Debugging Support for End-User Mashup Programming

Sandeep Kaur Kuttal, Anita Sarma, Gregg Rothermel
University of Nebraska - Lincoln
(skuttal, asarma, grother)@cse.unl.edu

ABSTRACT
Programming for the web can be an intimidating task, particularly for non-professional ("end-user") programmers. Mashup programming environments attempt to remedy this by providing support for such programming. It is well known, however, that mashup programmers create applications that contain bugs. Furthermore, mashup programmers learn from examples and reuse other mashups, which causes bugs to propagate to other mashups. In this paper we classify the bugs that occur in a large corpus of Yahoo! Pipes mashups. We describe support we have implemented in the Yahoo! Pipes environment to provide automatic error detection techniques that help mashup programmers localize and correct these bugs. We present the results of a think-aloud study comparing the experiences of end-user mashup programmers using and not using our support. Our results show that our debugging enhancements do help these programmers localize and correct bugs more effectively and efficiently.

Author Keywords
End-user programming; end-user software engineering; mashups; Yahoo! Pipes; debugging; programming barriers

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces– Evaluation/methodology

General Terms
Human Factors

INTRODUCTION

Web mashups are situational applications that allow their users to scrape information from the web. Professional and non-professional ("end-user") programmers create these applications by combining web services and processing their outputs in various ways. Examples of such applications include those for alerting drivers to the presence of traffic jams, tracking flights, finding apartments for rent in a given location, and so forth. Mashup programming environments such as Yahoo! Pipes [33], IBM mashup maker [12], xfruit [32], Apatar [2], Deri pipes [8], and JackBe [13] provide facilities for constructing such applications.

Like any programming activity, mashup programming can be difficult, and mashup programmers can create applications that contain bugs. Mashups interact with the web, and the web is a complex ecosystem of heterogeneous formats, services, protocols, standards and languages [7] all of which tend to evolve. The dependence of mashups on this complex ecosystem makes them vulnerable to unexpected behaviors. A further complication involves the interfaces by which mashups are created: mashup environments facilitate mashup creation by providing visual interfaces, which abstract the underlying code as black box features. While this black box structure allows users to "cobble together" solutions from existing resources, it can obscure the sources of bugs and hide distinctions between different types of bugs [15] – an understanding of which is important for debugging. These problems are exacerbated by the typical programming practices used to create mashup, which involve learning from examples and reuse of other mashups [30] – both of which can propagate bugs to other mashups.

To better understand the bugs found in mashups, we studied a large corpus (25,734) of Yahoo! Pipes mashups. We found that more than 64.1% of these pipes contained bugs. Moreover, 56.3% of the pipes were "cloned" (copied from existing pipes) and 27.4% of the pipes contained at least one sub-pipe (a pipe present separately in the repository). Clearly, the prevalence of bugs in pipes and the tendency for users to reuse pipes create problems for mashup dependability.

One implication of this data is that users frequently debug mashups. Debugging mashups, however, intrinsically involves distance and visibility issues due to distributed and black box dependencies, and this renders it time and effort intensive. In a study in which end users designed and created mashups, they spent 76.3% of their time in debugging [4]. We anticipate that the time required to debug mashups that inherit faults through reuse will be just as substantial.

The difficulties of debugging mashups are exacerbated by the debugging support available in mashup programming environments, which is limited to console output messages [10]. In general, mashup debugging activities require mashup programmers themselves to localize and fix bugs [7]. Debugging techniques such as static or dynamic debugging and source code manipulation are not available in these environments. This hinders developers from understanding when a particular piece of code is executed and in what context. Further, mashups are constrained by API boundaries and the reliance on external sources, which continuously evolve. As a result, run-time observation of program behavior is the primary approach used for debugging mashups [7].

Runtime observations are not always sufficient in debugging. Our prior work has shown that understanding barriers (difficulty in correctly comprehending the feedback) pose the most significant problem faced by mashup programmers when debugging [22]; these barriers occur when externally visible behavior obscures what a mashup does (or does not do) at runtime. Often runtime observations generate inadequate error messages, which raises the understanding barrier for end-user programmers. When mashup creators cannot understand error messages, fixing bugs becomes difficult.

After identifying the cause of a bug, to correct it a mashup programmer must understand the usage of the program elements (e.g., modules) involved. Past studies have found that end users struggle with this (due to use barriers [18]) in tasks such as selecting the appropriate modules and comprehending their usage [22]. Therefore, to facilitate debugging for end users there is a need for approaches that integrate various strategies for overcoming understanding and use barriers.

To investigate the issues involved in debugging mashups, and devise approaches that make the task easier for mashup programmers, we have implemented enhanced support for debugging in the Yahoo! Pipes environment. We provide techniques for detecting the presence of bugs in pipes, and feedback to mashup programmers that can help them locate and correct bugs more efficiently and effectively. In this paper, we describe our debugging support, and present the results of an empirical study investigating the impact of our debugging support on the experiences of end-user programmers. Our study shows that our debugging enhancements do help mashup programmers localize and correct bugs effectively and efficiently.

This work makes the several contributions, including contributions that generalize beyond the class of pipes considered in this particular paper, as follows:

- We identify specific classes of faults found in Yahoo! Pipes programs that can be used directly, or as a starting point, for identifying fault classes in other end-user programming domains, including web-development and visual programming paradigms.
- We provide empirical results about the classes of faults that arise because of program nesting, silent failures of programs, and program reuse and their effects on debugging by end users; these too have implications for attempts to apply our approach to other domains.
- We provide guidelines about the implications of the above effects on design and debugging support in end-user programming domains that generalize beyond Yahoo! Pipes to other programming environments.
- Finally, we illustrate an overall research approach that can be applied in other programming environments to help end-user programmers. This approach involves a methodology for identifying fault classes, defining and implementing detectors for faults in those classes, creating appropriate messages about detected faults and providing instructions for fixing those faults.

RELATED WORK ON DEBUGGING FOR END USERS

Most end-user programming environments support debugging only through consoles that print debugging output [10]. A few improvements, however, have been proposed.

“What You See Is What You Test" (WYSIWYT) [26] supplements the conventions by which spreadsheets provide visual feedback about values by adding feedback about “testedness”. The WYSIWYT methodology can also be useful for other visual programming languages [16] and for debugging of machine-learned classifiers by end users [20]. Our approach is similar to WYSIWYT, while also providing a “To-fix" list of errors and guidance for fixing them.

Topes [27] is a data model that helps end users validate and manipulate data in spreadsheets. A similar strategy could be used to identify mismatches in data formats. Whyline [17] allows programmers to ask “why did” and “why didn’t” questions about program output in visual programming languages such as Alice, as well as in Java. It employs both static and dynamic analyses of programs to provide answers when a programmer selects a question.

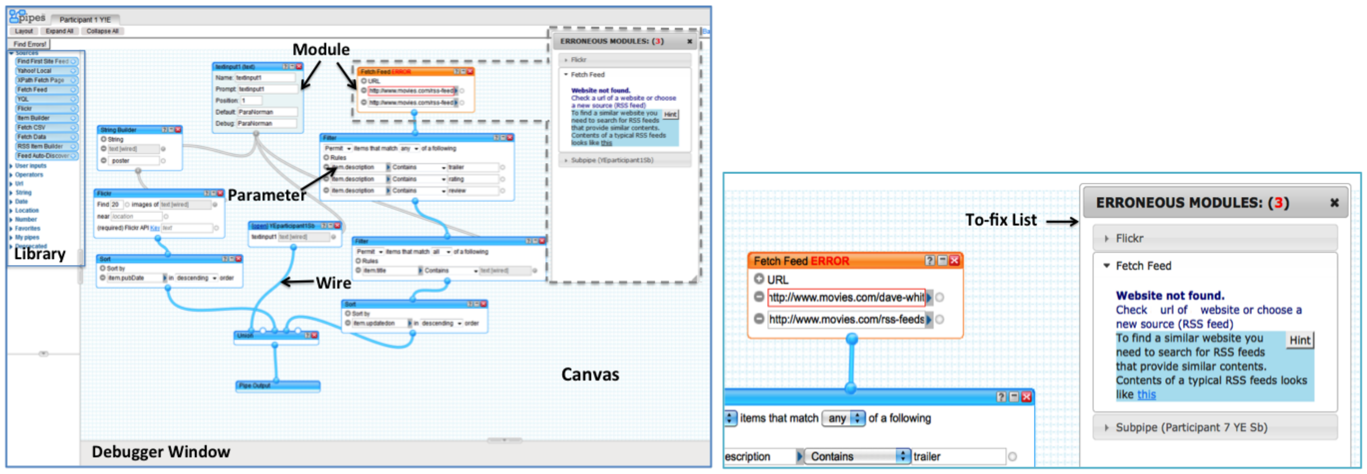
Assertions have been used to find faults in web macros [19] whose creation involves copying and pasting techniques, i.e., once data is entered by the user it is saved in the clipboard and before the data is reused it is tested for existence and datatypes. We use assertions but do not depend on data saved on clipboards; instead, our checks are based on anomaly classifications (discussed later).

Most mashup programming environments support debugging by providing a debugger console for the program. Yahoo! Pipes is the only such environment that provides a debugger console for every module. The only other debugging support for mashup environments that we are aware of is from our prior work [21], which allows mashup programmers to “tag” faulty and tested mashups. Faults can then be localized by “differencing” faulty and correct versions of a pipe.

YAHOO! PIPES

Yahoo! Pipes is a web-based visual programming environment introduced in 2007 to enable users to “rewire the web” [33]. Yahoo! Pipes [14] is arguably the most popular mashup programming environment, and is being used by professional and end-user programmers. As a visual programming environment, Yahoo! Pipes is well suited to representing solutions to dataflow based processing problems.

Yahoo! Pipes “programs” combine simple commands together in the form of modules. The Yahoo! Pipes engine also facilitates the wiring of modules together and the transfer of data between them. Figure 1(a) shows the Yahoo! Pipes programming environment interface with our debugging support extension in the upper-right corner. (Figure 1(b) presents a closeup of this extension and is described later). The Yahoo! Pipes programming environment consists of three major components: the canvas, library, and debugger. The library is located to the left of the pipe editor and consists of a list of modules categorized according to their functionality. The canvas is the central area in which users create their pipes by placing modules on the canvas and connecting them together


Module
Parameter
Wire
Library
Canvas
Debugger Window
To-fix List
ERRONEOUS MODULES: (3)
Flickr
Fetch Feed
Subpipe (Participant 7 YE Sb)
Fetch Feed ERROR
URL
http://www.movies.com/dave-whit
http://www.movies.com/rss-feeds
items that match any of a following
escription Contains trailer
Hint

Website not found.
Check url of website or choose a new source (RSS feed)
To find a similar website you need to search for RSS feeds that provide similar contents. Contents of a typical RSS feeds looks like this

(a) The Yahoo! Pipes interface overview. (b) Close up view of our debugging support extension.

Figure 1. Yahoo! Pipes with debugging support

with wires. The debugger helps users see the output from specific modules as well as the final output of the pipe.

Inputs to pipes can be HTML, XML, JSON, KML, RSS feeds and other formats, and outputs can be RSS, JSON, KML and other formats. Inputs and outputs between modules are primarily RSS feed items. RSS feeds consist of item parameters and descriptions. Yahoo! Pipes modules provide manipulation actions that can be executed on these RSS feed parameters. Yahoo! Pipes also allows users to define various datatypes such as url, text, number, and date-time.

A CLASSIFICATION OF BUG TYPES PRESENT IN PIPES

To better understand the types of bugs found in Yahoo! Pipes programs, we studied pipes extracted from the Yahoo! Pipes repository and created a classification scheme (Table 1). Column 1 of the table lists the types of bugs that we found. Broadly speaking, we categorize bugs as “intra-module” or “inter-module”. Intra-module bugs occur within individual Yahoo! Pipes modules, while “inter-module” bugs are related to interactions between modules.

Intra-module Bugs

Intra-module bugs involve links, missing expected content or parameters, problems with mismatched values, and syntax.

Link bugs occur when a web page referenced in a module cannot be found, such as when pages no longer exist, or when page access restrictions or server errors occur. Yahoo! Pipes detects these bugs, but the error message displayed is cryptic, typically of the form *“Error fetching [url]. Response forbidden (403)”*.

Missing bugs occur when the contents of web pages utilized by a pipe no longer include expected elements, or the values of parameters used in modules are missing. A typical *Content* bug involves the absence, in a fetched web page, of specific required source, such as delimiters that precede data being searched for. A typical *Parameter* bug occurs when a user omits a parameter value. Yahoo! Pipes provides no information regarding these bugs – pipes that contain them simply fail, or worse, provide incorrect results with no indication that a problem has occurred.

Mismatch bugs involve errors in the format of data contained in parameters, or ranges of values used in checks supported by Yahoo! Pipes modules. A typical *Format* bug can result from a user expressing data in a non-supported format (e.g., MM.DD.YYYY). A typical *Range* bug can result from a user entering an inappropriate range in a data checking context. Yahoo! Pipes provides no information regarding these bugs.

Syntax bugs are specific to particular modules, and occur when incorrect syntax has been used for operators or parameters. For example, the `Filter` module lets users filter web contents by defining filtering rules. If the user wishes to filter on titles that include a specific name, they can do this by defining a rule of the form <parameter1><criteria><parameter2>, (for example, <title><contains><Obama>). In such cases, selecting incompatible parameters can lead to problems; for example, <title><less than><Obama> is incorrect because the “less than” criterion requires both parameters to be of type “number”. Yahoo! Pipes provides no information regarding these bugs.

Inter-module Bugs

Inter-module bugs involve modules, connectors, and errors in data types.

Module bugs occur when pipe components themselves are missing, where components are deprecated modules, submodules, or sub-pipes. A typical *Deprecated Module* bug occurs when the module used in a pipe is no longer entirely supported by the environment. A typical *Sub-Module* bug can occur when a sub-module required for pipe functionality is not present in a pipe; for example, when a programmer fails to specify a submodule for inclusion in a `loop` module. A typical *Sub-Pipe* bug can occur when a sub-pipe employed by a programmer at an earlier point in time no longer exists. Yahoo! Pipes provides error messages for deprecated modules, but not for other module bugs.

Connector bugs occur when wires required for correct computation are missing, an extreme case of which involves the

Table 1. Classification of Yahoo! Pipes Bug Types

Bug Type	Details	Detection Mechanism	Freq. in Pipes (across 20200)	Freq. in Sub-pipes (5534)	Yahoo! Errors
INTRA-MODULE					
Link	Web page no longer exists, page access restriction or server error	Website check	6386	1572	Yes
Missing					
Content	Web page contents missing	Static analysis of code, web page	2289	182	No
Parameter	Parameter value missing	Static analysis of code	12233	3446	No
Mismatch					
Format	Data in non-supported format	Regular expressions or Topes	NA	NA	No
Range	Inappropriate range of numbers	Assertions	12	4	No
Syntax	Syntax for operators/parameters should be followed correctly	Static analysis of code	3651	1070	No
INTER-MODULE					
Module					
Deprecated Module	Module no longer supported	Static analysis of code	4637	1308	Yes
Sub-Module	Sub-module inside a module is not present	Static analysis of code	1	0	No
Sub-pipes	Sub-pipe of pipe does not exist	Database check	2	2	No
Connector	Wire missing between modules	Static analysis of code	1369	493	No
Data-Type	Incompatible modules attached to each other	Topes	1639	712	No

presence of an unconnected ("orphan") module [9]. Yahoo! Pipes provides no information regarding these bugs.

DataType bugs occur when type compatibilities exist between data elements being passed between modules. Yahoo! Pipes provides no information about these bugs.

AN APPROACH FOR DETECTING YAHOO! PIPES BUGS

Figure 2 illustrates the support architecture by which we enhance Yahoo! Pipes so that it automatically detects bugs. A proxy server (Squid 3.1.4[1]) manages communications between the client (web browser) and the Yahoo! Pipes server. Using the Internet Content Adaptation Protocol (ICAP[2]), a proxy wrapper intercepts request and response messages exchanged between a client and the server; among these it intercepts user events (e.g., requests to save or run a pipe) and message contents and stores them in a MySQL database. When a user makes requests of the Yahoo! Pipes user interface (UI), responses related to the interface are redirected to the Proxy Wrapper. The Proxy Wrapper modifies response messages by inserting "widgets" and code into the Yahoo! Pipes UI before delivering them to the client, where they are decoded by the Result Decoder.

Part (b) of Figure 2 illustrates the architecture of our Anomaly Detector. The Code Extractor begins by extracting pipe code from the local database – this is JSON code containing all relevant information about the pipe, including modules, parameters and connection information. The Execution Trace Extractor extracts information pertaining to the execution of the pipe, by sending the pipe code to the Yahoo! Pipes server and retrieving the information it produces, which includes the outputs of each module and error messages. Next, the Decoder decodes the pipe code and execution traces. Specific anomaly checkers (Link Checker, Parameter Checker, Deprecated Module Checker, Content Existence Checker, and so forth) then analyze the decoded data to detect anomalies. The Result Encoder generates a log file for the pipe providing information on anomalies and errors found in that pipe.

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

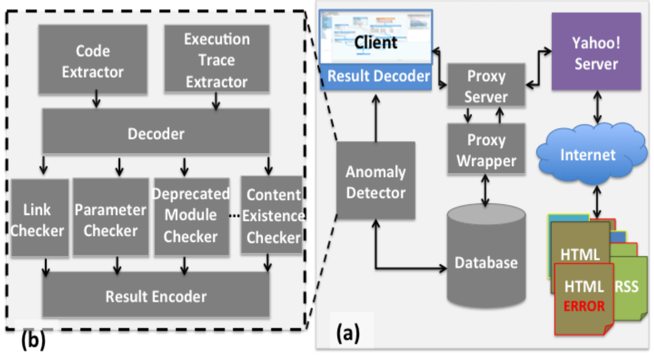


Figure 2. System architecture: (a) Overview (b) Anomaly Detector

A key technical contribution of this work involves the processes used by specific anomaly checkers to detect the presence of the various classes of bugs described in Table 1. Column 3 of the table summarizes the detection mechanisms used for each class of bug; we now provide additional details.

On intra-module bugs our anomaly checkers work as follows:

- **Link** bugs are detected by parsing pipe code for urls. Http requests for accessing these urls are sent to their respective servers and explicit statuses are analyzed. For status 200 (successful access to the page) no action is needed, but for other erroneous statuses a bug is made note of.
- **Missing:Content** bugs are detected by analyzing the pipe code for the parameters specified by users within the pipe. The contents of the web page accessed can be selected using DOM elements. The existing parameters are matched against the source code of the web page and checked to see whether these parameters exist in the requested page. If they do not, a bug is made note of.
- **Missing:Parameter** bugs are detected by statically checking fields of parameters. If a parameter value is empty the location is noted.
- **Mismatch:Format** bugs can be detected by using regular expressions or Topes [27]; however, our implementation does not yet support this functionality.

- **Mismatch:Range** bugs are detected by using assertions that check ranges of numbers used during pipe execution.
- **Syntax** bugs are detected by static analysis of the code. Our current analysis examines the structure of the rules defined for the modules being considered, and assigns a data type to each parameter of the rule. For example, in RSS feeds, attribute "title" is of type "text", "pubdate" is of type "date" and so on. We (dynamically) check the data types of parameters to determine whether they are appropriate, and check the operations performed on the parameters to determine whether they are compatible.

On inter-module bugs our anomaly checkers work as follows:

- **Module:Deprecated-Module** bugs are detected by statically checking for the presence of a module name that is no longer supported by the environment.
- **Module:Sub-Module** bugs are detected by statically checking modules that require sub-modules for the presence of a module name.
- **Module:Sub-Pipe** bugs are detected by statically analyzing pipes for inclusion of sub-pipes, and then sending a request to run the sub-pipe to the Yahoo! Pipes server; an empty response signifies that the pipe no longer exists.
- **Connector** bugs are detected by statically analyzing modules for the presence of incoming and outgoing wires.
- **Data-Type** bugs are detected by checking the type information outgoing from and incoming to a module, by statically analyzing the dataflow structure of the pipe.

AN ANALYSIS OF BUG PREVALENCE IN PIPES

To assess the occurrences of bugs in Yahoo! Pipes programs, we examined a large corpus of pipes. To do this we wrote a web crawler that collected unique pipe identifiers using a snowball sampling approach. Pipes accessible through the browsing mechanism were collected and their most frequent modules were extracted as keywords. These keywords were used to initiate searches for pipes containing them, and identifiers for those pipes were collected. Finally, identifiers of sub-pipes referenced in pipes were collected in the same way. This process retrieved 25,734 unique pipe identifiers. For each identifier we obtained the code for the pipe.

We executed our Anomaly Detector on the source for each pipe. Table 1, columns 4 and 5, show the numbers of bugs found in our sample of pipes. The most prominent class of bugs involved missing parameters. The next highest instances of bugs involved missing links and content. Bugs involving deprecated modules, missing connector elements, and data types were also relatively common.

We analyzed the outputs of buggy pipes to determine how Yahoo! Pipes responded to them. We found that 54.36% of the pipes containing bugs included at least one buggy module that emitted no error message on the console, 37.83% produced no output at all when executed, and 28.24% produced one or more Yahoo! error messages. Figure 3 illustrates the distribution of these types of outputs.

Because sub-pipes are implicated as a cause of bugs, we also analyzed the pipes that we had downloaded to obtain data on

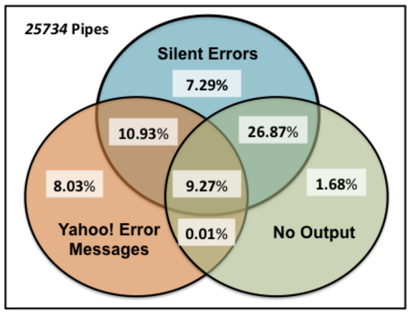


Figure 3. Types of pipe outputs and error messages observed

sub-pipe usage. Of the 25,734 unique pipes, we discovered that 20,200 were "top-level" pipes, and 5,534 were used in one or more of these as sub-pipes ("nested"). The number of sub-pipes used in a given pipe ranged from 0 to 22, and the maximum level of nesting of subpipes was seven. In fact, among the pipes that did use subpipes, 21.28% contained one or more sub-pipes at a nesting level of 1, and 16.86% contained one or more subpipes at a nesting level of 2.

CONNECTING USERS TO DEBUGGING SUPPORT

As noted earlier, run-time observation of program behavior is the primary means by which mashup programmers debug their mashups. However, users often face understanding barriers when there is insufficient or complex feedback from the Yahoo! Pipes environment [22]. To connect our debugging support to users, we designed an interface with two primary goals: (1) reducing understanding barriers to help users quickly locate and understand the causes of bugs, and (2) reducing use barriers by providing guidance on the correct usage of modules. We did this by (1) designing an automated approach for identifying bugs, (2) informing users of bugs and their causes through a user-friendly UI and messages, and (3) offering guidance on ways to fix bugs.

Our Anomaly Detector identifies bugs currently identified by Yahoo! Pipes and bugs that currently occur silently. Here we discuss our extensions to the Yahoo! Pipes user interface and the feedback messages we provide. We largely followed Neilsen's heuristics [25] in designing the interface with the main goal of reducing cognitive load on users [28], and Schneiderman's guidelines [29] for designing error messages. Figure 1(b) shows our user interface extensions, which include the following elements.

To-fix list of bugs: Prior work [11] on end-user debugging found that the use of a to-do list was a common debugging strategy, was consistently employed by users irrespective of individual differences, and helped reduce cognitive load. However, unlike professional development environments (e.g., Eclipse IDE), current end-user environments (Yahoo! Pipes, Excel) do not support to-do lists. Thus, to facilitate fault localization our UI provides a To-fix list task pane that is populated with information on bugs that need to be resolved and their properties.

The To-fix list is overlaid on the top, right-hand side of the canvas so that users can view both the pipe and the list of bugs. In the implementation we study here, we list erroneous modules in the order in which the modules appear on the canvas (top to bottom, left to right). Users might prefer this approach because a common debugging strategy involves spatially following the data-flow in a program [11]. An alternative option would be to list bugs grouped by type, allowing users to resolve all bugs of a particular type at one time; this could reduce the chance that the interface will overwhelm the user with too many alternatives [6].

An overarching goal of our interface is to provide a simple UI and provide information to users contextualized for their current task. Therefore, our To-fix list is designed as an accordion list, where only the bug that is in focus is expanded, while all other information is collapsed (Figure 1(b)). After a bug is resolved it is removed from the list.

The To-fix list is populated when the user saves or executes the pipe, or clicks on the "Find Error" button on the lefthand side of the canvas, actions that activate the Anomaly Detector. Note that in this work, the Anomaly Detector was invoked only when the Find Error button was used. To reduce cognitive load, we provide relevant information in the context of each To-fix item in the list; that is, we tie each bug in the list to the faulty module so that when a user clicks on a bug in the list, the erroneous module is highlighted (marked in orange) and parameters implicated in the bug (if any) are marked in red. We also provide reverse functionality; that is, when an erroneous module is selected (or hovered over) the To-fix list expands to reveal the bug(s) in that module.

Error Messages: We followed Neilsen's Heuristic (help users recognize, diagnose, and recover from bugs) to design error messages that are clear, concise, and use plain language. For example, a Yahoo! Pipes error message of the form: "Error [url] Response code. Error: [Invalid XML document.] Root cause: org.xml.sax.SAXParseException: [root element must be well-formed]", is translated to "Website does not contain RSS feed".

We provide constructive steps to help users arrive at solutions to bugs. Because users with different skill levels may require different amounts of information, we also provide a "Hint" button that can be expanded to provide further details for resolving the problem. In the foregoing example, we provide hints on how to find a similar website for the missing RSS feed and external documentation on how RSS feeds need to be structured. In other cases, we provide references to third party web applications such as FireBug that allow a user to inspect the elements of the webpage.

EMPIRICAL STUDY

To investigate the use of our debugging support we conducted an empirical study of our extension to Yahoo! Pipes. Our goal was to understand whether our approach can help mashup programmers locate and correct bugs in their mashups.

Study Setup

Participants

We sent emails to several departments in our university inviting students to participate in the study, promising a $20 gratuity for participation. We asked respondents to complete an online background questionnaire. To take part in our study, participants were required to have experience with at least one web language. Background in computer science was not allowed beyond the rudimentary requirements of majors. We selected 16 participants of varying backgrounds, namely: statistics, engineering, biological systems, actuarial science, physics, classics, entomology, and food science. We employed stratified sampling to categorize participants based on their experience with the web, programming languages known, gender, and knowledge of the Yahoo! Pipes interface. Participants were divided into two groups: eight participants (five males and three females) served as a control group, and eight other participants (five males and three females) served as a treatment group. Only two of the participants had knowledge of Yahoo! Pipes, and these were assigned to the control and treatment groups, respectively. Statistical tests (paired t-tests) on questionnaire data showed no significant differences ($p=0.731$) between the groups in grade point average, Yahoo! Pipes experience, web language experience or programming experience.

Procedure and Tutorial

To avoid learning effects we employed a between-subjects study design [31], with half of the participants performing debugging tasks with our extension to Yahoo! Pipes and the other half doing so with the ordinary Yahoo! Pipes interface. We used a think-aloud protocol [23], asking participants to vocalize their thought processes and feelings as they performed their tasks. We chose this protocol to obtain insights into the users' thought processes, and the barriers and problems they faced. We administered the study to each of the 16 participants on an individual basis in our usability lab.

At the beginning of each session, participants were asked to complete a brief self efficacy questionnaire [3, 5] (formulated as a Likert scale between 1 and 10); this was followed by a tutorial of approximately ten minutes on Yahoo! Pipes, which included information on how to create pipes and the functionalities of modules. The treatment group also received instructions on how to invoke the Anomaly Detector. The tutorial included a short video of a think-aloud study so that users could understand the process. After participants completed the tutorial, we asked them to create a small sample pipe to give them hands-on training and familiarity with Yahoo! Pipes. We began the experiment only after users told us that they were comfortable using the environment.

We asked participants to complete two debugging tasks. We audio recorded each session and logged the users' on-screen interactions using a screen capture system (Morae [24]). The total time required for completion of a session per participant was approximately 80 minutes, which included an average of 50 minutes for task completion. After participants completed the tasks, we conducted interviews to collect feedback or any other additional thoughts from them.

Tasks

We designed two tasks for the study. One task (Task Y!E) involved pipes containing bugs for which Yahoo! Pipes provides error messages, and the second task (Task SE) involved pipes containing bugs for which Yahoo! Pipes provides no error messages ("silent" errors). We counterbalanced the tasks to compensate for possible learning effects. We seeded three

Table 2. Details on Seeded Bugs in Tasks

Task	Class	Bugs	Details
Yahoo! Error	Top Level	B1	API key Missing
		B2	Website not found
	Nested	B3	Website not found
Silent Error	Top Level	B4	Website contents changed
		B5	Parameter missing
	Nested	B6	Parameter missing

Table 3. Bugs Localized and Fixed per Control Group Participant

Participants	Yahoo! Errors						Silent Errors					
	B1		B2		B3		B4		B5		B6	
	L	F	L	F	L	F	L	F	L	F	L	F
P1	1	1	-	-	-	-	-	-	1	-	-	-
P2	1	1	1	-	-	-	-	-	-	-	-	-
P3	1	1	1	1	-	-	1	1	1	1	-	-
P4	1	1	1	-	-	-	1	-	1	1	-	-
P5	1	1	1	-	1	-	1	-	-	-	-	-
P6	1	1	1	-	-	-	1	-	1	-	1	-
P7	1	1	1	-	1	1	1	-	-	-	-	-
P8	1	-	1	-	-	-	1	-	-	-	-	-
Total	**8**	**7**	**7**	**1**	**2**	**1**	**6**	**1**	**4**	**2**	**1**	**0**

bugs into each of the two pipes (see Table 2 for details). All bugs were located on separate dataflow paths to avoid interaction effects. We also included one bug related to sub-pipes in each pipe, to help us study the effects of nested errors

For Task Y!E we created a pipe with specification paraphrased as: "the pipe should display (1) a list of the top 10 rated movies (based on `rottentomatoes.com`) and their ratings (in descending order), (2) a poster of a selected movie (from `Flickr`) and (3) a review of the movie". For this task we seeded a "Link" bug and "Deprecated module" bug since these were the most prominent Yahoo! errors found in our study of the corpus. As a third bug we embedded another link error in a sub-pipe.

For Task SE we created a pipe for which the specification can be paraphrased as: "the pipe should display (1) a list of theaters in a given area, (2) a list of movies in each theater with their show times and (3) trailers of the top 10 movies (based on `rottentomatoes.com`). For this task we seeded the two most prominent silent errors found in our study of the corpus; namely, Missing:Content and Missing:Parameter bugs. We also included a Missing:Parameter bug in a sub-pipe.

We told participants that they had been given pipes that were not working correctly and were required to make them work correctly (that is, to find the bugs and correct them). To guide them, we gave them specifications of the pipes and of the output each pipe was intended to produce.

Limitations of our Study

We have studied only one mashup environment; however, it is representative of a broader class of web-development environments (e.g., Apatar [2], Deri pipes [8], App Inventor [1].) Our study considered only two tasks that built on only two types of pipes. Our 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. Additional studies are needed to examine other types of mashup environments, tasks, and usage contexts.

Table 4. Bugs Fixed per Treatment Group Participant

Participants	Yahoo! Errors			Silent Errors		
	B1	B2	B3	B4	B5	B6
	F	F	F	F	F	F
P1	1	1	1	1	1	1
P2	1	1	1	-	-	1
P3	1	-	-	1	1	1
P4	1	1	1	1	1	1
P5	1	1	1	1	1	1
P6	1	-	1	1	1	1
P7	1	1	1	1	1	1
P8	1	-	1	-	-	1
Total	**8**	**5**	**7**	**6**	**6**	**8**

Other limitations 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 three non-participants and using their feedback to adjust the pipes and the tasks. We performed Wilcoxon rank tests on our time data to quantitatively study the effects of time; however, because our participants were performing in think aloud mode, timing measures may be affected.

Improving Debugging Success

Turning to the results of our study, we find that the treatment group performed better than the control group in every performance measure: number of bugs localized, number of bugs fixed, time needed to fix the bugs, and perceived increase in self-efficacy. Tables 3 and 4 provide data on the debugging success of control (C) and treatment (T) groups, respectively. Table 3 shows which participants correctly localized (L) bugs and which participants fixed (F) them. Because our debugging support provided a list of buggy modules to the treatment group, for those participants the localization task was automatically completed; thus Table 4 does not include bug localization data.

Numbers of Bugs Localized

To localize a bug a control group participant needed to first correctly understand the failure of the pipe (incorrect results compared to those shown in the provided output) and then locate the buggy module. Participants in the control group spent a majority of their time attempting to do this, but were not very successful (they localized only 28 out of 48 bugs). A key reason for this involved understanding barriers – difficulty understanding the state of the program (in the case of silent errors, discussed later) or the feedback provided (in the case of Yahoo! errors, discussed next) [18].

All participants in the control group (in the Y!E task) started with Y!E:B1 which included a message, "API key missing", that was directly traceable to the buggy (`Flickr`) module, since it included the text "API key". This bug was localized by all participants and fixed by seven. However, in other cases, the Yahoo! errors included computer science jargon making them inaccessible to end users. An example error message seen by participants was: *"Error fetching [URL]. Response: OK (200). Error: Invalid XML document. Root cause: org.xml.sax.SAXParseException: The markup in the document preceding the root element must be well-formed"*. Messages such as this left users struggling to answer questions such as: (1) what is SAXParse?; (2) what is a root element?; and (3) what is meant by well-formed? Participant C.P4, for example, commented: *"What? I have no idea what it is"*, and participant C.P5 mentioned: *"Instead of error numbers, if these messages were in simple languages maybe I would have figured out the errors"*.

As noted earlier, understanding barriers are known to be notoriously difficult and are often considered insurmountable [18]. Our interface reduced these barriers by providing a list view and highlighting buggy modules, so that users could immediately recognize that there was a problem with the pipe and find the location of the problem, and by providing more appropriate error messages. These error messages were found to be helpful, as reflected in our exit interviews in which the majority of treatment group participants stated that they found the interface user friendly. T.P3 commented that *"We can see where the error message was located [and it] was very helpful. When we see the code, you can't know where the error is. If these red boxes [localization of bugs with colors] are in code also it will be helpful in coding"*.

Numbers of Bugs Fixed

After localizing a bug, the user needed to understand the correct usage of the buggy module and its external data sources (e.g., RSS feeds, URLs). This was difficult for participants in the control group (use barrier [18]), and caused them to fix statistically significantly fewer bugs than participants in the treatment group (Wilcoxon test: W=2 and p=.0008). For example, while most participants (7) localized the fault in Y!E:B2, few (only C.P3, with five years of web development experience) found the correct URL for it.

Most participants in the control group explored alternative strategies to fix the bug and needed to backtrack when they were not successful. For example, C.P4 spent 51 minutes in unsuccessful explorations, and then before continuing to the next task commented: *"I couldn't understand the error messages, they just said error but didn't tell me how to solve it. Probably if there were steps to solve I could have solved it. They expected you to know that this is an error . . . I am not familiar with how to solve them"*.

Some users were frustrated because they could not "undo" their changes and some because examples available through Yahoo! Pipes were not a close match or did not execute correctly. For example, C.P7 commented: *"Many times the examples don't work . . . if those examples worked I could have done better"*. At other times, the available examples were too complex for users to follow.

Our interface was effective for reducing use barriers because it provided hints on how to fix bugs. Most participants in the treatment group were able to use the hints, including those referring to external documentation on RSS feeds or using external helper applications such as FireBug that allows inspection of web elements. Participant T.P4 commented: *". . . the hints were helpful. I was not familiar with Yahoo! Pipes then by following the hints [I] can solve it"*.

Time Required to Fix Bugs

Another important measure of debugging success is the time spent by participants in debugging. If debugging activities take too long, end users may become frustrated and stop. Also, measuring time taken helps us track the presence of understanding and use barriers.

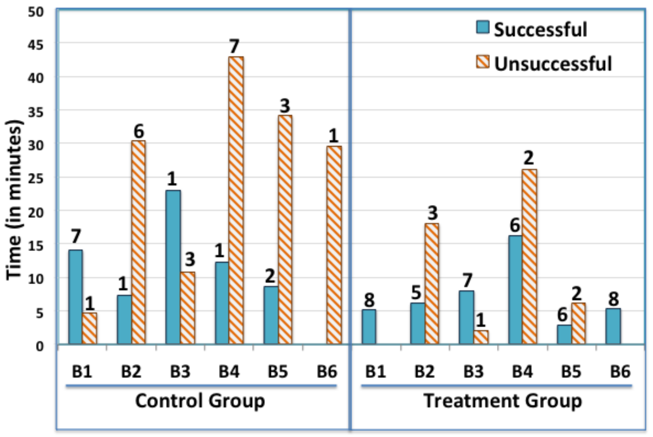


Figure 4. Average time spent by participants from each group for each bug. Solid bars indicate successful debugging, crosshatched bars represent unsuccessful attempts. Numbers over the bars indicate the number of participants who attempted to address the associated bugs.

On average, participants in the treatment group found and fixed each bug statistically significantly more quickly (Wilcoxon results W = 10, p-value = 0.010) than participants in the control group. Figure 4 shows the average times spent by participants per bug (B1 . . . B6). If a user did not attempt to address a bug their data is not shown. Overall, our interface helped users in the treatment group pinpoint bugs and solutions more quickly from the very start, and users kept this advantage throughout the tasks. Here, we discuss two bugs (B4, B2) that took the longest for the treatment group.

Bug SE:B4, in which the contents of a web page had changed, was time intensive to address and only six participants fixed it. To fix this bug, users needed to know how to check the code for the html page (i.e., page source) or how to operate an external application to inspect webpages.

Bug Y!E:B2, which contained an incorrect URL, was the next largest time sink, especially for unsuccessful participants. Participants had difficulty with this bug because there were two URLs in the `Fetch Feed` module. Most participants checked both URLs, and some removed the lower URL (which was correct) while needing correct the first URL. In these cases, since there was no way to retrieve the old URL, participants had to proceed with their task and were considered unsuccessful even if the original faulty URL was fixed.

Silent Failures of Pipes

We next investigate differences in bug resolution results between bugs in which Yahoo! Pipes displays an error message (Y!E) and bugs for which Yahoo! Pipes is "silent" (SE). Participants in the control group were considerably less successful than participants in the treatment group, identifying 17 Y!Es compared to 11 SEs and resolving 9 Y!Es compared to 3 SEs. This difference occurred primarily because without error notifications, participants were not able to tie failures to faulty modules. For example, in SE:B2, where a heading ("list of movies") in the output was missing and the buggy module had empty parameters, users could not tie the two together. C.P1 spent 40.15 minutes trying to localize this problem. In the treatment group, where faulty modules were automatically identified, we find no distinction between the two error types (20 Y!Es and 20 SEs were fixed).

Nesting of Modules

Faults seeded in subpipes (nested as modules in the original pipe) were much more difficult for control group participants to localize and fix than for treatment group participants. In the case of Y!E:B3, only one participant in the control group was able to detect and fix the bug. In the case of SE:B6, only one control group participant was able to identify the nested (buggy) module, and none fixed it. In fact, many control group participants did not recognize that modules were subpipes. Participant C.P3, on obtaining an error message on Y!E:B3, commented: *"Where is this coming from"*. In Yahoo! Pipes, to debug a subpipe users must open and execute it; C.P3 did not realize this and spent 10.44 minutes investigating the bug by clicking on other modules in the pipe. He then moved onto the next task after commenting: *"I don't know what it's saying"*.

Using our interface, a majority of the treatment group participants were able to fix the nested bugs (Y!E:B3 was fixed by 7 and SE:B8 was fixed by 8). In our archival analysis, we found thousands of subpipes, with (some) subpipes nested up to the 7th level. Without better debugging support it is highly unlikely that end users will be able to fix erroneous subpipes, especially those that fail silently.

Improvements in Perceived Self-efficacy:

Past work in the domain of end-user programming has shown that individuals with higher self-efficacy, a form of self-confidence, are more flexible in the strategies they use when performing programming-related tasks, and are inclined to explore newer, untaught functionalities [4]. In our study we found that while participants in both groups were similar in terms of self-efficacy (W = 35.5, p-value = 0.663), the treatment group performed better.

We found that participants in the control group were frustrated in their debugging tasks, and this included even those with high self-efficacy scores. Participant C.P1, with a self-efficacy score of 8.5, comments: *"I am not sure what I am doing . . . I think you can stop [recording], I am just experimenting with different tasks and I feel I am not doing the right task"*. Another participant (C.P4 with a score of 8.4) commented after being stuck in his task: *"That's why I am not a CS major"*. In contrast, participants in the treatment group — even those with low self-efficacy scores — were excited about the tasks and had a positive experience with the environment. Participant T.P5, with a score of 6.6 commented: *"I really enjoyed this [task]. It enhanced my knowledge about this [domain]. I will also make my own pipe"*.

Design Guidelines for Debugging Strategies

Our study showed that it was difficult for end users to identify and localize faults. Given that end users often program by cloning from examples and the fact that many of the available programs are erroneous (64% of the Yahoo! Pipes in our set were faulty) debugging support is likely to significantly help them. Here we provide design guidelines for incorporating debugging support into end-user environments.

Automated fault localization. In environments that use visual programming and a black box abstraction methodology, visually identifying a faulty module (and any downstream impact of that module) will help users focus their debugging efforts. We found that in the absence of such support users spent substantial effort exploring unsuccessful alternate strategies and backtracking.

To-fix list. Understanding barriers can be reduced if bugs are automatically identified and aggregated in a list. Users preferred to have the To-fix list overlaid on top of the canvas, so that they could view the pipe (or a particular module) and error message at the same time. When this was not the case (in the control group), users sometimes missed the error messages since they were sequentially listed after the generated output. In cases where a large amount of text was output users did not read all the way through and were not aware of the bugs. For example, participant C.P3 commented that *"Messages should be on top, so that they are noticeable"* in response to why he overlooked a bug.

Having a To-fix list also gave users the flexibility to select which bugs they wished to consider first. One design issue that needs to be considered in creating such a list (which was precluded by the design of our study) is interaction effects among bugs. A bug in one module can potentially manifest itself as a different bug in another module. In such cases, the To-fix list should appropriately display the interaction effects and group cascading bugs together, so that users know which module to begin with when debugging.

Avoiding technical jargon. Error messages that use simple language and avoided error codes are more successful with end users, which confirms the results of prior HCI research.

Cross-linking faults with error messages. Cross-linking faulty modules with their corresponding error messages allows users to view the error message when they are ready to debug the corresponding faulty module. Providing the error messages in the context of the debugging activity helps prevent developers from being overwhelmed by the number of errors that need to be fixed and focus on the error at hand.

Contextualized help. Providing contextualized help (suggestions for solving a problem through "hints") is more useful to users than providing example pipes and documentation at the beginning of a task.

Incremental Assistance. Presenting a high-level overview of a possible solution, followed by "hints" that users can employ, helps sustain users' interest while not overwhelming them. In our study, such incremental help enabled users with low self-efficacy to perform as well as users with high self-efficacy. In fact, our exit interviews indicate that using our approach helped raise self-efficacy in our participants.

Versioning support. When debugging, users (especially in the control group) followed various strategies and often needed to backtrack. In such cases many users explicitly asked for "undo" functionality. Traditionally, versioning support has been considered necessary only for professional programmers and is typically not supported by end-user programming environments. We found, however, that end users have become accustomed to (and have started to expect) versioning support because of their interactions with applications such as MS Word/Excel, Google Docs, Dropbox, etc., which provide rudimentary forms of such support.

CONCLUSIONS AND FUTURE WORK

We have presented an approach for detecting the presence of bugs in Yahoo! Pipes mashups, and for helping pipe programmers localize and fix these bugs. Our study of this approach shows that it does help end users find and fix bugs more effectively and efficiently than users who do not have its support.

We have implemented and studied our anomaly detector as an aid to debugging, and we have required users to explicitly request its help by clicking on a widget. The detector could also be integrated into the mashup programming environment in such a way that, on each save or execution of a pipe, anomaly detection is performed. This could help users detect bugs in their mashups that might otherwise go undetected, including bugs that emerge in initially correct mashups when features that they rely on in external environments change.

As mentioned in the introduction, our fault classification as well as our methodology for identifying fault classes and defining and implementing detectors for faults in those classes can be generalized beyond mashup environments to other web-development and visual programming environments. For example, App Inventor [1], a visual programming environment for creating mobile apps, faces the same types of inter and intra module bugs that we have identified, although some sub-classes of these defects will need to be refined to suit the particular domain. We plan to validate and refine our approach by investigating App Inventor next.

Acknowledgments

This work was supported by AFOSR (FA9550-10-1-0406) and NSF (1016134) grants. We thank our study participants.

REFERENCES

1. App Inventor: http://appinventorapi.com/.
2. Apatar: http://apatar.com/.
3. Beckwith, L., Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., and Hastings, M. Effectiveness of end-user debugging software features: are there gender issues? In *CHI* (2005), 869–878.
4. Cao, J., Rector, K., Park, T. H., Fleming, S. D., Burnett, M., and Wiedenbeck, S. A debugging perspective on end-user mashup programming. In *VLHCC* (2010), 149–156.
5. Compeau, D. R., and Higgins, C. A. Computer self-efficacy: development of a measure and initial test. *MIS Quarterly 19*, 2 (1995), 189–211.
6. Constantine, L. L., and Lockwood, L. A. D. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. ACM Press/Addison-Wesley, 1999.
7. Cypher, A., Dontcheva, M., Lau, T., and Nichols, J. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010, Ch. 22.
8. Deri Pipes: http://pipes.deri.org/.
9. Dinmore, M. D., and Boylls, C. C. Empirically-observed end-user programming behaviors in Yahoo! Pipes. In *PPIG* (2010).
10. Grammel, L., and Storey, M.-A. An end user perspective on mashup makers. In *Technical Report DCS-324-IR*, Department of Computer Science, University of Victoria (2008).
11. Grigoreanu, V. I., Burnett, M. M., and Robertson, G. G. A strategy-centric approach to the design of end-user debugging tools. In *CHI* (2010), 713–722.
12. IBM Mashup Maker: http://www.ibm.com/software/info/mashup-center/.
13. Jackbe: http://www.jackbe.com/.
14. Jones, M., and Churchill, E. Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In *CCT* (2009), 51–60.
15. Jones, M. C., Churchill, E. F., and Twidale, M. B. Mashing up visual languages and web mash-ups. In *VLHCC* (2008), 143–146.
16. Karam, M., and Smedley, T. A testing methodology for a dataflow based visual programming language. In *HCCLE* (2001), 280 –287.
17. Ko, A. J., and Myers, B. A. Designing the Whyline: A debugging interface for asking questions about program behavior. In *CHI* (2004), 151–158.
18. Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. In *VLHCC* (2004), 199–206.
19. Koesnandar, A., Elbaum, S., Rothermel, G., Hochstein, L., Thomasset, K., and Scaffidi, C. Using assertions to help end-user programmers create dependable web macros. In *FSE* (2008), 124–134.
20. Kulesza, T. Toward end-user debugging of machine-learned classifiers. In *VLHCC* (2010), 253 –254.
21. Kuttal, S. K., Sarma, A., and Rothermel, G. History repeats itself more easily when you log it: Versioning for mashups. In *VLHCC* (2011), 69 – 72.
22. Kuttal, S. K., Sarma, A., and Rothermel, G. On the benefits of providing versioning support for end-users: An empirical study. In *Technical Report TR-UNL-CSE-2012-0008*, Dept. of Computer Science, U. Nebraska (2012).
23. Lewis, C. H. Using the "Thinking Aloud" method in cognitive interface design. RC 9265, IBM, 1982.
24. Morae: http://www.techsmith.com/morae.asp.
25. Nielsen, J., and Molich, R. Heuristic evaluation of user interfaces. In *CHI* (1990), 249–256.
26. Rothermel, G., Li, L., DuPuis, C., Burnett, M., and Sheretov, A. A methodology for testing form-based visual programs. *TOSEM 10*, 1 (Jan. 2001), 110–147.
27. Scaffidi, C., Cypher, A., Elbaum, S., Koesnandar, A., Lin, J., Myers, B., and Shaw, M. Using topes to validate and reformat data in end-user programming tools. In *WEUSE* (2008), 11–15.
28. Sharp, H., Rogers, Y., and Preece, J. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2007, Ch. 15.
29. Shneiderman, B. Designing computer system messages. *CACM 25* (1982), 610–611.
30. Stolee, K., Elbaum, S., and Sarma, A. End-user programmers and their communities: An artifact-based analysis. In *ESEM* (2011), 147–156.
31. Wohlin, C., Runeson, P., Hóst, M., Ohlsson, M., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering: An Introduction*. Springer, 2000.
32. xfruits: http://www.xfruits.com/.
33. Yahoo! Pipes: http://pipes.yahoo.com/pipes/.