Towards More Intelligent and Interactive Interfaces

James S. Jennings and Nicole D. Terry
Department of Electrical Engineering and Computer Science
Tulane University, New Orleans, LA 70118
{jennings|terry}eecs.tulane.edu

Abstract

Our goal is to support the development of more intelligent and more interactive text-based interfaces for general computer use. In order to enable automation of user tasks, we find a new command shell architecture is needed. Our working prototype, *Fish*, maintains a global, persistent knowledge repository across concurrent interactive sessions. For example, a user can define a shell function in one session and this function is immediately available to the user's other concurrently running sessions, as well as future sessions. Our approach is to centralize knowledge in a persistent way while parallelizing execution. *Fish* is extensible and customizable, with a full programming language. It supports efficient communication between the machine and the user in many ways, most notably by giving the user access to the results of previously executed commands. Indexical references into previous results are possible by using a sophisticated command language which contains a suite of syntactic and semantic analysis functions. Significant strides in learning are possible due to the centralization of knowledge and its persistence. We illustrate the possibilities with an example of how learning may be uniquely applied using *Fish*.

Introduction

Intelligent interfaces hold the promise of freeing computer users from routine drudgery, from the mental maintenance of staggering amounts of arbitrary information, and also from the need to learn arcane programming methodologies such as scripting languages, programs like Unix's `cron`, etc. Our focus is on general use of modern networked computers – the kind of activity currently accomplished through a text-based command shell. This is a challenging domain, with many degrees of freedom. On a typical Unix workstation, such as one in our department, there are approximately 1,700 different programs one can run; there are over 250,000 files and directories on our small network; our 22 public workstations are fairly homogeneous, but most of our 70 private machines have either local file systems, special devices attached, or both. When one considers what is accessible over the Internet, the numbers above increase astronomically.

Granted, many users of networked workstations make little use of the programs, data, and other local machines which are available. However, certain users *will* use a large variety of programs, access an enormous amount of data, and execute programs on several host machines at once. System administrators, programmers, students, professors, and so-called "power users" fall into this category. Due to the complexity of their tasks, text-based computer interfaces are almost unavoidable because they allow the user to express *in language* concepts difficult to express by direct manipulation of icons and menus. Thus, the "command shell" survives despite the ubiquity of icon-based (graphical) direct manipulation interfaces such as Microsoft Windows in the PC domain and X Windows in the Unix domain.

In order to increase the efficiency with which people use text-based command shells, we need to increase the level of automation these interfaces provide. Examples of automation in existing command shells include *tab completion*, in which a partially typed file name is completed by the shell, and *shell scripts* which are programs written for the purpose of running other programs. Opportunities for increasing the level of automation abound. For instance, users could more easily incorporate domain knowledge in the form of their own procedures and data for accomplishing repetitive or complicated tasks. Learning programs could study the user's activity and provide guidance, suggestions, or even develop procedures which automate or abstract the user's tasks. Agents can provide a great deal of automation as well, enabling the user to perform remote or delayed computation, and agents can handle some tasks automatically. In our domain, text-based interfaces on modern networked computers, implementing any of these approaches to automation is challenging.

Before we can build an effective learning agent for our domain, or design a facility for the incorporation of domain knowledge, or integrate a mobile agent sys-

tem into the user interface, we need to develop a new command shell architecture. This paper describes an implemented prototype of such an architecture, called *Fish*. Our prototype allows the user to interact with the machine in many concurrent sessions; these sessions can share information such as procedures, history, and other data; agents and other programs can communicate with the user through our system. We illustrate below how a learning agent could observe the user's actions and instantiate a procedure which automates them, despite the fact that the user interacts with the machine through several concurrent sessions.

In the next section we present several important challenges to conventional interfaces, followed by a sketch of the architecture of *Fish* and examples of how *Fish* facilitates the use of domain knowledge and agent technology.

Challenges

Traditional shells, such as the Bourne shell(Bou78) and its derivatives, present many challenges to automation. Their behavior can be modified in only a few predetermined ways (e.g. custom command completion, turning on/off command history, setting the PATH variable). Behaviors such as the order in which substitutions are performed and how wildcards are expanded, and services such as file redirection and pipelining cannot be changed by the user.

Shell languages also present a challenge to automation. These "strange" languages lack the full functionality of traditional programming languages and are typically restricted to string-valued variables. They lack procedural abstraction, modularity, and robust exception handling. Newer scripting languages, such as Perl, alleviate some of these problems, but present others.

The ability to launch persistent agents is also difficult to achieve. If a user would like a sequence of commands to be run autonomously (i.e. while the user is logged out or is doing other work), he must write a script and use the `at` or `cron` commands. Programs backgrounded by the user are generally killed upon user log out by the hang-up signal. This can only be avoided if the user had the forethought run it via the `nohup` command or in a `screen` session.[1]

The incorporation and organization of domain knowledge presents a problem too. With traditional Unix shells, all shell context is stored in the startup files. Shell context includes command aliases, environment and shell variables, shell functions, and command history. If the user wishes to create a function or piece of data that will persist in future shell sessions, he must explicitly write the definition of the item to one of the startup files. Since the shell provides only a global namespace, a unique name must be used for each context item. With a large collection of context items created over time, the startup files can become cluttered and confusion of names can occur.

Attempts to automate and incorporate learning into current shells are further complicated when multiple shells run concurrently. When a startup file is edited, the changes are only accessible to those shells in which the user has re-loaded the file. Also, command history is automatically saved to a specific file upon exit of the shell, so as successive shells exit, this file is overwritten with different a history. Therefore, there is no consistent command history context.

[1]The meanings of these Unix commands are somewhat inessential to the discussion. See the Unix `man` pages for more information.

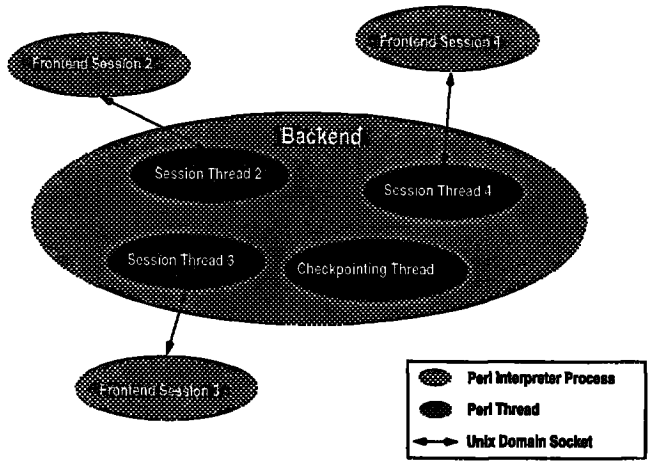

Figure 1: System Architecture: In the figure, three frontends are running concurrently. Each frontend communicates with a session thread in the backend which was specifically spawned to handle its activity.

Architecture

The core of *Fish* is a long-lived process which we refer to as the *backend*. The first time a user uses *Fish*, the backend process starts. In general, it never stops; it is already running and users merely connect to it via a thin client which we refer to as the *frontend*. The user may start as many frontends as desired, all of which connect to the single backend. The backend is multithreaded for concurrent processing of frontend requests. Refer to Figure 1 for a graphical representation of the architecture.

First, a few key definitions:

Session The interaction between a frontend and the backend. The lifetime of the session is the lifetime of the frontend.

Project Environment A project environment contains the context associated with a particular task which changes gradually over time and persists indefinitely.

Session Environments A session environment contains the temporally local context specific to a session.

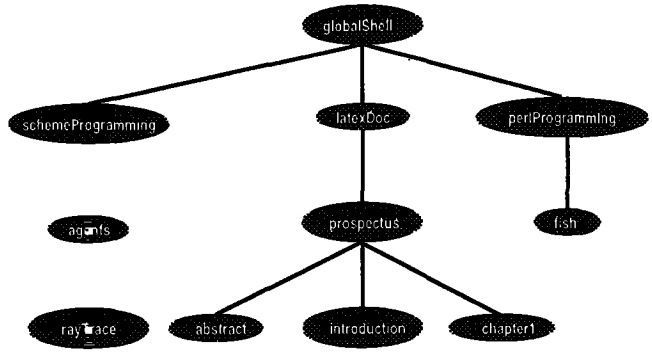

Figure 2: The figure depicts the project hierarchy created by one of the authors in order to subdivide the context of her current tasks. For example, the project *rayTrace* inherits the context of *agents* since it is a particular type of agent. The project *agents* inherits the context of *schemeProgramming* since the agent system was implemented in scheme. Lastly, the project *schemeProgramming* inherits the context of *globalShell* since the author chose to place all context she wished to be globally accessible in that project.

Knowledge Environment

Knowledge shared between two people helps to keep their conversations brief. *Fish* persistently stores knowledge which is shared with the user; we refer to this shared knowledge simply as *context*.

In addition to the context items provided by traditional shells such as variables, shell functions, command aliases, and command history, our system also implements result history. A result is defined as the text sent to both standard out and standard error by a process in addition to its exit value. Result history provides the power to make use of previous results at any time, without requiring the forethought to capture them via file redirection or pipes.

In order to organize a large amount of context accumulated over time, *Fish* maintains a hierarchy[2] of *project environments*. A *project* is a namespace used to store related items. Users create and delete projects, and can switch at any time from one project to another, much as they change directories now. Each project has its own context.

The use of multiple, distinct contexts yields task-oriented namespaces which allow the user to have variables, aliases, and functions of the same name with different values or functionalities in the separate project environments. The project environments are used for locality of reference much as scoping constructs are used in programming. As with programming, our project hierarchy describes nested scopes, as depicted in Figure 2.

Projects and their associated context persist between logins and even across reboots or power outages. In order to be able to recover from reboots and power outages, the backend periodically checkpoints each active project to the file system.

Agent-like Properties

In traditional shells, `cron` jobs and backgrounded processes provide agent-like behavior to a certain degree. Unfortunately, communication with running `cron` jobs and backgrounded processes is limited and difficult to achieve. Any results sent by backgrounded processes to standard out or standard error and not re-directed are lost. Upon completion of a `cron` job, results are emailed to the user.

The persistence of the backend allows for flexible autonomous computation by proxy. While the user is idle or logged out, functions running in the backend still perform useful work. Since the backend runs with the permissions of the user, it can perform the same operations as the user himself. He must merely write and execute a function that performs its task at a certain time, at certain intervals, or when a certain condition occurs. These functions can communicate with the user not only through traditional means such as the file system, signals, email, windowing, etc., but also through the backend message queue and the knowledge environment itself. Means of communication is only limited by the user's imagination and/or programming ability. If a modem is attached to a machine to which that backend has access, the user could even be paged.[3]

Interface Programming Language

Well underway is the design and implementation of a suitable interface programming language for *Fish*. (The current one is Perl.) Commands are processed by *Fish*'s `read-execute-print` loop (repl), roughly analogous to the read–eval–print loops of a Lisp or Scheme interpreter. However, the syntax and semantics of the *command language* accepted by the *Fish* repl are distinct from those of the programming language used to customize and extend *Fish*.

Extensibility By analogy, users interact with Emacs by striking keys bound to Emacs Lisp functions that execute when a key is struck. But Emacs users can also directly write and execute Emacs Lisp code in an interpreter within Emacs. Similarly, *Fish* users type commands in a command language at a command prompt. A small command interpreter translates commands into an *interface programming language (ipl)*. The resulting program is sent to a thread in the backend component of the *Fish* shell for execution; that is, the *ipl* interpreter executes the command. By "escaping" to the *ipl*, users can define new commands, functions, and variables, and

[2]Currently, only single inheritance is supported. Multiple inheritance might in fact be more appropriate here.

[3]We have in fact implemented this feature. However, we are pressed to find a driving need for it.

even alter the way in which commands are interpreted.[4]

For example, one might decide to generalize the concept of a "working directory" to include multiple concurrent working directories. For a project requiring C language programming, the set of working directories might include `/usr/include` in addition to the user's source code directory, e.g. `/home/clinton/server/src`. Multiple working directories permit efficient shorthand because the user can type "`less regexp.h`" or "`emacs my-program.c`" and the shell will be able to find these files by looking in more than one place. The small amount of machinery necessary to extend *Fish* in this way is directly accessible by the user, including the ability to install an exception handler which could, for instance, query the user or signal an error when faced with a filename which exists in more than one of the working directories.

Knowledge Representation Defining new commands and storing data in variables are the mechanisms by which the user can directly encode domain knowledge into the shell as procedures and data. While actual procedures may not be a suitable encoding of procedural knowledge for some applications, they are quite general. In other words, the *ipl* is a complete programming language; hence, suitable abstractions may be built within *Fish* for encoding different types of information.

Agent Interaction Because our goal is to empower the user by infusing the user interface with capabilities for automation, we must consider the needs of users who interact with autonomous software agents, even mobile agents. *Fish* provides facilities for asynchronous two-way interaction between the user and programs, whether those programs are simply backgrounded processes or intelligent, autonomous, even mobile agents. The user-level view of this interface is still primitive, and is under development. It seems clear that extensible syntactic and semantic standards are needed; work on, e.g. KQML (Fea93) proceeds in this direction.

Manipulating Results A primary influence on *Fish* is the notion that real interaction between humans and computers can only take place when the conversation goes two ways. Generally, the user executes commands which produce output. In traditional shells this output is displayed but otherwise discarded. By capturing the output of programs as text and making it accessible to the user, *Fish* allows the user to indexically reference the results of computation.

The command language and interface programming language are currently being redesigned to incorporate an extensible set of parsers which can be used to extract appropriate syntactic units from raw text. For instance, the output of the Unix `finger` command often includes information provided by a user such as their home page address. Applying an appropriate parser to this text would produce a list of URLs based on syntactic analysis. Next, simple *ipl* functions can be applied to determine which URL's are valid. Finally, we observe that the syntactic analysis and the semantic filtering (the "validity" test) may be performed only when needed. In the next version of *Fish*, the user will be able to associate *types* with textual *ipl* function parameters. Thus, when such an *ipl* function is invoked on a body of text, such as the result of the previous command, *Fish* can automatically extract the appropriate information from the text.

For example, suppose the user writes an *ipl* function called **visit** which takes one argument, a URL, and signals a running web browser process to visit that URL. A function prototype for **visit** might look like this: `proc visit(URL: place)`.

If the command language keyword **it** indicated a reference to the result of the previous command, then the following transcript would cause the user's web browser to visit `http://www.whitehouse.gov`:

```
Fish$ cat msgfile
Check out Al's vacation photos,
available through our main
page at http://www.whitehouse.gov!
Fish$ visit it
Address ''http://www.whitehouse.gov'' sent to
netscape.
```

Other Front Ends

Other programs can communicate with the *Fish* backend process and take advantage of its store of knowledge, such as user-defined mechanisms for automatically locating files using multiple working directories, as mentioned above. For example, a small Emacs Lisp program would allow the Emacs editor to interact with *Fish*. Thus the Emacs "find file" function could interact with the user's shell to locate files. Emacs could also display messages that *Fish* relays from agents or other programs. In this respect Emacs functions as a frontend for the persistent *Fish* backend, as could other programs. For example, agents can interact with the user by utilizing the services of *Fish* as well, whether to collect data on the activities of the user, to provide data to the user, or to provide other services.

The persistent nature of *Fish* helps it fulfill the role of a global (to one machine) repository of the knowledge shared between the user and the machine. In ongoing work we are examining how this knowledge could be effectively shared between different machines with which the user has had different interactions.

Examples

In the following examples a mock-up learning agent watches the user's command input in order to automate the creation of procedures for tasks involving repetitive command sequences. Since *Fish* allows the inter-

[4] Inspiration and guidance in building robust malleable systems can be found in recent work on meta-object protocols (KdRB91).

```
1  globalShell::2> enter latexDoc
   Now in project latexDoc::2.

4  latexDoc::2> cd LinguisticAnalysis
   /home/terry/Papers/LinguisticAnalysis

9  latexDoc::2> latex DigitalLibrary
   LaTeX Warning: There were undefined references.
   LaTeX Warning: Label(s) may have changed.
   Rerun to get cross-references right.
   Output written on DigitalLibrary.dvi (8 pages).

10 latexDoc::2> latex DigitalLibrary
   Output written on DigitalLibrary.dvi (8 pages).

13 latexDoc::2> latex DigitalLibrary
   LaTeX Warning: There were undefined references.
   Output written on DigitalLibrary.dvi (8 pages).

14 latexDoc::2> bibtex DigitalLibrary

15 latexDoc::2> latex DigitalLibrary
   LaTeX Warning: There were undefined references.
   LaTeX Warning: Label(s) may have changed.
   Rerun to get cross-references right.
   Output written on DigitalLibrary.dvi (8 pages).

16 latexDoc::2> latex DigitalLibrary
   Output written on DigitalLibrary.dvi (8 pages).
```

Figure 3: **Frontend Session 1** *Learning*

leaving of the command history of multiple frontends[5], data is available for learning that cannot be provided by current shells. The agent runs in its own thread in the backend and only makes its presence known when it produces a suggested function for the user. This function is instantiated in the *Fish* environment by the learning agent, which then tells the user the name and definition.

Once a function is suggested, the user may choose whether to keep the function. The user may also modify this automatically generated function to include domain knowledge.

Throughout these examples, only the learning agent is a mock-up. That is, we constructed a program by hand which responds to a particular (repeating) pattern of input. In practice, a learning technology such as that exhibited by the Eager system (Cyp93) (adapted for textual interaction) would be used. Also, in these examples the output of several commands was edited for presentation purposes; the user's input to *Fish* is shown as typed.

Learning Functions

[5]The common usage of *Fish* is to have multiple `xterms` running in the X Windows environment with a separate frontend running in each `xterm`. This allows the user to easily switch between frontends.

```
2  globalShell::3> enter latexDoc::2
   Now in project latexDoc::2.

5  latexDoc::2> loadFile /home/terry/Fish/Learn.pm
   File /home/terry/Fish/Learn.pm has been loaded.

6  latexDoc::2> runAsThread learn latexDoc::2
   Thread=SCALAR(0x838c518)

11 latexDoc::2> dvips DigitalLibrary

12 latexDoc::2> emacs DigitalLibrary.tex &
   success
```

Figure 4: **Frontend Session 2** *Learning*

```
3  globalShell::7> enter latexDoc::2
   Now in project latexDoc::2.

7  latexDoc::2> latex DigitalLibrary
   LaTeX Warning: There were undefined references.
   Output written on DigitalLibrary.dvi (8 pages).

8  latexDoc::2> bibtex DigitalLibrary

17 latexDoc::2> dvips DigitalLibrary

18 latexDoc::2> ls
   DigitalLibrary.aux
   DigitalLibrary.bbl
   DigitalLibrary.bib
   DigitalLibrary.blg
   DigitalLibrary.dvi
   DigitalLibrary.log
   DigitalLibrary.tex
   DigitalLibrary.tex~

   -- Message from agent 'learn' ----------------
   A repeated pattern suggests the creation
   of a new function:

   sub latexDoc::2::f1 {
      my $res = "";
      $res .= `latex DigitalLibrary`;
      $res .= `bibtex DigitalLibrary`;
      $res .= `latex DigitalLibrary`;
      $res .= `latex DigitalLibrary`;
      $res .= `dvips DigitalLibrary`;
      return $res; }

   Use the keep function to make f1 persistent.
   ------------------------------------------------

19 latexDoc::2> keep f1 texDL
   Procedure texDL now exists in latexDoc::0.

20 latexDoc::2> texDL
   Output written on DigitalLibrary.dvi (8 pages).
   Output written on DigitalLibrary.dvi (8 pages).
   Output written on DigitalLibrary.dvi (8 pages).
```

Figure 5: **Frontend Session 3** *Learning*

```
21 latexDoc::2> { sub latexDoc::0::texFile {
      my $res = "";
      $res .= 'latex $_[0]';
      $res .= 'bibtex $_[0]';
      $res .= 'latex $_[0]';
      $res .= 'latex $_[0]';
      $res .= 'dvips $_[0]';
      return $res; } }
```

Figure 6: **Frontend Session 1**
Incorporating Domain Knowledge

Figures 3, 4, and 5 are transcripts from concurrent sessions with *Fish*. The numbers at the left hand margin indicate the sequence in which commands were entered across the various sessions; they were entered by the authors for presentation.

In this example, we are processing a document with the `latex` system. In steps 1 through 3 we enter the same session environment in each frontend, thereby interleaving command and result histories. In steps 5 and 6 we load the definition of a mock learning agent and execute it in a background thread. Next, we perform the repetitive task of processing the document `DigitalLibrary` (steps 7 – 11). We then edit the file (step 12) and perform this processing again (steps 13 – 17). The mock learning agent recognizes a pattern and creates a function in the session environment to automate those commands (step 18). We then run the `keep` builtin function on it in order to place this function in *latexDoc* project environment (step 19). Lastly, we execute the function by its new name, `texDL`, and it works as expected (step 20).

There are two key points to consider. The first is that our learning agent was easily able to observe all of the user's interaction with *Fish* across several concurrent sessions. The second is that the agent interacts with the user by messaging and also through the *Fish* environment, which it shares with the user.

Incorporating Domain Knowledge

Figures 6, 7, and 8 continue the same three concurrent sessions from the previous example. We now incorporate some domain knowledge into the new function produced by the learning agent. In step 21 we edit the definition of `texDL` at the command line to create the generalized function `texFile` which now takes a filename argument. Next, we change into another directory to do more `latex` document processing (step 22). We run our newly created function on the file `security` (step 23) and discover that there are errors in the document. We decide to incorporate error checking into the function definition to avoid running unnecessary commands. In step 24 we save the definition of `texFile` to a file. Then we edit its definition in Emacs. The new function `texCheckError` is loaded in step 26. Lastly, we run `texCheckError` on `security` and the error is caught, as expected (step 27).

```
22 latexDoc::2> cd ../Networks
   /home/terry/Papers/Networks

23 latexDoc::2> texFile security
   ! LaTeX Error: Something's wrong--
                       perhaps a missing \item.
   LaTeX Warning: There were undefined references.
   Output written on security.dvi (5 pages).
   ! LaTeX Error: Something's wrong--
                       perhaps a missing \item.
   LaTeX Warning: There were undefined references.
   LaTeX Warning: Label(s) may have changed.
   Rerun to get cross-references right.
   Output written on security.dvi (6 pages).
   ! LaTeX Error: Something's wrong--
                       perhaps a missing \item.
   Output written on security.dvi (6 pages).
```

Figure 7: **Frontend Session 2**
Incorporating Domain Knowledge

```
24 latexDoc::2> saveToFile texCheckError.pm \
   $suggestedFunc
   Sucessfully wrote file texCheckError.pm.
```

User edits `texCheckError.pm`

```
25 latexDoc::2> cat texCheckError.pm
    sub texCheckError {
       my $res = "";
       $res .= 'latex $_[0]';
       if ($res =~ /LaTeX Error:/) {
          return($& . $');
       }
       else {
          $res .= 'bibtex $_[0]';
          $res .= 'latex $_[0]';
          $res .= 'latex $_[0]';
          $res .= 'dvips $_[0]';
          return $res;
       }
   }
   return 1;

26 latexDoc::2> loadFile texCheckError.pm
   File texCheckError.pm has been loaded.

27 latexDoc::2> texCheckError security
   LaTeX Error: Something's wrong--
                       perhaps a missing \item.
   Output written on security.dvi (6 pages).
```

Figure 8: **Frontend Session 3**
Incorporating Domain Knowledge

First we note that automatic generalization is a difficult problem. But it is easy for the user to generalize the `texDL` function manually. By providing access to its definition, the learning agent can encourage it. Second, we observe that the enhanced `texCheckError` function makes use of the available result history. That is, it scans the output of `latex` for errors. Ordinary shells would need the filesystem for the temporary storage of such results, or they would pipe such results through other programs for processing; neither is needed here.

Related Work

We are inspired by the design principles of Norman(Nor88) and Winograd(Win96) to improve upon current shells. Other attempts to improve the shell interaction language include *Rc*(Duf90), *Es*(HR93), and *Scsh*(Shi94). Attempts to ease the burden of complexity on the user range from Essence(HS94) which works on top of existing technology (namely Unix) to alleviate file system complexity, to Softbots(EW94) which allow the user to specify *what* information he wants rather than *how* to retrieve it, and to Plan 9(PPD+95) which is a revolutionary operating system that provides the user with a customized view of the local network.

Structured information found in everyday documents has been exploited by Apple Data Detectors(NMW98). Once a document or portion of a document is parsed with the available grammars, a list of actions relevant to the type(s) of information found is presented to the user. The user may then select an action to be performed on the data found. In our system, the user can inform *Fish* when it should parse text and for what type of data, an "on demand" approach which we believe will scale well. Of course, *Fish* users could automate these actions if desired.

Various learning agents gather knowledge by observing the user. Eager(Cyp93) and Maxims(Mae94) attempt to automate the repetitive tasks of graphical user interface manipulation and email handling, respectively. They record the user's actions in response to certain situations and when similar situations occur, the agents suggest the predicted action. When the user gains enough confidence in the agent's recommendations, the user may let the agent perform the action on his behalf. Letizia(Lie97) and COACH(Sel94) attempt to assist the user in performing their task rather than performing the task for them. Letizia attempts to suggest web pages of interest to the user based on previously requested pages, and COACH builds a model of the user's expertise by observing input keystroke by keystroke, with the goal of proactively offering the user help.

Conclusion

Our goal is to increase the level of automation in the user interface for a particular group of computer users. This group is characterized in part by its use of text-based command shells in addition to or instead of icon-based direct manipulation interfaces. The scale of typical file systems and local networks, not to mention the resources available over the Internet, makes automation challenging in this domain. Our approach has been to centralize knowledge in a persistent way while parallelizing execution. Command and result history, as well as procedures and other data, can be shared across sessions. All information is organized into a hierarchy of "projects" which act as scopes in which *Fish* resolves bindings in interactive commands and *interface programming language* programs. A concise *command language* is translated into *ipl* for execution, giving the user the power of a full programming language but also an efficient command syntax. The ability to process text resulting from command execution further empowers the user (and software programs) in automating tasks.

The learning example presented above illustrates some of the advantages of a centralized knowledge repository combined with parallel threads of execution. We also wish to point out the ease with which persistent domain knowledge may be incorporated by the user and organized as well.

We believe that many of the techniques described in the Related Work section above will prove to be fruitful in new domains. In order to apply these and other methods in the domain we have chosen, we have had to design a new interface structure between the user and the machine. Some critical aspects of this project are naturally outside the scope of this paper, including novel garbage collection methods for persistent interactive processes and automating the processing of previous command results in a principled and efficient way. The end result is a system which enables and encourages automation by the user and for the user in a complex and dynamic domain.

References

S. R. Bourne. UNIX time-sharing system: The UNIX shell. *Bell System Technical Journal*, 57(6):1971–1990, 1978.

Allen Cypher. Eager: programming repetitive tasks by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–217. MIT Press, Cambridge MA, 1993.

Tom Duff. Rc – a shell for Plan 9 and UNIX systems. In *UKUUG Conference Proceedings*, pages 21–33, 1990.

Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, July 1994.

Tim Finin and Jay Weber et al. Draft specification of the kqml agent communication language. http://www.cs/umbc.edu/kqml/kqmlspec/spec.html, June 1993.

Paul Haahr and Byron Rakitzis. *Es*: A shell with higher-order functions. In *1993 Winter USENIX Technical Conference*, pages 53–62, 1993.

Darren R. Hardy and Michael F. Schwartz. Customized information extraction as a basis for resource discovery. *Transactions on Computer Systems*, 14(2):171–199, May 1994.

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

Henry Lieberman. Autonomous interface agents. In *CHI '97. Conference Proceedings on Human Factors in Computing Systems*, pages 67–74, 1997.

Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, July 1994.

Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Communications of the ACM*, 41(3):96–104, March 1998.

Donald A. Norman. *The Design of Everyday Things.* Basic Books Publishers, Inc., New York, 1988.

Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. http://plan9.bell-labs.com/plan9/doc/9.html, 1995.

Ted Selker. Coach: a teaching agent that learns. *Communications of the ACM*, 37(7):92–99, July 1994.

Olin Shivers. A scheme shell. Technical Report TR-635, Laboratory for Computer Science, MIT, 1994.

Terry Winograd. *Bringing Design to Software.* ACM Press, New York, 1996.