the Control group; whereas in the Experimental group scores for CSE and EU participants were 100.0% and 67.5%, respectively.

Due to a presence of interactions, we block over participant group when performing the SPANOVA. Results are statistically significant (F=10.189 and p=0.004) indicating that versioning did help pipe programmers in debugging with respect to the correctness of their output.

Ha2.2: Versioning helps pipe programmers debug pipes more quickly.

Here, results differ across participant groups. The median time cost for CSE participants was 17.6 minutes and 10.4 minutes for the Control and Experimental groups. The median time cost for EU participants was 18.1 minutes and 17.6 minutes for the Control and Experimental groups. In this case, a SPANOVA did not reveal statistically significant differences between the treatments (e.g. F=2.873 and p= 0.1036), and we cannot support our hypothesis.

Additional Observations. In the pipe creation task, both CSE and EU participants were able to produce pipes of comparable quality when using versioning. The correctness "boost" obtained by CSE participants was less than that obtained by EU participants, in part because a much larger number of EU participants had relatively low correctness scores when creating pipes without the versioning support. EU participants also required somewhat more time than CSE participants on the creation task. We suspect that this difference reflects the EU participants' lesser knowledge of versioning concepts, which would lead them to require more time on the creation tasks, which were newer to them conceptually than to CSE participants.

In the pipe debugging task, again both participant groups experienced increases in median correctness values, but CSE participants experienced greater benefit increases overall, likely due to their greater experience with debugging in general. The lack of significant differences in debugging task times, however, could also in part be due to learning effects since both groups performed the debugging task second.

V. SUMMARY AND CONCLUSION

We have added versioning support for mashup programmers to Yahoo! Pipes. Our results studying the use of that support in mashup creation and debugging shows that it helps both end users and CSE students create more correct pipes; however, we did not see similar benefits in the time it took participants to complete tasks. This is not unexpected given that our participants needed to master a new programming paradigm (pipe based programming) and new technical interfaces (for Yahoo! Pipes and versioning support). However, we believe the success of our versioning support is evident in the correctness of the pipes that were generated and the participants' positive feedback.

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. We also intend to conduct additional studies of our versioning approach in larger-scale scenarios.

Acknowledgments

This work is partially supported byAFOSR FA9550-09-1-0129. We thank Dr. Kathy Hanford for providing statistical expertise and Branden Barber for helping in the experiments.

REFERENCES

[1] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A debugging perspective on end-user mashup programming. In *VLHCC*, pages 149–156, Sept. 2010.

[2] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. End-user mashup programming: Through the design lens. In *CHI*, pages 1009–1018, Apr. 2010.

[3] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for Research, 3rd Edition*. Wiley, 2004.

[4] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *TOSEM*, 14:383–430, Oct. 2005.

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

[6] M. Jones and E. Churchill. Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In *CCT*, pages 51–60, June 2009.

[7] S. K. Kuttal, A. Sarma, A. Swearngin, and G. Rothermel. Versioning for mashups - an exploratory study. In *IS-EUD*, pages 25–41, June 2011.

[8] P. Teetor. *R Cookbook*. O'Reilly, first edition, 2011. ISBN: 978-0-596-80915-7.

[9] Yahoo! pipes. http://pipes.yahoo.com/pipes/.

[10] N. Zang and M. Rosson. What's in a mashup? And why? Studying the perceptions of web-active end users. In *VLHCC*, pages 31–38, Sept. 2008.

[11] N. Zang and M. Rosson. Playing with information: How end users think about and integrate dynamic data. In *VLHCC*, pages 85–92, Sept. 2009.