Empirically-Observed End-User Programming Behaviors in Yahoo! Pipes

Matthew D. Dinmore

Applied Information Sciences Department
Johns Hopkins University
Applied Physics Laboratory
Matthew.Dinmore@jhuapl.edu

C. Curtis Boylls

Applied Information Sciences Department
Johns Hopkins University
Applied Physics Laboratory
Curt.Boylls@jhuapl.edu

Keywords: end-users, visual languages, mental models, phenomenology

Abstract

Yahoo! Pipes is a well-known, widely used visual programming environment for creating data mashups by aggregating, manipulating, and publishing web feeds. It provides a natural laboratory for observing a range of end-user programming (EUP) behaviors on a large scale. We have examined more than 30,000 Pipes compositions in a search for regularities that might inform the design of EUP systems and their services. Although Pipes primitives span a broad range of functionality and can be richly parameterized and composed, we find a number of patterns that govern the structure and parameterization of Pipes in the wild. Most users sample only a tiny fraction of the available design space, and simple models describe their composition behaviors. Our findings are consistent with the idea that users attempt to minimize the degrees of freedom associated with a composition as it is built and used.

1. Introduction

Research focused on user behavior in end-user programming (EUP) has typically been conducted through observation of the activities of a few users in a relatively small environment, for example a workgroup within an organization or laboratory experiment (Myers, et. al., 2006). The growth of the Internet and EUP tools for it now offers a large-scale, open-access, live environment in which it is possible to observe many end users at work, albeit primarily through the artifacts they create rather than observation of their direct interactions with the EUP tool.

Here we report preliminary results from an empirical examination of artifacts created by users of the Yahoo! Pipes web-based visual programming environment. Our research questions are motivated by our interest in understanding users' compositional behaviors, and in particular, how they make use of the collection of primitives provided by the environment, whether this collection adequately covers the range of needs they attempt to satisfy with their compositions, and how they manage complexity. This paper proceeds as follows: after a brief review of the literature, we describe the Pipes environment, the measurements that can be derived from it, and our rationale for selecting it for this study. We then detail our data collection and preparation methods. Using the resulting data corpus, we next examine the question of how users who create pipes (end-user programmers) compose their solutions given the primitives provided by the environment, and we identify and discuss several models that fit our observations.

1.1 Background

Our primary motivation for this work is a desire to understand how end users compose solutions to problems in their domain; our focus is on the collection of primitives offered to the user and their behaviors in using them. From an engineering point of view, how can we model these behaviors for the purposes of either designing a set of primitives, or given alternative collections of primitives, choosing among them for a particular task? Theoretical models such as Blackwell's (2002) Attention Investment model consider this question in the context of the "cost" for a user to compose a solution in a particular environment. Can empirical observations relate measurable costs to typical behaviors?

Likewise, what are the reuse behaviors of users? It has been observed that end users tend to learn by building on the work of others (Gantt & Nardi, 1992); can an environment facilitate this? To what degree to users make use of the ability to copy and modify? If offered the opportunity, do users create hierarchical abstractions – reusable modules – for themselves and others? And, once a user has solved a problem, to what extent are parameters exposed to other users for run-time customization? This study begins to address these questions, with the focus primarily fixed on the design-time choices users make in composing their solutions.

1.2 Other Large-Scale Studies

Few large-scale empirical studies of end-user programming have been previously reported. Several studies have made uses of the EUSES spreadsheet corpus (Fisher & Rothermel, 2005) to examine the behaviors of end-user spreadsheet developers; this corpus contains 4,498 artifacts. Bogart, et. al. (2008) studied the CoScripter web scripting environment, and asked questions similar to our own about user behavior. While both CoScripter and Pipes automate web processes, CoScripter is different in that it could be described as text-based rather than visually-based, and execution of scripts occurs at the browser (client), rather than at the web server. The authors examined a corpus of 1445 unique scripts, in which they characterized user behaviors and scripting processes, to include reuse.

While Yahoo! Pipes is often cited as an example in work on web mashups, there have been few specific studies of it or its artifacts. The largest we are aware of involved an examination of the social network around Pipes by Jones and Churchill (2009), who looked at communications among users on web groups associated with Pipes. They studied the posts of over 2,000 users, categorizing the user network and the kinds of exchanges hosted on it.

2. The Yahoo! Pipes Environment

Yahoo!Pipes is a web-based, visual programming environment introduced by Yahoo! in 2007 with the intent of enabling users to "rewire the web."[1] Pipes is a dataflow system in which the data is sourced from the web (RSS feeds, web pages, raw data) and flows through an interconnected set of modules that act upon it, ultimately producing some result; the Pipes name is inspired by the concept of pipes in Unix operating systems that enabled the composition of command line sequences of Unix tools through which data “flow” for processing.

As a visual programming environment, Pipes is well suited to representing solutions to dataflow-based processing problems (Whitley, 2006), and is also quite accessible to end-user programmers. Here, we use the goals-based definition of end-user programmer—one who is creating software to solve a problem in his/her domain of expertise, as contrasted with a professional programmer who creates software for users in other domains (Ko, et. al., 2008); we suspect our data contain artifacts created by both trained and untrained programmers.

The visual programming paradigm is one of the more common end-user programming modalities (Burnett, 1999) and has been the subject of many end-user programming studies and tool development projects (Kelleher & Pausch, 2005). Pipes offers the opportunity to extend this research to incorporate the behaviors of thousands of users motivated to mash up content and services available on the web for their own purposes. The compositions that they create in this process can be examined from a number of perspectives, and doing so *en masse* offers an aggregate reflection of their approach to problem solving through programming.

Pipes consists of a publicly accessible website that allows users to find existing pipes (by browsing or searching by keyword), and to execute those pipes. Execution often involves entering runtime parameters and results in the production of a web page. Users can also edit existing pipes and create entirely new compositions. Our focus is on the structure of composed pipes, and in particular, three elements of Pipes composition over which the users have direct control: the selection of modules, the wiring of those modules, and the settings of parameters within the modules. Related to the last of

[1] http://pipes.yahoo.com/

these are also the parameters that the end-user developer exposes to users of the pipe in the form of runtime settings. Figure 1 depicts these elements as they occur in a typical pipe.

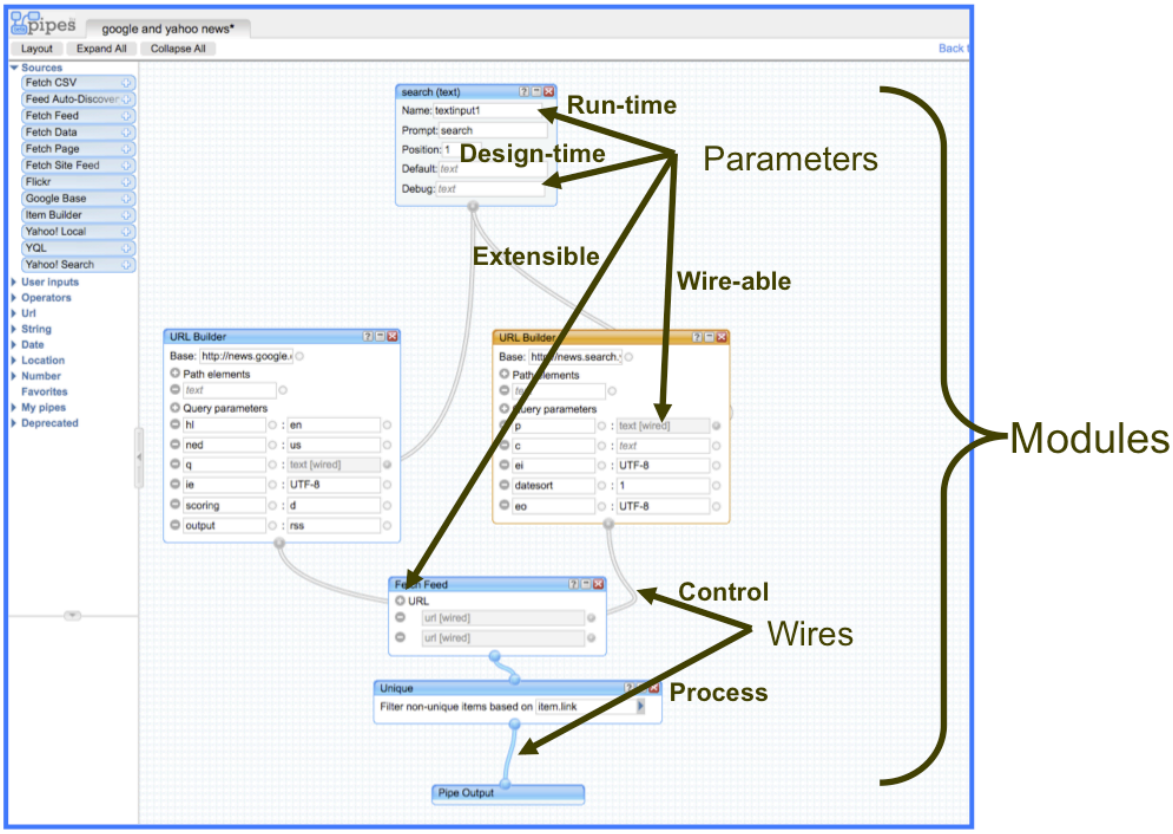

Figure 1. Annotated view of a typical pipe's elements.

A pipe is assembled from a selection of modules. Presented as visual tokens along the left side of the Pipes editor, these modules can be dragged onto the composition canvas, where their design-time parameters are presented for editing, and they can be connected to other modules to form the pipe. There are 51 primitive module types arranged in eight categories, including sources, user inputs (that present a run-time parameter to a user), and classes for manipulating strings, URLs and other types of data.

Within the pipe, the primary or “process” dataflow, which must originate in one or more source modules, is denoted with blue "wires." Pipes also contain “control” wires, colored gray, that enable parameters captured by one module to be wired to the outputs of other modules, allowing for those parameters to be programmatically set.

The parameters for each module are presented as elements in a form within the body of the module. Some offer a selection of settings (including binary on/off, radio buttons or a pick-list), while others are freeform. As noted previously, certain parameters can be wired at design-time to the outputs of other modules.

Pipes created by a user are saved in the user’s account; they can also be published, which makes them visible in searches to other users (however, it is not necessary for a pipe to be published to be publicly accessible, provided its URL is known). A user can also clone another user's pipe and employ it as the starting point for developing their own; this facilitates a common "scaffolding" behavior in end-user programming (e.g. Repenning & Ioannidou, 2006; Nardi & Miller, 1990) and represents a kind of reuse.

Users also have the ability to augment the suite of primitive Pipes modules by designating an existing pipe as a “subpipe,” a property that we will exploit in the analysis below. In this case, the (sub)pipe appears in the user’s workspace as another module, and its runtime parameters are presented as the

module's design-time parameters. Subpipes can be embedded within other pipes to any number of levels. Subpipes do have one significant limitation: they cannot accept a blue-wire dataflow as an input. It thus is not possible to create a pass-through "processing" subpipe. As a result, subpipes are typically used as specialized data sources. Because users design and employ subpipes to "extend" the repertoire of Pipes modules, we might expect to see different usage behaviors between pipes and subpipes. We thus have separately analyzed the data from each, looking for such differences and their implications.

3. The Pipes Data Corpus

3.1 Data collection

We collected the data set for this study over the 52 days between 31 December 2008 and 21 February 2009. Yahoo! Pipes does not provide direct access to the underlying "source code," nor does it offer the ability to simply list all available pipes. Instead, it is possible to browse a list of pipes presented by the interface, and one can also search by keyword. However, access to search results is limited to 1010 items. These restrictions and others necessitated a snowball-styled sampling approach that proceeded as follows: those pipes offered through the browsing mechanism were collected and their most frequent keywords extracted. These keywords were then used to initiate searches, and those pipes were collected, as well. Part of the collection process involved searching for a pipe by searching on its specific title; in addition to returning the pipe of interest, this also often resulted in other hits, which were similarly collected. Finally, some pipes included subpipes, which were subsequently collected in their own right, as well as pipes related to them. The entire process resulted in a collection of 70169 pipes.

We obtained two kinds of data about each pipe instance. First, we gathered metadata about the pipe: when it was created, the author, the title and descriptive text provided by the author, whether it had been cloned or identified as a favorite, and its run count (the number of invocations) on the date collected. All of these fields were databased for analysis by collection date; pipes were revisited to attempt to capture changes made over time, resulting in some cases in multiple records for each pipe.

We also collected the structure of the pipe by downloading its JavaScript Object Notation (JSON) file and associating it with the metadata. This structure includes all of the details necessary to reconstruct the pipe, including the modules and their placement, the wiring, and the parameter settings.

3.2 Data refinement

In our early examination of the data, we removed duplicates as well as those that had damaged JSON files. It became clear there were large numbers of replicated pipes that were structurally the same, but differed in the feeds they were creating. These proved to be targeting web pornography. As these instances biased our data with their large numbers and did not satisfy our interest in problem solving by end-user developers, we chose to exclude them from our analysis. We found that this could be done by purging pipes in Pipes accounts containing 115 or more pipes; 41 accounts representing 27115 pipes, or 661 pipes per account appeared to be in this business (again, without exception). This reduced our collection to 43054 pipe instances. For an additional 1599 instances, we found that we were unable to collect some portion of the data (typically the structural file), and these were also discarded.

We note here that we have made a decision about a class of functionality that is not of interest to us (namely mass-produced pornography-related pipes) based on the observation that the class consists largely of replicated pipes having different parameterizations. We have also not otherwise attempted to understand the semantics of the pipes that we are studying (see Discussion), so our data may include other instances of pipe replication with little novel end-user design, as well as instances in which we have retained pipes with no meaningful utility; we believe the numbers of these would be small, relative to the class of pipes we eliminated, but this nonetheless represents a limitation in our understanding of the corpus.

For the remaining 41455 instances, we databased the following data pertinent to our study:

- The unique pipe identifier (UID) assigned by Yahoo! when the pipe was created
- A self-assigned account identifier for the owner of the pipe (nominally the "author")
- The pipe configuration: modules with their identifiers, inter-module wiring, parameters
- The total number of invocations ("runs") of the pipe at the time it was collected
- Whether or not the pipe was used as a subpipe, and which other pipes "called" it
- Whether or not the pipe was published (made visible and searchable by visitors to the site)
- Whether or not the pipe was marked as having been "cloned"

We also obtained other information (e.g., the order in which users selected modules for the pipe, the location of module representations in the visual workspace, etc.) that will be useful in further investigations (see Discussion).

We quickly discovered that "raw" Pipes data include phenomena that could obscure our view into the final programming choices made by end-user authors. Most of these nuisance variables arise because Pipes is a live construction zone: authors are free to leave and return from work in progress arbitrarily, and no Pipes metadata element, including publication status, reliably indicates that a pipe is "complete." However, we can easily spot and eliminate metadata from two types of non-functional pipes that probably represent intermediate stages of construction:

a. All pipes are required to have an "output" module, so any pipe having only one module cannot be doing anything interesting. We found 86 instances in our data.

b. All pipes having N modules, but fewer than N-1 wires, cannot be connected as a single pipe. At least one module must be an "orphan." Our data contained 2656 examples.

Eliminating these two categories, our corpus shrank to 38713 instances. We also considered culling the data further by ruling out unpublished pipes (2384 instances) or pipes with few invocations, but we decided against that because the affected population is small and the possible relationship with end-user programming behavior is ambiguous.

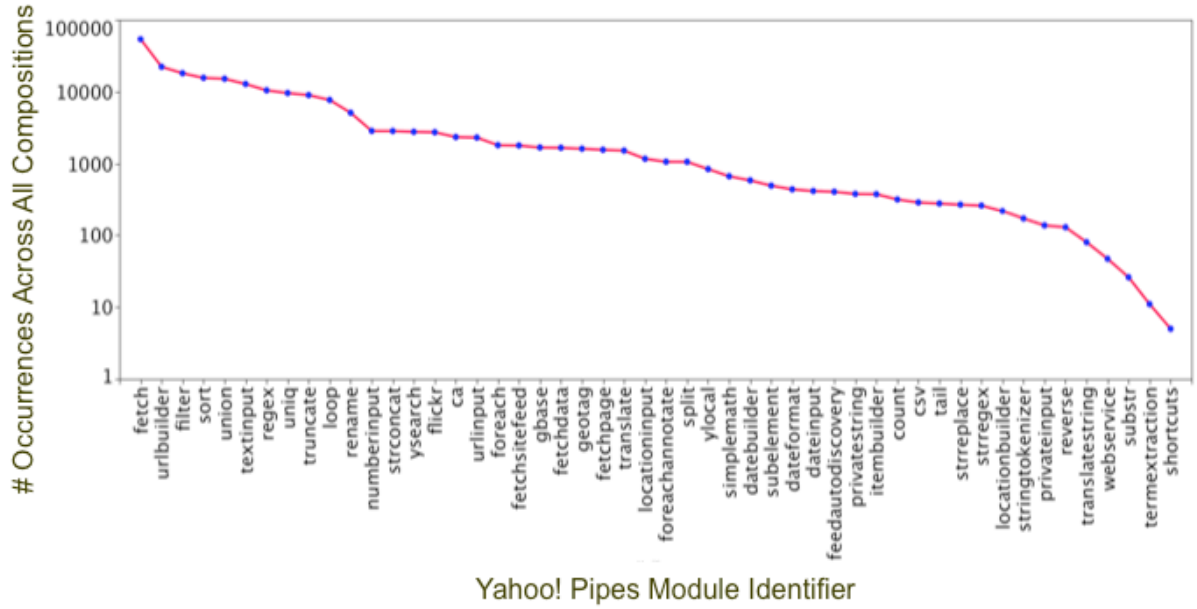

Figure 2. Primitive module usage.

We next attempted to infer something about the tasks that end-user programmers are accomplishing with Pipes. We counted the number of times that each module type was found in a Pipes composition and rank-ordered the modules accordingly. Fig. 2 plots the results of this module "popularity contest" (see section 4.3 for another use of this information). Note that the most frequently used modules (by

far) are those associated with fetching and parameterizing data (*fetch, URI builder*), followed by modules that filter, transform, and merge data (*filter, sort, union*, etc.). Pipes is advertised as a vehicle for creating data mashups, and these observations are consistent with that use. In section 4.3, we will use the same "popularity" information to assess other influences on users' choices of modules for compositions.

Pipes metadata does not directly tell us anything about the actual end-users whose work is represented in the corpus. Users must create Pipes accounts to author compositions; but since multiple accounts are allowed, one user, in principle, could have created all of the data. We did, however, observe 22285[2] unique account identifiers across all instances in the corpus, an average of 1.74 instances per account. The maximum number of instances owned by one account was 110 (recall that we eliminated pipes by authors with 115 or more instances, as these were all pornographic and probably mass produced). We cannot draw strong conclusions about the size of the user population from these figures alone. But it does seem reasonable to assume that a goodly number of "real humans" account for the data, perhaps a population larger than has heretofore been the case in studies of end-user programming in the wild. Further, we made no attempt to characterize the degree of programming experience each user had, accepting that those with professional training may approach problem solving differently than those without. We do believe that that future efforts with this data, particularly with amplifying information such as that presented by Jones and Churchill (2009), could potentially categorize users based on the artifacts we can observe.

Lastly, we partitioned the corpus into metadata from pipes that were never used as subpipes (36676 instances) from those that were (2037). Recall that, once designed and published, subpipes take on the persona of a Pipes data-source module, including an analogous visual representation in the workspace. This property will allow us to examine how end-users extend the Pipes environment as meta-designers for other end-users.

4. End-User Programming Behaviors in Pipes

4.1 Choosing modules

To create a new composition in Yahoo! Pipes, an end-user must first drag modules from a menu into a workspace before wiring them together and setting module parameters. Other than requiring a single "output" module (that Pipes supplies automatically), Pipes places no limits on how many modules a user employs, and the visual workspace expands to accommodate additions. Pipes also supplies "layout" functions to regularize complex constructions.

Despite the support provided for creating large compositions, the pipes and subpipes in our corpus proved to be quite parsimonious, as shown by the descriptive statistics in Table 1. We see that, while outliers exist, the median number of modules is only 4 for pipes and 6 for subpipes, and compositions of roughly 40 or fewer modules account for more than 99% of the data.

	Pipes	Subpipes
Instances	36676	2037
Minimum modules/composition	2	2
Maximum modules/composition	177	87
Mode	3	4
Mean	6.2	7.4
Median	4	6
99.5% Quantile	33	43

Table 1. Modules in compositions.

[2] We note that Jones and Churchill (2009) report "over 90,000 developers" at the time of their study in December, 2008, suggesting that our contemporaneous sample of pipes represents roughly 1/3rd of the author population.

If we look at the empirical probability of finding compositions of length N in our collection (Fig. 3), we see that the distributions for pipes and subpipes are similar, but that subpipes are biased toward including more modules (see also Table 1).

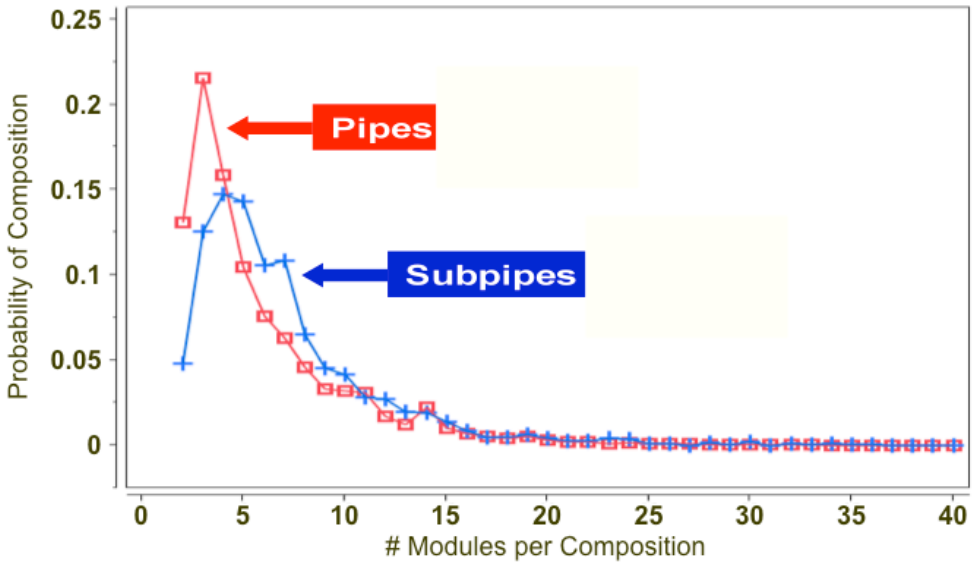

Figure 3. Distribution of pipe and subpipe size (number of modules).

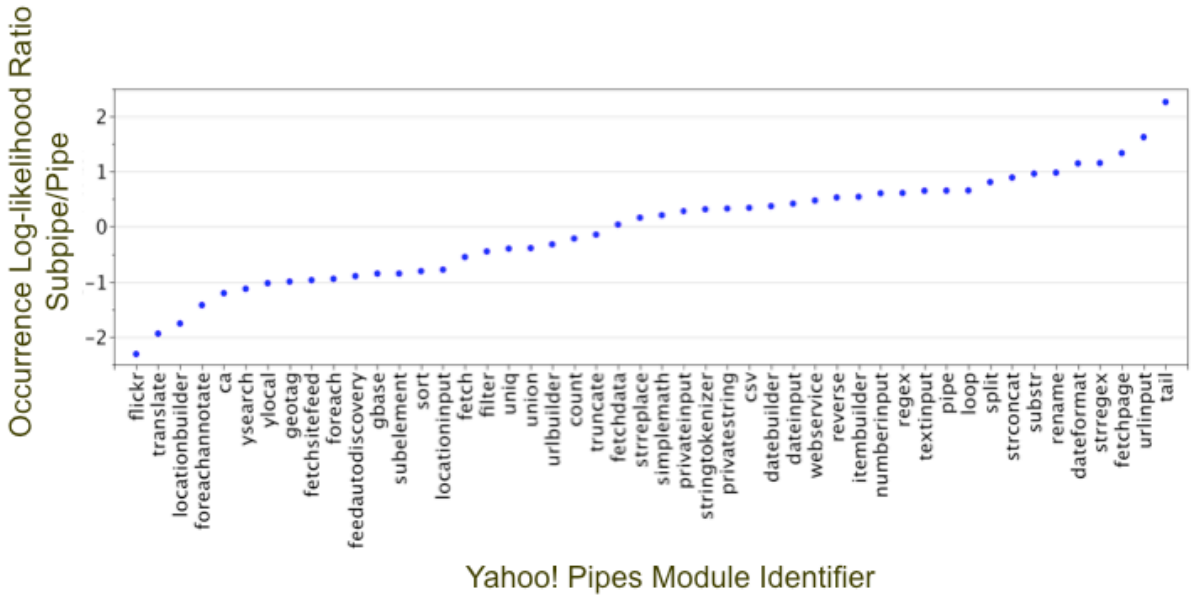

Figure 4. Log-likelihood ratio of use in a pipe vs. subpipe for each module type.

If, through log-likelihoods, we compare how often specific modules are used in pipes versus subpipes (Fig. 4), we see that the utilization of roughly ¼ of Pipes modules differs by several orders of magnitude between pipes and subpipes. Subpipes are not just a random subset of pipes that users have decided to encapsulate as extensions of the Pipes module set.

Given their tendency to include more, and different, modules in subpipes relative to pipes, it might be that users are attempting to use subpipes to hide complexity behind simple interfaces. In Fig. 5, we explore this hypothesis by comparing, for both pipes and subpipes, the average number of modules encapsulated in a composition—a coarse-grained indicator of functional complexity—with the

number of interface parameters[3] that the author provides to users of the composition. We see that as the count grows, the number of parameters in both pipes and subpipes initially increases, but then levels off, even as "complexity" increases (see upper and middle panels of Fig. 5). This favors the "complexity-hiding" argument. However, except for very small compositions, the subpipes:pipes ratio of these counts hovers around 2.5:1, indicating that complexity-hiding is found in both types of compositions. Note: the erratic subpipe behavior at higher module counts is a sparse-data artifact.

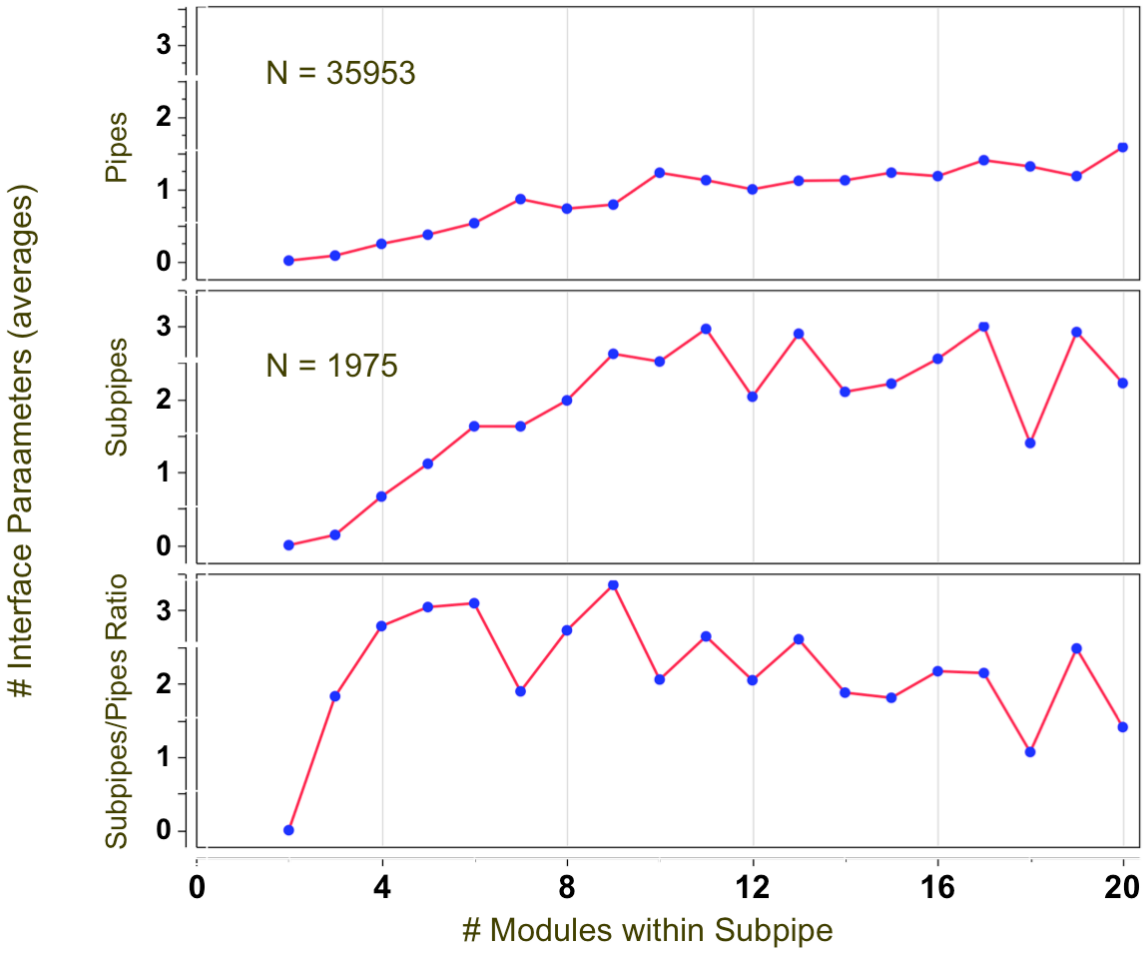

Figure 5. Comparative use of exposed interface parameters in pipes vs. subpipes

The "exponential" shape of the right-hand tail of each distribution in Fig. 2 led us to examine the log probabilities in those tails (Fig. 6). Not only is the exponential fit reasonably good, but the slopes of the fitted regression lines are both approximately 0.18. This suggests that, to a first approximation, an end-user adds modules to a composition based upon the outcome of tossing a figurative coin with p = exp(-0.18) = 0.84 in favor of the addition. Furthermore, this (binomial) process qualitatively describes module composition for both pipes and subpipes.

[3] When a pipe is converted into a subpipe, the pipe's run-time parameters become the subpipe's design-time parameters (section 2). We refer to both collectively as "interface parameters."

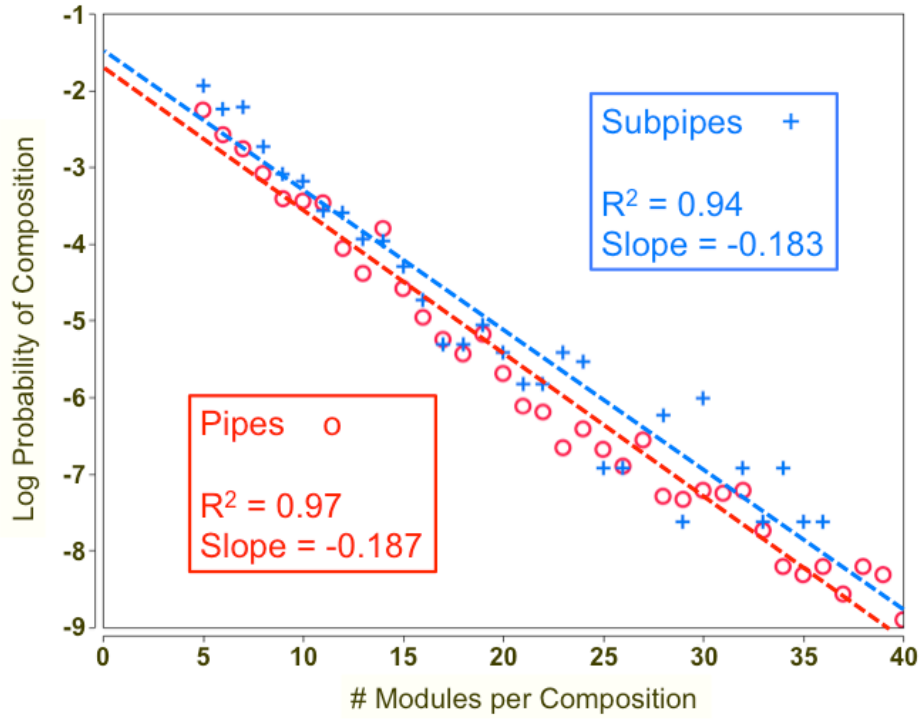

Figure 6. Log fit of probability of observing a pipe or subpipe containing N modules.

4.2 Choosing wires and configurations

As an environment originally intended for mashing-up RSS feeds, the Pipes environment includes features that encourage convergent "stream-processing." Not only is a composition allowed only one output, but wires are segregated into those that carry data and those that carry parameters; dataflow in both is one-way, and most modules have only one output (as does a pipe composition) and default to a handful of inputs. However, Pipes does include explicit "split" and "union" modules that allow streams to be split, merged, and processed in parallel, and a "loop" construct allows iteration. Many modules also allow expansion of parameters and their associated wires. The question is, do end-user programmers exploit this richness?

The answer is, "not very often." Fig. 7 plots the average and the range of the number of wires in a composition as a function of the number of modules in that composition. We do this for both pipes and subpipes, and we make no distinction between data- and parameter-carrying wires. For both pipes and subpipes, the average number of wires is almost perfectly predicted by the relation: # wires = # modules – 1. In other words, users tend to employ only the minimum number of wires needed to connect modules. Moreover, as evident in the range bars in Fig. 7, this propensity is even stronger in subpipes than it is in pipes. Given this finding, it seems reasonable to propose that end-users configure most Pipes as directed acyclic graphs, often a linear chain or cascade of modules.

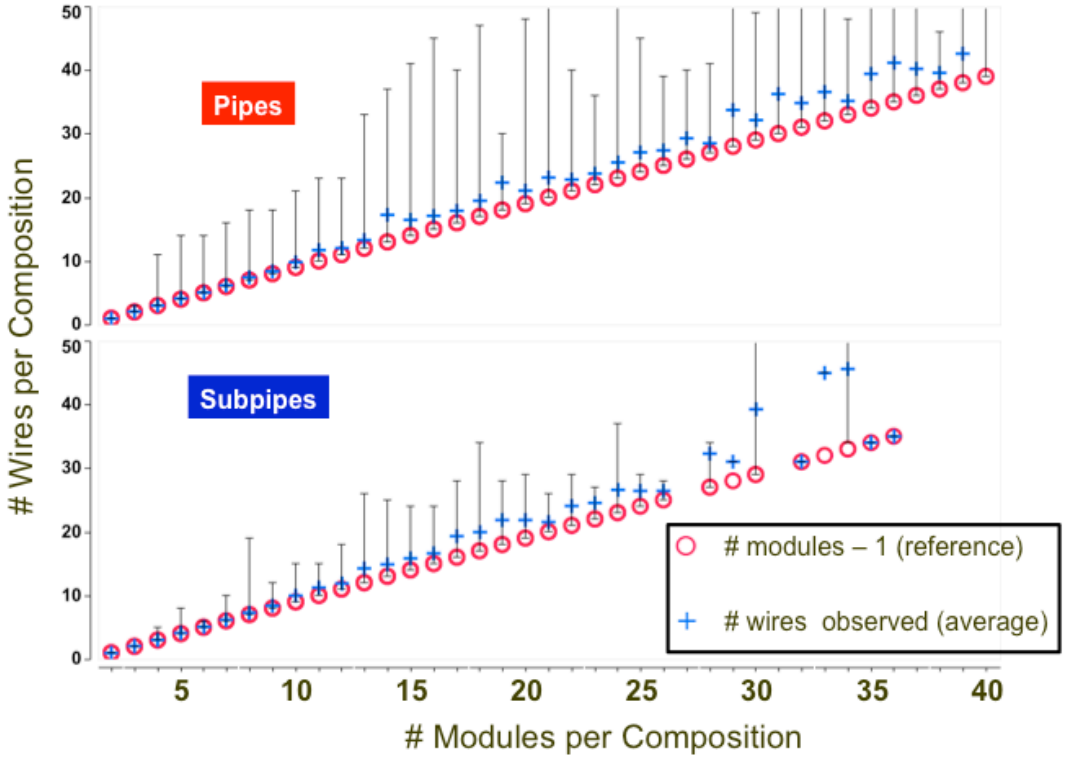

Figure 7. Relationship of number of modules and connecting wires for pipes and subpipes.

4.3 Choosing parameters

Most of the 51 primitive Pipes modules have parameters that end-user programmers must either default or set, some when the pipe or subpipe is configured ("design-time"), and some when it is executed ("run-time"). Fig. 8 shows how the design-time parameters distribute over the module collection as initially presented to the user as a "module menu."

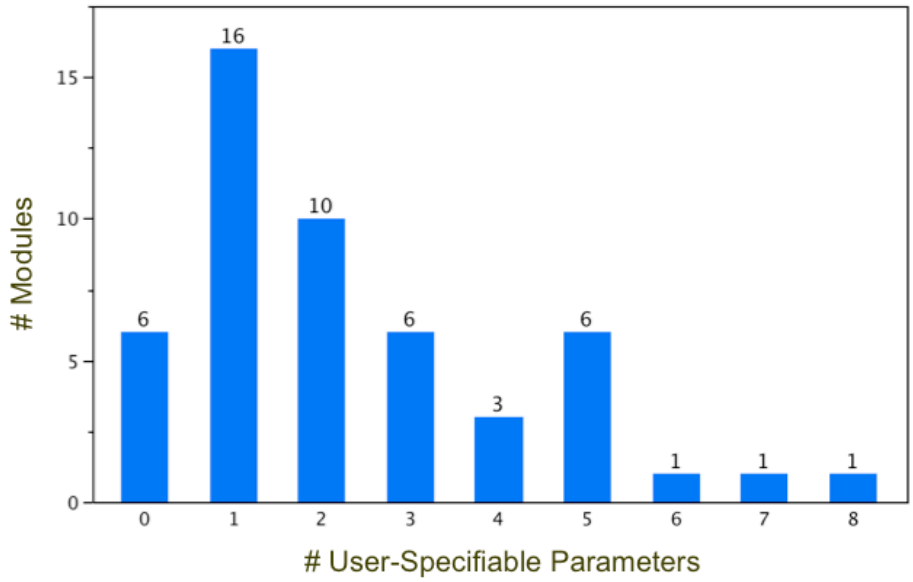

Figure 8. Distribution of available design-time parameters in the Pipes primitives.

We know from the module "popularity contest" in Fig. 2 (section 3.2) that users have strong preferences for certain modules in compositions, and that these preferences correlate with Pipe's use as a data mash-up environment. However, we also wondered if there might be a relationship with the number of parameters that a user must contend with in using a module: too few parameters might

seem to limit flexibility; too many increases complexity (or "cognitive load") and work-factor. Fig. 9 plots the number of parameters in Pipes modules as a function of module usage or "popularity." As in Fig. 2, the most frequently used modules appear on the left, and the least-used on the right. We see no obvious pattern, perhaps because none exists, or perhaps because of two confounding factors: first, Pipes modules are functionally independent. One cannot readily derive the function of a given module from some allowable combination of the others. Pipes users thus have no choice about using certain modules for certain functions; unless users forego those functions altogether, their "preferences" make no difference in module choice. Second, during composition, users are able to add an arbitrary number of parameters to many modules (example: additional regular-expression patterns to the *regex* module). A "simple" module can thus become very complex at the user's discretion. Fig. 9 shows only the default number of parameters, not the results of such extensions.

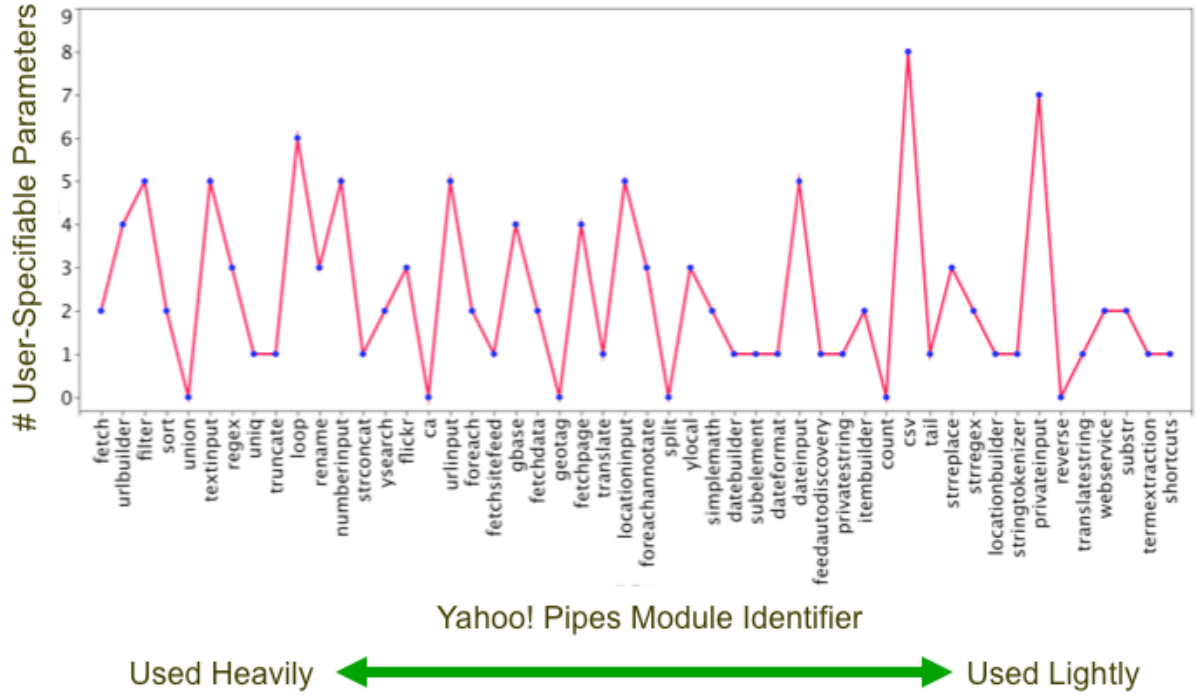

Figure 9. Relationship between primitive selection and design-time parameter count.

We can, however, gain insight into the influence of module parameters on end-user compositional behavior by looking at the fabrication and utilization of modules that users themselves create: the subpipes. Recall that when an end-user author publishes a subpipe, any interface parameters for the subpipe appear in the subpipe's representation as an addition to the "module menu." Authors are free to specify as many parameters as they wish. But if we look at how interface parameters actually distribute across subpipes, we see a familiar pattern (Fig. 10): subpipes with no parameters are the most common, and the frequency falls steeply as the number of parameters increases. The falloff is exponential and fits a binomial, coin-tossing model with p approximately 0.4. Thus, the process of adding interface parameters when a subpipe is originally composed is empirically akin to the process of adding modules to compositions in general (section 4.1).

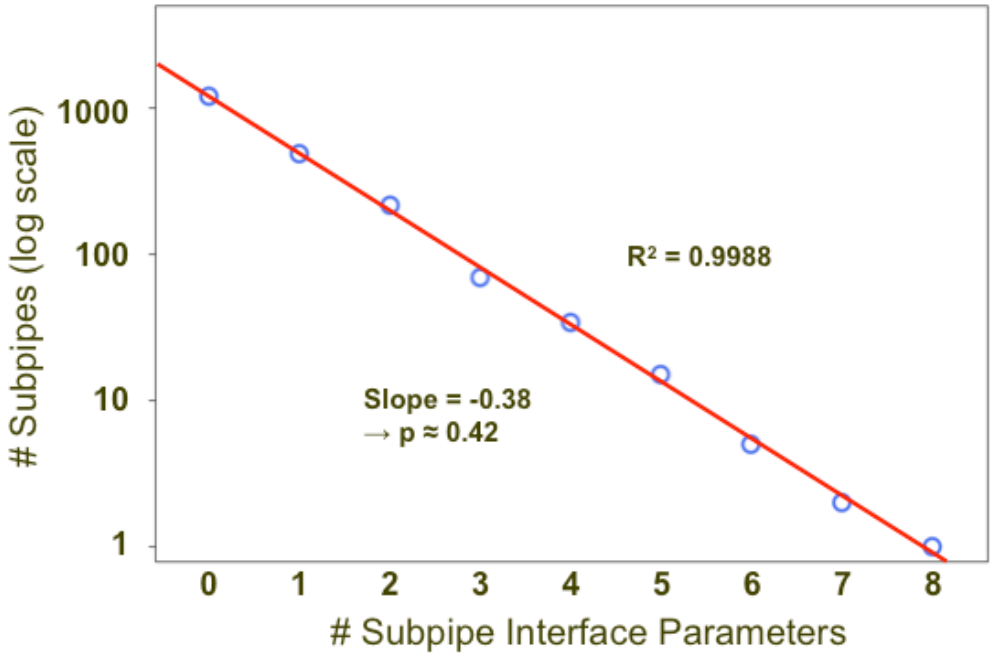

Figure 10: Log fit of relationship between observed subpipes and number of interface parameters.

How, then, might the number of interface parameters affect the likelihood that an end-user will choose to employ a published subpipe in another composition? Since subpipes are presumably created to save users the effort of fashioning specialized functions, we might expect to see the same "random" relationship between parameters and the likelihood of subpipe utilization as we do with other Pipes modules (Fig. 9). To examine this, we first need to find those subpipes that: (a) have actually been made available to other end-users by their authors; and (b) have actually seen service. Of the 2037 subpipes in the corpus, only 199 meet these criteria. Fig. 11 illustrates how often these subpipes were used in other compositions (average and range) as a function of the number of interface parameters that they present. It appears to us that, except when a subpipe has no parameters at all, the number of parameters has little influence on whether a subpipe is selected for use.

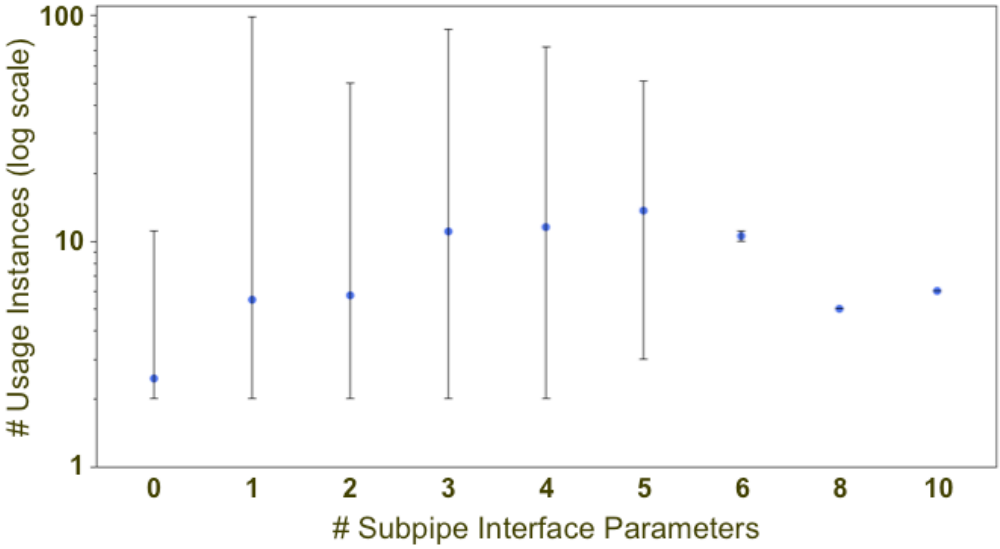

Figure 11. Relationship of exposed subpipe interface parameters and observed reuse.

A final question that we asked involved what we refer to as the reduction in degrees of freedom from the composition space to the use space. When a user composes a pipe, he or she has to make choices about which parameters to set within the composition at design time – thereby fixing those values – and which to expose to other users as run-time parameters. The ratio for a given pipe of parameters

offered to users at run-time to those set and fixed at design-time is the reduction in degrees of freedom. We would expect that developers of pipes seek to make a task easier, and therefore attempt to reduce the number of parameters that need to be set by a user. However, providing too few run-time parameters may make the pipe less generalizable to other purposes, reducing its utility to users other than the developer. Figure 12 shows the distributions of design-time parameters actually set by the developer (this *includes* extended parameters, but *excludes* defaulted parameters), as well as the distribution of run-time parameters.

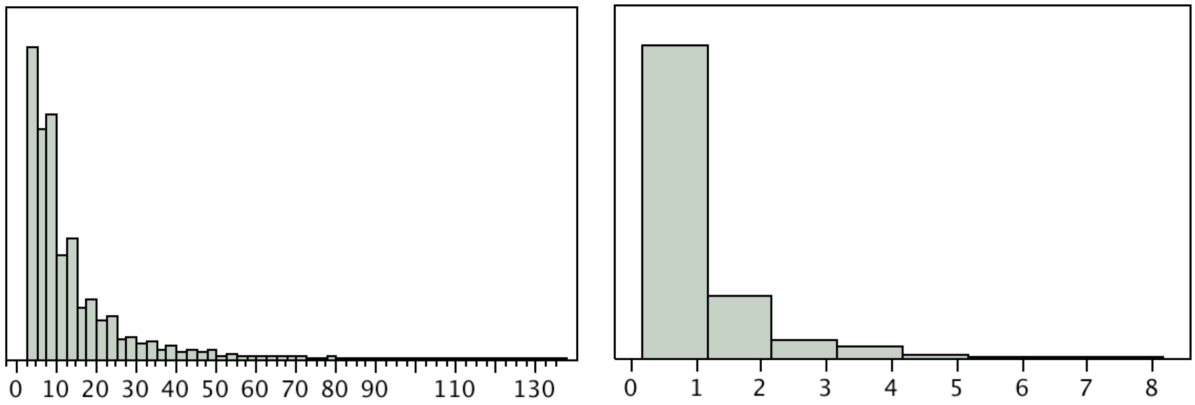

Figure 12. Design-time (left) and run-time (right) parameter frequencies.

The first thing that we note is that the majority (76%) of pipes offer users no run-time parameters at all. In the cases of both design-time and run-time parameters, we again see the binomial coin-tossing model (see fig. 10). However, when compared to each other (fig. 12), there does not appear to be an obvious relationship that would enable us to predict the number of run-time parameters given the number of design-time parameters. To better visualize this, we grouped ranges of design-time parameters (15 per bin), and then plotted the number of pipes corresponding to each binned design-time parameter count and run-time parameter counts as bubbles; it appears that these selection processes are independent (fig. 13).

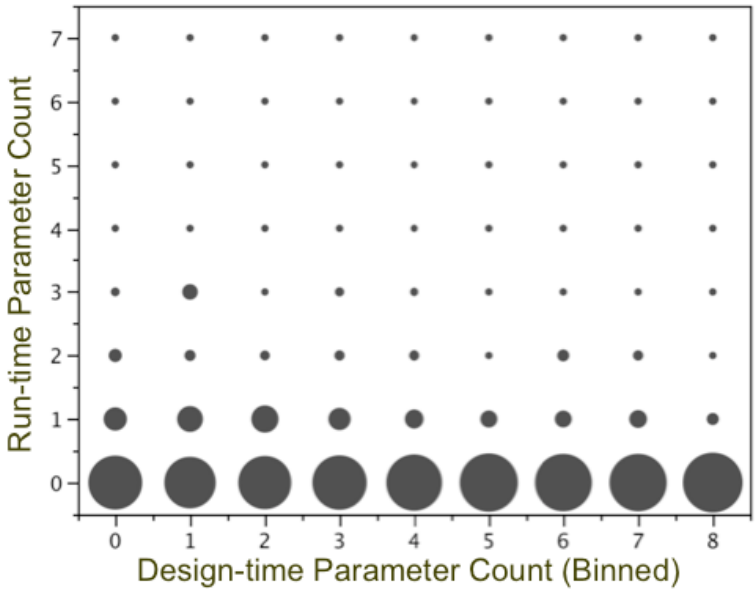

Figure 13. Run-time parameters as predicted by design-time parameters.

5. Discussion

In our view, it would be premature to aim for deep conclusions about end-user programming behavior from the empirical results offered here. Pipes is a specialized EUP environment, and we clearly need

to compare our findings with those from other domains operating at similar scale. Nevertheless, certain principles do seem to govern how end-users approach Pipes:

First, and most obviously, users employ only a small number of the programming options that Pipes offers to them,. They regularly make use of perhaps one-quarter of the available primitives and typically compose them in pipes consisting of only three or four modules. Further, while they are able to create more complex, multi-branch pipelines, they strongly favor simple, straight-through pipes.

Second, we repeatedly see behaviors that fit simple models quite well. In particular, the recurring "coin tossing" model suggests that the decision process for adding a module or interface parameter to a pipe is invariant and independent. This design behavior seems to reflect a process in which, in order to achieve a particular objective, users choose modules from those available, at each step determining if they have or have not achieved the objective (the metaphorical coin toss). The result is what would appear to be the simplest working solution to the problem, evidenced by the strongly linear structure of pipes, in effect, leaving no real choice for wiring. Similarly, users set design-time parameters – as few as possible – to achieve an objective, again asking the question, “am I done?” Decoupled from this is the choice of parameters to expose at run-time. Usually, users simply hardwire the pipe and offer no run-time parameters, though when they do, we again see the coin-toss behavior.

5.1 Future Work

Our study of the Pipes data has only looked at the surface aspects of compositional behavior. There are several other research questions that we propose could be answered by this data. First is a deeper study of the nature of subpipes, clones and other reuse behaviors. Initial research we have done shows that it is possible to identify the lineage of individual pipes to form "families" of related pipes; we have found that the cloning metadata associated with a pipe is an unreliable indicator of actual cloning behavior.

Additional design-time information available to us will allow us to examine questions such as the order in which a user chose modules to place in the workspace, and the spatial organization of pipes. The first may offer insights into intent and planning, while the second may show the evolution of the design and whether the available screen space is a constraint, despite the availability of an essentially unlimited canvas.

We also collected other use-time information, such as the run count for each pipe; this was collected longitudinally, so we can examine the relationship between pipe design (as studied here) and "popularity" at an instant and over time.

Of course, we have completely ignored the semantics of the pipes involved; what was the user trying to accomplish? Information available in tags associated with the pipes, text content, and perhaps even the use of certain modules, could all be used to categorize pipes. An interesting question is whether any particular category exhibits significant differences from the aggregate behaviors we have described here.

6. Acknowledgments

The authors would like to thank our colleagues Rose Daley and Peter Li for their contributions to our study of the Pipes data.

7. References

Blackwell, A. (2002). *First Steps in Programming: A Rationale for Attention Investment Models*. Paper presented at the IEEE Symposia on Human-Centric Computing Languages and Environments.

Bogart, C., Burnett, M., Cypher, A., & Scaffidi, C. (2008). *End-User Programming in the Wild: A Field Study of CoScripter Scripts*. Paper presented at the IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC '08), Herrsching am Ammersee, Germany.

Burnett, M. (1999). Visual Programming. In J. G. Webster (Ed.), *Encyclopedia of Electrical and Electronics Engineering.* New York: John Wiley & Sons.

Fisher, M., & Rothermel, G. (2005). *The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms*. Paper presented at the Proceedings of the first workshop on End-user software engineering, St. Louis, Missouri.

Gantt, M., & Nardi, B. A. (1992). *Gardeners and gurus: patterns of cooperation among CAD users*. Paper presented at the SIGCHI conference on Human factors in computing systems, Monterey, California, United States.

Jones, M. C., & Churchill, E. F. (2009). *Conversations in developer communities: a preliminary analysis of the yahoo! pipes community*. Paper presented at the The fourth international conference on Communities and technologies, University Park, PA, USA.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv*, *37*(2), 83-137.

Ko, A., Abraham, R., Beckwith, L., Blackwell, A. F., Burnett, M., Erwig, M. et al. (2008). The State of the Art in End-User Software Engineering. *ACM Computing Surveys*.

Myers, B. A., Ko, A. J., & Burnett, M. M. (2006). *Invited research overview: end-user programming*. Paper presented at the CHI '06 extended abstracts on Human factors in computing systems, Montréal, Québec, Canada.

Nardi, B., A., & Miller, J., R. (1990). *An ethnographic study of distributed problem solving in spreadsheet development*. Paper presented at the Proceedings of the 1990 ACM conference on Computer-supported cooperative work.

Repenning, A., & Ioannidou, A. (2006). What Makes End-User Development Tick? 13 Design Guidelines. *End User Development,* 51-85.

Whitley, K., N., Novick, L., R., & Fisher, D. (2006). Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility. *Int. J. Hum.-Comput. Stud*, *64*(4), 281-303.