PipeFitter: A SYSTEM FOR CONSTRUCTING PIPELINES

Gary Perlman
Lynn M. Snider

Department of Computer and Information Science
The Ohio State University, Columbus, Ohio, USA 43210
614-292-2566
perlman@cis.ohio-state.edu

ABSTRACT

PipeFitter provides users with a platform-independent intuitive user interface to pipeline construction, thereby increasing the ability of users to compose commands that were previously the exclusive domain of experienced users. It is designed to allow the economical (even automatic) development of pipeline environments that provide task-oriented help with ready-made applications. Yet PipeFitter allows a high degree of customization if it is needed. Dynamic feedback that coordinates control-panel input with command-line input allows multiple modes of specifying tool options. The direction of dataflow in pipelines is specified by vertical positioningof tool dialogues, and is further reinforced by dynamic feedback during execution.

KEYWORDS Pipelines, Dataflow, Direct manipulation, Graphical interaction, Gulf of evaluation and execution, Model-based design, Portability, User interface specification, Command language, Control panel

Computing reviews category:

H.5.2: User Interfaces - Interaction styles

Figure 1. Pipeline from program A to B to C.

INTRODUCTION

An Overview of Pipelines

One of the most elegant and powerful features of UNIX systems is the ability to construct complex pipelines from simple tools. A *pipeline* is a sequence of commands in which subsequent programs read data produced by preceding programs. This is depicted in Figure 1 in which the output from program *A* is directed to be the input to program B, the output of which is made the input to program C. Rather than create intermediate files, a user constructs a pipeline command and the data communication is handled by the system. Although pipelines are usually associated with UNIX systems, they are also part of DOS systems, and the concept of a linear series of transformations is useful on *any* computing system.

There are problems with pipelines. Although elegant in design, our intuition is that pipelines are underutilized for several reasons. *A* major factor is the large amount of prerequisite knowledge needed for their construction. First, a problem must be decomposed into subproblems that are solvable by individual tools. This in turn requires a good knowledge of the existence, purpose, and basic functions of tools. This may not be easy, because many tools are only useful in pipelines with others, making incremental learning difficult. Then, even after users know what tools to use, they must know how to use the individual tools and the syntax of pipelines to integrate them. For example, to run off documents in the **troff** typesetting language, it would not be an easy task for a novice to determine that the correct pipeline for a specific document might be:

eqn myfile ¦ pic ¦ tbl ¦ troff -me ¦ lpr -Plw323

Experienced users can avoid memorizing such complex incantations by saving them in files of commands, but that requires the skills to create parameterized functions.

pipelines are Useful and Sometimes Usable. With so many impediments to overcome, it should not be surprising that pipelines are underutilized. But, despite these indications, we have good evidence that users with little or no programming experience can and do construct pipelines in UNIX and DOS environments. The ¦STAT data analysis package [9] makes heavy use of pipelines to connect data analysis tools, and these programs are used in hundreds of universities for basic instruction. We think people can readily adapt to the idea of pipelines of statistical routines because statistics is a *natural* application for pipelines in that analyses can

usually be viewed as a series of transformations followed by graphical display, descriptive, and inferential statistics. Hutchins et *al* **[5]** and Miller *et al* **[6]** both describe graphical interfaces to pipelines of statistical commands, and the S system for data analysis was originally based on a model of pipelines, even though the user interface was a functional command language [2]. A survey of ISTAT users showed that people used ISTAT because it was readily available on UNIX and DOS, and that analyses could be done quickly in a familiar environment. However, most respondents complained about having to recall how to use program options and the difficulty of integrating programs into long pipelines.

Previous Pipeline Construction Systems

Several researchers have addressed the problem of connecting simple tools to accomplish complex operations. Hutchins et *al* [5] proposed a direct manipulation interface to a statistical system. In it, users would select data manipulation, analysis, and display tools, and connect then to form larger analyses. In their discussion, Hutchins *et al* argued that a direct manipulation graphical interface could improve system usability by reducing the articulatory and semantic distances between users' goals and the state of the activity. In a model of stages of action, Norman [7] and Hutchins et *al* [5] recommend reducing articulatory and semantic distances in the gulfs of execution and evaluation (see Table I). This model of interaction suggests areas where system design can improve usability — areas that might otherwise be overlooked without an organizing model. This work has been influential in the development of systems for connecting tools. After a discussion of relevant systems, we will discuss the design of our system, PipeFitter, and how it addresses usability and scalability concerns.

VSTAT **[6]** allows users to construct complex statistical tools from basic ones by drawing dataflow diagrams. Datasets are represented as objects and are connected to the tools via input and output ports. Data and tools are connected with data flowing from left to right on the screen.

Squish **[4]** is a graphical shell for UNIX that lets users select tools from a menu of commands with meaningful descriptions. Users select tools and interact with customdesignedcontrol panels to supply the parameters to commands. The completed control panels are arranged on the screen left-to-right to connect the tools. The created command is shown in an editable window.

ConMan **[3]** supports interactive application construction with a visual programming language that allows the connection of multiple input/outputs among several active processes. Like VSTAT, ConMan uses the dataflow metaphor, but unlike VSTAT, the connections among components are multidirectional, and are designed for interactive control of *active* processes.

IShell [1] is a visual UNIX shell that promotes a visual programming language in which information processing machines are arranged on a desktop. Machines are depicted as icons, and inputs and outputs are objects that can be dropped into machines for processing. When a machine is activated, it prompts for a limited set of parameters.

Common Themes and Unsolved Problems

In the systems described, there are several common themes and some unsolved problems. These, along with the work by Norman **[7],** have influenced our approach to the design of PipeFitter.

Scalability. The first unsolved problem that we would like to address is that of *scalability,* the ability to solve realistically large problems (e.g., manage over 100 tools). There are several dimensions of scalability:

The Effect of Scale on Usability. **A** system that provides a good interface to a few tools may not provide a good user interface to many tools. We would expect all of the systems described to have increasing usability problems as the number of tools increased, mainly because they view all commands as part of a single set and have no way of grouping sets of related commands. Two systems described did not attempt to address this problem: VSTAT was designed to only apply to one set of related commands for statistics, and ConMan was designed to develop interactive graphical controls. The single menu of descriptive phrases in Squish and the customdesigned icons in IShell would both develop usability problems (e.g., visual search and identification) as the number of tools increased.

The Effect of Scale on Development Costs. If a system requires that existing work must be discarded, or that a great deal of highly skilled work is needed, then the acceptability of the development of a new system is lowered. Ideally, existing tools (e.g., on UNIX) could be used in a graphical environment with no changes to those tools, and relatively low cost to provide an improved interface. A common criticism of graphical programming, particularly interactive graphical programming such as programming-byexample, is that the technique does not scale-up to larger systems. For most of the systems discussed, it is necessary to modify existing tools to provide a graphical interface; this would necessarily hinder development. The notable exception is Squish, that uses a separate specification to define the interface to individual commands. *ALL* of the systems (except IShell, which simply prompts for a limited set of parameters) use control-panels or ported-icons that are customdesigned. This requirement makes it difficult to add new functionality, and impedes the use of many

classic pipeline tools in a graphical environment because of high development costs. For simple tools (e.g., UNIX wc, head, tail) or simple uses of moderately complex tools (e.g., UNIX **sort),** it is possible to automatically generate control panels [10,11]. Arguably, custom interfaces are needed for even limited subsets of some functions (e.g., UNIX tools that take expressions as parameters, such as grep, awk, and **sed).**

Stages of Action. Norman's [7] stages of action suggest a series of problems — gulfs of execution and evaluation — that users must bridge to act effectively. Although the systems described above provide some help, there are areas in which more help can be provided. The follow areas move from the formation of goals, across the gulf of execution, and over the gulf of evaluation. Table I shows Norman's stages and the problems with pipelines that occur at each stage. Later, we will discuss how PipeFitter addresses each of those problems.

Task-Oriented Problem Decomposition. None of the systems provide high-level task-oriented help with problem decomposition by providing examples of how groups of related commands can be integrated. To promote system use, special-purpose toolkits (e.g., statistics, text formatting, bibliography use), examples of their use (existing pipelines), and descriptions of example pipelines are all critical. Toolkits (or ***workbenches*** [8]) reduce the amount of knowledge required and reduces the number of choices users must make. Initial examples can be used as ready-made applications that help users get started.

Tool Compatibility. None of the systems provide tool-compatibility help by trying to determine if the output format of one tool matches the input format for a subsequent tool. On UNIX systems, most tools have few requirements (e.g., textual ASCII input) but subsets of programs have stricter input requirements (e.g., tabular/numerical, special languages, bibliographic records), which could be used to suggest relevant tools or provide feedback to users.

Tool Identification. Most of the systems try to help users identify tools to make them more recognizable than, for example, standard UNIX names. Squish presents meaningful phrases, VSTAT uses meaningful phrases to annotate a few different types of icons, while IShell presents a customdesigned icon for each tool (although with uncertain identifiability). Descriptive phrases and icons are useful to identify the purpose of tools, but these aids may need to be customized for different application areas because the same tool may have different uses in different contexts.

Full Functionality. All of the systems provide a simplified interface to handle parameters (options and operands), in part because the parameters are presented in custom-designed control panels, in input slots (which require longer development time), or in serially presented prompts (which could require extensive user interaction). Simplified interfaces, however, preclude providing full functionality, particularly to advanced users. It should be possible to design a usable interface that still allows full functionality for expert use.

Dual-Mode Command Composition. Only Squish presents the constructed command to users and allows them to edit it, however, the changes to the textual command are not then reflected in the control panels. It is necessary to provide dual-mode command composition, to allow novices to see the command constructed, but also importantly, to let experienced users supply options directly to a command. Users should be able to change the control panel and immediately see the changes in textual commands and ***vice versa.*** Allowing textual input of (or addition to) commands is particularly desirable if the control panel interface is simplified, leaving some parameters only available via the command line. To allow such dual mode editing, it is necessary to have a separate specification of parameters and their attributes; users actions change the separate specifications through the views and these changes are propagated back to both control panel and textual views. This specification is also useful for generating character-based and graphical interfaces automatically [10], which is an important issue of scalability.

Tool Layout. The layout of tools in the display is one-dimensional (left-to-right) for Squish, and two-dimensional for the other systems, although in the case of IShell, the 2D layout is unnecessary for one-dimensional dataflow. For systems with multiple inputs and outputs, the dataflow among tools can be confusing, particularly for complex applications (e.g., ConMan can have eight inputs and eight outputs per process). Systems can and should simplify layout needs if only simple dataflow is involved (i.e., pipelines).

Tool Connection. All of the systems (except Squish) connect tools with directed lines. Based on the experiences of Miller ***el at*** with VSTAT, we think that users might require training to learn how to connect the tools; this is contrary to the basic design goal of making it easy for users to construct pipelines. For one-dimensionaldataflow, directed lines may be unnecessary. Taken together, the issues of tool layout and connection can be addressed simultaneously for pipelines. We think that implicitly ordered connections by ordering tools along a spatial dimension is intuitive. Squish's horizontal layout (which matches pipeline command-line syntax) makes it difficult to construct large pipelines. This scaling problem makes us conclude that inferring

pipeline connections from vertical layout would be the best solution, particularly if critical information is at the top of tools.

Feedback. Only IShell specifically addresses the issue of feedback of operation and dataflow by providing viewers that can be attached to pipelines, and Borg also discusses the potential use of animation to show active dataflow. Based on experiences with a pipeline monitoring program, io, we concluded that the feedback provided by pipeline monitoring commands is useful. io was written at UCSD around 1981 by the first author and Don Norman. It allows the monitoring of the percent or absolute amount of data processed, for each tool in a pipeline. io also provides a file data ***pump*** for the start of pipelines (similar to UNIX **cat),** a mid-pipeline redirection facility (similar to UNIX ***tee),*** and a data storage ***sink*** for the end of pipelines (with safe overwrites of files). The functionality of io, along with a viewer to examine the ***content*** of pipelines, should be generic in any pipeline construction system, and should be integrated into pipelines like other tools.

Design and Implementation Goals

Based on the discussion above, we developed specific design and implementation goals for a pipeline construction system. These were derived from two general concerns: (1) to make it easy for users to create and use pipelines; and (2) to make it possible to ***scale up*** the development to include potentially hundreds of tools. The concern for usability led to the following goals to:

- be easy to use by novice users;
- allow the reuse of pipelines over long periods;
- allow continued use by experts;
- provide good feedback of operations.

To be applicable to many tools, we constrained our design to:

- require little or no modification of existing code;
- allow potentially automatic construction of control panels (from code or independent specifications);
- be portable to many environments, graphical and non-graphical, and to many hardware/operating system platforms.

It was ***not*** our goal to fix UNIX. Pipelines are available in other environments, such as DOS, and there are several command shells available on Apple Macintosh systems. Rather, we viewed the problem as one of integrating, using, and reusing loosely coupled tools.

It ***was*** a goal to allow pipelines in a graphical user interface because we wanted to make the capabilities of pipelines and their component tools readily available in graphical environments, such as the Apple Macintosh, in which non-graphical interfaces would be rejected.

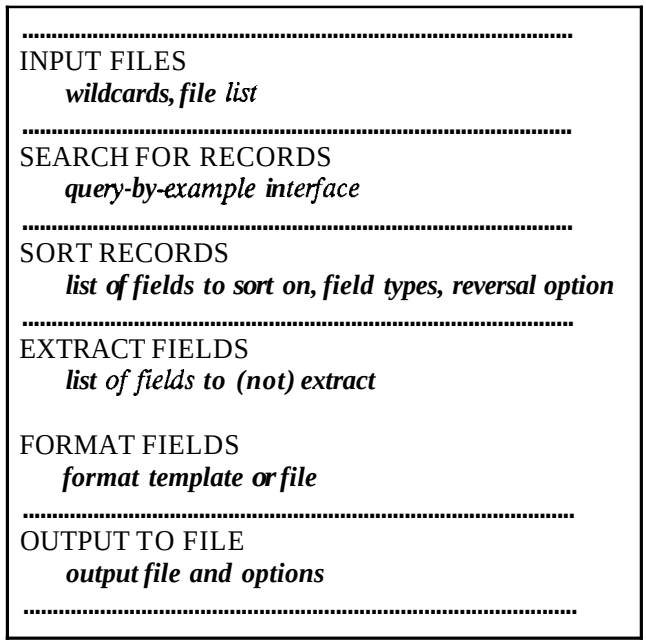

Figure 2. A schema for form-based interface for information retrieval.

THE DESIGN OF PipeFitter

PipeFitter is a system to reduce the articulatory and semantic distances in the construction and monitoring of pipelines of commands. The design of PipeFitter is based on a metaphor of a form that is filled from top to bottom, as is depicted in Figure 2. There are fields in sections that correspond to parameters to individual operations. ***An*** early design of a bibliographic retrieval environment for [12] presented a form with the series of transformations from input to search to sort to format to display or output. Although the layout in Figure 2 was thought to be adequate for many uses, there were other tools (e.g., to ***add*** or ***merge*** fields) that were not included, and there were realistic cases where the order of operations might be different (e.g., fields might be extracted before or after search). The desire for flexibility of ***which*** commands were to be used and in ***what*** order, led to a generalization of the form that allows the construction of pipelines of any commands (e.g., standard UNIX filters, data analysis programs [9]). The idea that led to the design of PipeFitter was to cut up the form into separate tools, while maintaining the overall form-filling interface, and be able to construct command sequences by making selections from toolkits.

Table I is based on Norman's [7] seven stages of action and the design questions that can be asked to help improve usability at each stage. In Table I, we summarize some of the problems that users would encounter with pipelines at each stage, and the way we addressed each in the design of PipeFitter.

An example PipeFitter screen is shown in Figure 3. It

Table I. Stages, Problems, and Solutions

Stage of Activity (Design Questions)	Problems with Pipelines	PipeFitter Solutions
1 Forming the goal (Can the user determine the function of the device?)	lack of problem decomposition knowledge	pipeline descriptions identify the purpose of pipelines
2 Forming the intention (Can the user tell what actions are possible?)	lack of awareness the existence of tools; unknown capabilities and compatibility of tools; obscurity of tool names	saved pipelines provide menus of related commands with descriptions of tools
3 Specifying an action (Can the user determine the mappings from intention to physical movement?)	lack of knowledge of how to use, combine, and run tools	familiar widgets; pipelines constructed by positioning; instructions in pipeline descriptions
4 Executing the action (Can the user perform the action?)	typing errors, syntax errors, lack of help on commands, data entry errors	control panels simplify interaction
5 Perceiving the system/world state (Can the user tell what state the system is in?)	lack of feedback of which tools are connected, which are active, and what they are doing	tool position implies direction of dataflow; active tool is brought to foreground; viewers can be inserted in pipeline
6 Interpreting the system state (Can the user determine the mapping from system state to interpreting the outcome?)	difficulty of looking at the outputs/inputs of commands at each point in the pipeline	pipeline viewers allow examination of intermediate steps; meters give feedback
7 Evaluating the outcome (Can the user tell if the system is in the desired state?	lack of dynamic feedback of effect of actions	command lines and control panels are coordinated

contains a pipeline to search for, sort, and extract fields from records in a database of bibliographic records, and is based on a pipeline stored in the format shown in Figure **4.** Although PipeFitter is designed with a toolkit philosophy, we thought that by providing ***canned*** applications (i.e., stored pipelines) such as the one in Figure 3, it would help users learn the tool-combination concept, and if not, then at least users would have useful applications. The availability of example pipelines should increase users' knowledge of tools and should help them by providing examples of ***problem decompositions*** known to be effective.

Figure 3 shows a pipeline of five tools. Two of the tools are from a small set that are built in to PipeFitter: the **input** tool, the **viewer** tool, the **output** tool, and the **generic** tool. Pipelines are opened and saved from the **File** selection in the menu bar. When a tool dialogue is inserted from the Tools menu, it is placed on an implicit diagonal grid; then the tool can be moved. The **View** selection controls the arrangement of tool dialogues on the screen (they can be automatically arranged along the implicit grid) and the display of the ***pipeline description*** (it can be displayed and edited at any time). The **Execute** selection runs the current pipeline. Figure 3 shows the display immediately after opening the pipeline file **init.pip**. Pipeline files are plain text that contain three sections: (1) The ***description*** is placed at a user-controlled X-Y coordinate, and may be displayed automatically when the pipeline is opened; (2) The ***commands*** section contains the **X-Y** locations of the tools, along with any options for those tools; (3) The ***tools*** section contains the names of all tools available to build a new pipeline, and the descriptive phrases for the tools in the pipeline. Figure 6 is an example of a saved pipeline for another environment, the troff command in the introduction to this paper.

When a pipeline is opened, the tool options are parsed and propagated to the appropriate parts of the control panels, and the tool dialogues are named and displayed at the saved locations. The pipeline description can contain instructions about how to execute the pipeline. To execute the pipeline, users set parameters in the tool dialogues and select **Execute** from the menu bar. The tools are executed in the order of their vertical position.

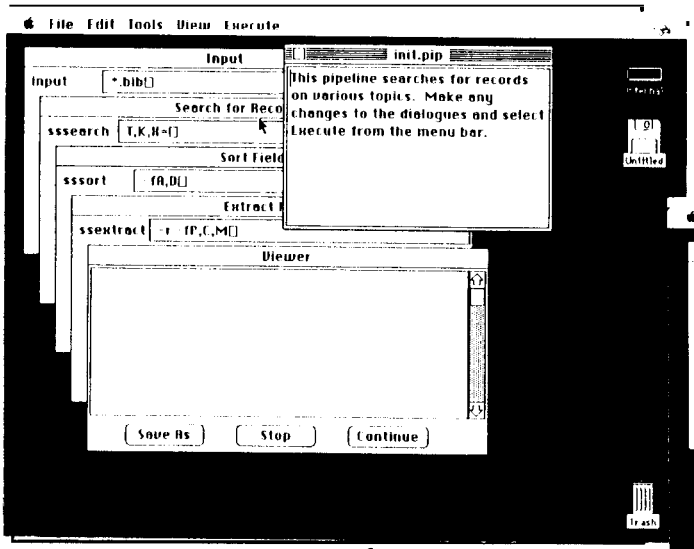

Figure 3. A PipeFitter display for a stored pipeline.

```
@Description ----------------------
 264  46 AUTO
This pipeline searches for records
on various topics.  Make any
changes to the dialogues and select
Execute from the menu bar.
@Commands -------------------------
  20  50 input *.bib
  35  95 sssearch T,K,X=
  50 140 sssort -fA,D
  65 185 ssextract -r -fP,C,M
  80 230 viewer
@Tools(input output viewer generic)
ssadd      Add Fields
ssextract  Extract Fields
ssforder   Order Fields
ssformat   Format Fields
ssmerge    Merge Fields
sssearch   Search for Records
sssort     Sort Fields
```

'igure4. Pipeline file for Figure 3.

Figure 5 shows the result of some editing on Figure 3: some tool dialogues have been moved, a tool to order fields has been inserted, parameters have been filled in, and the pipeline has been executed. As the pipeline was running, each tool's dialogue was brought to the foreground as it was activated; this provided feedback of the state of the execution. Although the layout in Figure 5 is hard to parse (it is that way to show parts of control panels), it is easier to follow when you are the one making the changes. It is always possible to have PipeFitter reorganize the display so that the tool dialogues follow the implicit grid. In Figure **5**, parts of the control panels for extract and **order** are shown. The one for field extraction is a custom dialogue that presents more descriptive fields names for options and a checkbox for exclusion of fields. As options are changed in the control panel, the command line changes, and *vice versa*. **At the bottom of Figure 5 is the result of the pipeline in the generic viewer tool, which can be inserted anywhere in a pipeline. The viewer can save its contents and it can stop the pipeline.**

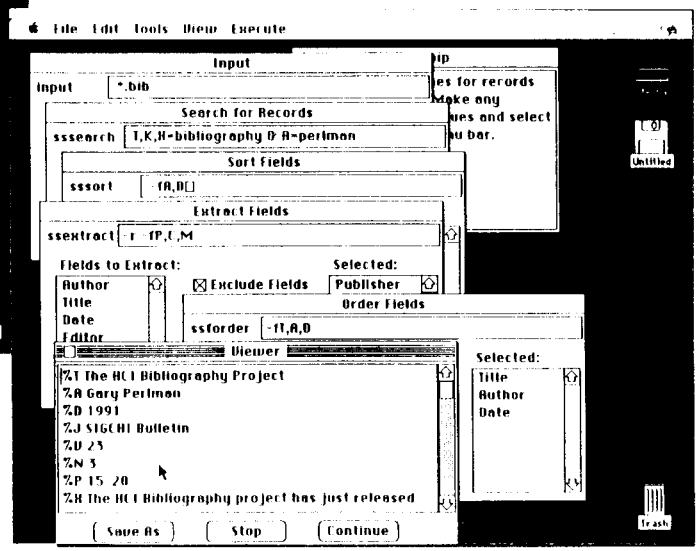

Figure 5. A PipeFitter display after editing and execution.

```
@DESCription
265  50 AUTOmatic
This pipeline will let you run off
a manuscript file written in the
troff formatting language.
@COMMands
 20  50 input
 35  95 eqn
 50 140 pic
 65 185 tbl
 80 230 ptroff      -me
 95 265 lpr         -Plw323
@TOOLs
ptroff   Postscript Runoff
eqn      Equation Processor
pic      Picture Processor
tbl      Table Processor
lpr      Laser/Line-Printer
```

'igure 6. A saved pipeline for the troff command.

IMPLEMENTATION OF PipeFitter

PipeFitter is implemented in C using the XVT (Extensible Virtual Toolkit) library. XVT allows the development of one system that is portable to many graphical and non-graphical environments. The first implementation of PipeFitter is on a Macintosh, where tools were implemented as code resources.

Underneath the portable user interface of PipeFitter is a portable application connection environment. UNIX-compatible systems, with few exceptions, support

pipelines with memory buffers that are written by one process and read by another. Such operating system functions are not available on other systems, such as DOS, Macintosh, or **VMS.** DOS implements pipelines with temporary intermediate files, hidden from the user. For environments like DOS and Macintosh, PipeFitter creates temporary files and manages the execution of the sequence of commands in pipelines. This has some performance implications, but allows basic pipeline functionality in all environments.

There are some user interface implications of allowing pipelines on single-process systems like DOS and Macintosh versus multi-processing systems like UNIX. On systems like DOS and Macintosh, PipeFitter executes commands in pipelines strictly sequentially, while on UNIX systems, the processes can be concurrently active and can begin to read input as soon as it is made available by preceding processes. Therefore, the feedback from an asynchronous pipeline is more complex than that from a sequential pipeline. For example, a sequence of commands might be 80%, 40%, and 20% done on a UNIX system.

There are three strategies for the development of control panels for tools. These trade off user expertise against development cost.

1. Control panels are hand-crafted and may involve iterative design. This strategy is used for tools that have complex options that require help to supply (e.g., a Boolean search expression). This strategy requires a large investment of time and effort for each command.
2. Control panels are generated automatically from specifications of programs and their options. Perlman [10] describes a system, **setopt**, for generating textual and form-based user interfaces to UNIX commands based on a specification of commands and options. For each option, its single-character ***flag,*** internal ***variable,*** brief ***description,*** data ***type, default*** value, ***size*** (scalar or vector), ***range,*** and implied ***actions,*** are specified in a tabular format. From a database of option specifications, a variety of user interfaces can be generated automatically from code templates. We are planning to investigate automatic intelligent layout of the options based on an axiomatic model of information presentation [11]. This strategy requires a minimal investment for each command, but it cannot be used for all commands, and may require more knowledge of commands and options by users.
3. A generic control panel allows users to supply a command name and its arguments. This strategy requires almost no investment for each command, but requires detailed knowledge by users.

There is one notable exception to the coordination of textual options and graphical specification of options. Any options that are not known to the graphical option specification manager are maintained at the end of the textual option field, for use when the tool executes. This makes it possible to design simplified user interfaces to commands, yet still allow full use of options by knowledgeable users.

CONCLUSIONS

PipeFitter was designed to help users span the gulfs of execution and evaluation encountered when constructing pipelines of commands. The concept of a pipeline can be conveyed intuitively, and supported by software, even on systems with which pipelines are not ordinarily associated. The obstacles of prerequisite knowledge needed to construct pipelines can be bridged in simple cases by providing ***canned*** applications that reduce the use of typical pipelines to a task of form-filling. Obstacles of construction of new pipelines can be bridged by reducing the task to one of modification. Dynamic feedback helps bridge the gulf of execution.

PipeFitter provides users with a platform-independent intuitive user interface to pipeline construction, thereby increasing the ability of users to compose commands that were previously the exclusive domain of experienced users. It is designed to allow the economical (even automatic) development of pipeline environments that provide task-oriented help with ready-made applications. Yet PipeFitter allows a high degree of customization if it is needed. Dynamic feedback that coordinates control-panel input with command-line input allows multiple modes of specifying tool options. The direction of dataflow in pipelines is specified by vertical positioning of tool dialogues, and is further reinforced by dynamic feedback during execution.

Our current plans for PipeFitter are to complete the port to other windowing environments (i.e., DOS and UNIX) and to gather long-term feedback on its use, particularly with statistical [9] and bibliographic [12] applications. There are also several enhancements that we discussed, but have not yet explored:

- better interface code generation from specifications;
- checking the compatibility of data formats between tools based on specifications, and suggesting when certain tools may be more or less appropriate;
- multiple percent-done/flow indicators attached to tool dialogues that are executing in parallel;
- richer feedback about the state of tools (whether they are reading, processing, or writing), what resources they are using (e.g., CPU), and finer-grained metering (e.g., number of ***records*** processed), perhaps using sound;
- saving command histories for later reuse;

ACKNOWLEDGEMENTS

This work was funded in part by The Ohio State University and Apple Computer, which in no way should be taken as an endorsement of this work.

UNIX is a registered trademark of AT&T. VMS is a trademark of Digital Equipment Corporation. Macintosh is a registered trademark of Apple Computer. XVT is a trademark of XVT.

REFERENCES

1. Borg, K. IShell: A Visual UNIX Shell. In **Proceedings of CHI'90 Conference on Human Factors in Computing Systems,** ACM, New York, **1990,** pp. **201-207.**

2. Becker, R. A. & Chambers., J. M. Design of the S System for Data Analysis. **Communications of the ACM, 27:5,** pp. **486-495, 1984.**

3. Haeberli, P. E. ConMan: A Visual Programming Language for Interactive Graphics. In Proceedings of SIGGRAPH'88 Conference on Computer Graphics, **Computer Graphics, 22:4,** ACM, New York, **1988,** pp. **103-111.**

4. Henry, T. R. & Hudson, S. E. Squish: A Graphical Shell for UNIX. In **Proceedings of Graphics Interface** '88, **1988,** pp. **4349.**

5. Hutchins, E. L., Hollan, J. D. & Norman, D. A. Direct Manipulation Interfaces. In D. A. Norman & S. W. Draper (Editors) **User Centered System Design,** 87-124, Lawrence Erlbaum Associates, Hillsdale, NJ, **1986.**

6. Miller, J. R., Hill, W. C., McKendree, J., Masson, M. E. J., Blumenthal, B., Terveen, L., & **Zaback,** J. The Role of the System Image in Intelligent User Assistance, In **Proceedings of INTERACT'87, 885-890, 1987,** North Holland, Amsterdam.

7. Norman, D. A. **Cognitive Engineering.** In D. A. Norman & S. W. Draper (Editors) **User Centered System Design, 33-61,** Lawrence Erlbaum Associates, Hillsdale, NJ, **1986.**

8. Perlman, *G.* MENUNIX: A Menu Interface to UNIX Files and Programs. **Proceedings of the 1982 Summer USENIX Conference.** USENIX Association, El Cerito, California, **1982.**

9. Perlman, **G. &** Horan, F. L. Report on UNIX¦STAT Release **5.1** Data Analysis Programs for UNIX and MSDOS. **Behavior Research Methods, Instruments, & Computers, 182,168-176,1986.**

10. Perlman, G. Multilingual Programming: Coordinating Programs, User Interfaces, On-Line Help, and Documentation. **Proceedings of the ACM SIGDOC Fourth International Conference on System Documentation, 1986,** pp. **123-129.**

11. Perlman, **G.** An Axiomatic Model of Information Presentation. **Proceedings of the 31st Annual Human Factors Society Meeting.** Human Factors Society, Santa Monica, CA, **1987,** pp. **1229-1233.**

12. Perlman, G. The HCI Bibliography Project. **ACM SIGCIII Bulletin, 23:3,** pp. 15-20, **1991.**