RC23218 (W0405-126) May 21, 2004
Computer Science

IBM Research Report

Programming Shell Scripts by Demonstration

Tessa Lau, Lawrence Bergman, Vittorio Castelli, Daniel Oblinger
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Programming shell scripts by demonstration

Tessa Lau, Lawrence Bergman, Vittorio Castelli, and **Daniel Oblinger**
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
tessalau@us.ibm.com

Abstract

Command-line interfaces are heavily used by system administrators to manage computer systems. Tasks performed at a command line may often be repetitive, leading to a desire for automation. However, the critical nature of system administration suggests that humans also need to supervise an automated system's behavior. This paper presents a programming by demonstration approach to capturing repetitive command-line procedures, which is based on a machine learning technique called version space algebra. The interactive design of this learning system enables the user to supervise the system's training process, as well as allowing the user and system to alternate control of the learned procedure's execution.

Introduction

A recent study (Kandogan & Maglio 2003) has shown that most system administrators perform their management and troubleshooting tasks via command-line interfaces rather than using the variety of graphical user interfaces at their disposal. Command line interfaces (CLIs) have several advantages for system administration work over graphical interfaces (GUIs). First, they allow tasks to be automated, which is often necessary when performing the same task across multiple machines in a cluster, or when human error while performing a task could lead to costly downtime. Second, they preserve organizational knowledge about how to accomplish a task in a human-readable, executable form. Third, they allow administrators to easily share knowledge (in the form of copied and pasted shell commands) with their colleagues via instant messaging and email.

On the other hand, the use of shell scripts to capture procedural knowledge has its drawbacks. Administrators in the study said that they reused old scripts, handed down from previous administrators, without fully understanding what the scripts did. The cost of authoring a script may be too high, or require too much programming knowledge. The time and effort required to diagnose failures of an automated script may be prohibitive.

This paper proposes a programming by demonstration approach to the problem of capturing and automating repetitive system administration procedures. A programming by demonstration system learns how to perform a procedure by observing the user perform the procedure one or more times, directly in the user interface. Given concrete examples of the procedure's execution, the system induces variables, conditionals, and loops in the underlying procedure. It then lets the user execute the learned procedure in order to repeat the task directly in the user interface.

We have constructed a system, which we call SMARTshell, that learns Unix command-line procedures by observing the interactions between a user and a terminal. SMARTshell is an adaptive system that learns procedures from human-generated examples, and is capable of refining its behavior based on feedback from the user during the playback process. Underlying SMARTshell is a machine learning algorithm, based on version space algebra (Lau *et al.* 2003). We have chosen the machine learning algorithm carefully to enable the system to respond to user feedback, and provide a user experience in which both the user and the system collaborate to achieve the goal.

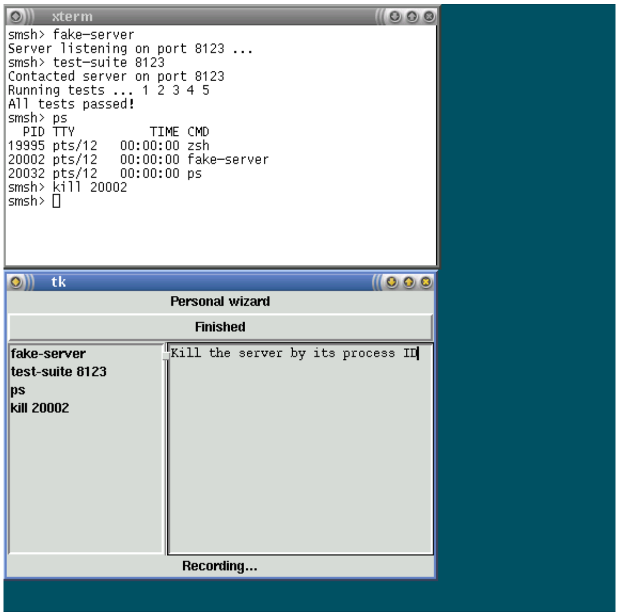
The next section presents the user interface for our system, and illustrates it with an example scenario. The section that follows describes the learning component of the system. We then discuss lessons learned, and conclude with some open questions.

SMARTshell implementation

We illustrate the SMARTshell system on a representative scenario: testing and restarting a development server. Imagine a developer who is making changes to a server, and needs to repeatedly verify that his changes have not broken any of the tests in the test suite. Each time he builds a new server, he must run the following steps in a console:

- start up the server, and make note of the port number it chose to start up on
- run the test suite, passing the server's port number as an argument
- bring up a process listing to determine the server's process id
- send a kill signal to the server using the server's process id

The developer can use SMARTshell to quickly construct a test harness to automate this repetitive task. SMARTshell

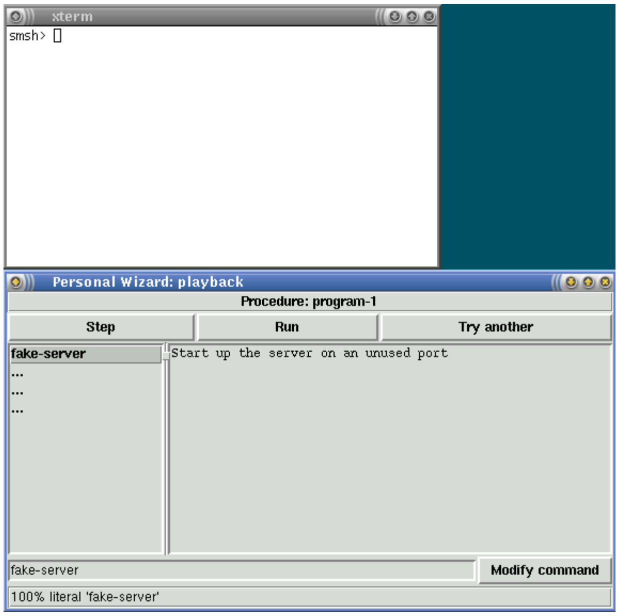


```
smsh> fake-server
Server listening on port 8123 ...
smsh> test-suite 8123
Contacted server on port 8123
Running tests ... 1 2 3 4 5
All tests passed!
smsh> ps
  PID TTY          TIME CMD
19995 pts/12   00:00:00 zsh
20002 pts/12   00:00:00 fake-server
20032 pts/12   00:00:00 ps
smsh> kill 20002
smsh> 
```

```
fake-server
test-suite 8123
ps
kill 20002
```

Kill the server by its process ID

Figure 1: SMARTshell recording interface



```
smsh> 
```

```
fake-server
...
...
...
```

Start up the server on an unused port

fake-server

100% literal 'fake-server'

Figure 2: SMARTshell playback interface: starting playback

operates like a macro recorder. Before performing the task, the developer types “smsh start” into a shell window to bring up the SMARTshell recording interface (Figure , lower window). From then on, every command he types into the shell window (top window in the figure) is recorded by SMARTshell. He has the option of annotating each step with human-readable text that explains the command, which will be displayed when the procedure is later played back.

After he has completed the task, the developer closes the recording window and SMARTshell saves the procedure for later reuse. At a future point in time, when the developer has to perform the same task again, he starts up SMARTshell in playback mode (Figure). The system indicates that the procedure is four steps long and that the first step is to start up the server. When the developer clicks on the “step” button in the playback interface, the system automatically performs the command in the console above. The user also has the option of modifying the command before it is executed; although not yet implemented, the learning algorithm could use this as feedback that the command it predicted was not the user’s intended command.

Playback continues on the next steps in the procedure, running the test suite and bringing up a process listing (Figure). The kill command in the next step requires as its first argument the process id of the currently-running server, which was printed out as a result of the `ps` command. In this case, SMARTshell guesses that the user wants to run the command “kill 20068”, using the correct process id, despite the fact that when he had recorded the procedure, the user had typed “kill 20002”.

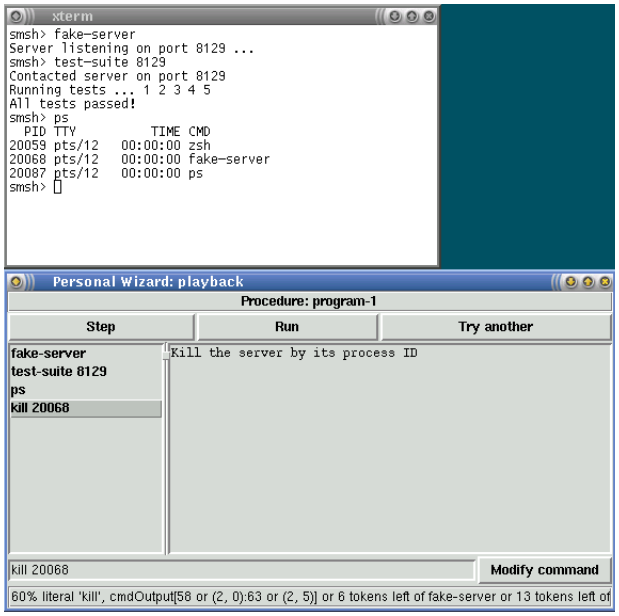
In this case, SMARTshell has correctly guessed that the user wants to extract the process id from the output of the previous command, and uses that as the argument to the current command. The bottom of the playback window shows SMARTshell’s rather cryptic explanation for selecting these particular five characters: extract the text range starting either at character 58 in the string or at row 2, column 0; and the range ending either at character 63 or row 2, column 5. All of these hypotheses extract the same text (the string `20068`). If there were a different number of lines in the process listing, the row-based and the index-based hypotheses would have produced different guesses. Given the examples seen thus far, however, the system cannot distinguish between these hypotheses.

If the user accepts this command, he can continue on to the next step in the procedure. If for any reason, this is not the right command, he may either modify it in the playback interface, or ask SMARTshell to try another guess by clicking on the “Try another” button in the interface. SMARTshell maintains a set of candidate hypotheses, and trying other guesses produces the next most likely hypotheses in that space. Whichever command is accepted, this information can be used to update the learning algorithm for future invocations of the procedure.

Once the user is fairly certain that the system has learned the correct procedure, he can automate the remainder of the procedure by clicking on the “Run” button in the playback interface.

Learning shell scripts by demonstration

We have formalized the problem of learning shell scripts by demonstration as a machine learning problem using version space algebra (Lau, Domingos, & Weld 2000; Lau *et al.* 2003). Version space algebra is a method for modelling a machine learning problem by decomposing it into smaller, independently-learnable parts, and combining the results of the subproblems into an answer for the complete



```
smsh> fake-server
Server listening on port 8129 ...
smsh> test-suite 8129
Contacted server on port 8129
Running tests ... 1 2 3 4 5
All tests passed!
smsh> ps
  PID TTY          TIME CMD
20059 pts/12   00:00:00 zsh
20068 pts/12   00:00:00 fake-server
20087 pts/12   00:00:00 ps
smsh> 
```

Personal Wizard: playback

Procedure: program-1

Step | Run | Try another

fake-server
test-suite 8129
ps
kill 20068

Kill the server by its process ID

kill 20068 | Modify command

60% literal 'kill', cmdOutput[58 or (2, 0):63 or (2, 5)] or 6 tokens left of fake-server or 13 tokens left of

Figure 3: SMARTshell playback interface: selecting one of multiple hypotheses

learning problem. It provides many of the desiderata of a PBD learning algorithm, including: incremental, online learning; learning efficiently in realtime; explanations for its inferences; and a user-understandable learning algorithm.

Version space algebra is based on the concept of version spaces (Mitchell 1982). Each version space can be thought of as a compact way of representing a set of hypotheses (drawn from a hypothesis space H) that are consistent with a sequence of observed examples. A hypothesis h is consistent with a labelled example (i, o) iff $h(i) = o$. For a given sequence of examples $D = (i_1, o_1), (i_2, o_2), ...(i_{|D|}, o_{|D|})$, a hypothesis h is consistent with D if it is consistent with each example in D. In other words, $C(h, D) \equiv \bigwedge_{(i,o)\in D} h(i) = o$. Then, a version space $VS_{H,D}$ is the set of hypotheses in H that are each consistent with all of the examples in D:

$$VS_{H,D} = \{h \in H : C(h, D)\}$$

Although version spaces were originally used for binary classification, we note that the above formulation holds for any functions that map from an input to an output. In this case, a hypothesis is a mapping from the domain to the range of the function space, and the version space consists of functions that correctly produce the output label given the input label.

Version space algebra consists of a framework for combining simple version spaces into composite ones, through the three operators *union*, *join*, and *transform*. The union allows multiple simpler version spaces to be combined into a single version space that contains the union of their hypotheses. The join enables the cross product of two spaces to form a new space; an example of a join is sequencing two commands together, where all possible combinations of a first and a second command are considered to be in the joint space. A transform converts a version space that contains functions from one domain and range to another domain and range, and enables the reuse of common components applied to a new problem. Lau (2003) has complete details on the version space algebra.

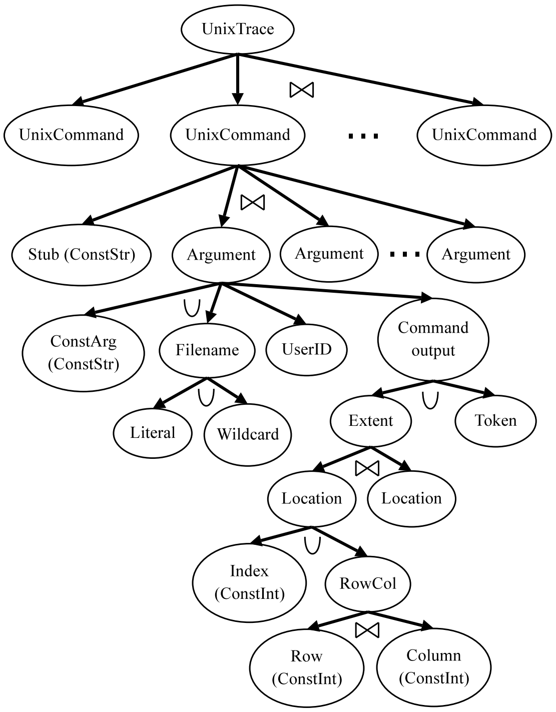


Figure 4: SMARTshell version space. The bowtie symbol indicates a version space join, while the union symbol indicates a version space union.

Figure 4 shows the SMARTshell decomposition for learning shell scripts by demonstration. A trace, consisting of a sequence of typed commands, is broken down into a sequence of commands, and each command is learned independently of the others. The top-level hypothesis space, containing all possible procedures, is the cross product of the sets of commands in the UnixCommand version spaces (e.g., one command from the first space, followed by one command from the second space, and so on). The number of commands in the procedure is lazily determined after the first example is observed, and that number is assumed to be fixed for the life of the procedure. We assume that the user performs the same steps in the same order for each demonstration of the repetitive task.

Each unix command is further broken down into the stub (the first word on the command line, typically a program like `ps` or `gcc`), and the individual arguments. As with commands, the number of arguments is lazily determined when the first example of this procedure step is observed. Each argument may either be a constant, a filename, a Unix user identifier, or some function of the previous command's output. A filename could either be a literal, constant string, or a wildcard whose value changes each iteration through a loop (for example, when touching all the files in a directory in sequence).

The CommandOutput version space contains hypotheses that describe different ways to extract information from the output of the previous command. The information to be extracted is defined as an extent, or a range of characters spanning two locations in the output. The Index version space, for example, contains hypotheses that describe locations in the character string based on offset from the beginning of the string. The RowCol version space describes locations based on row and column position in the previous command's output, and can be used (for example) to define a substring that begins at the start of the third row of output.

The version space diagram in Figure 4 describes the structure of the version space. We next describe how to update the structure in response to training examples. With each new example, the version space is pruned to only contain hypotheses that are consistent with the observed example. Examples are decomposed into sub-components that are used to train the corresponding version spaces. For example, the trace of Unix commands is separated into a number of individual Unix commands, and each command is used to update the corresponding version space. Each unix command is parsed into stub and arguments, and the values used to update the corresponding version spaces. For example, the stub `ls` must match the stub observed in previous examples of this step, otherwise the Stub version space collapses (contains no constant-string hypotheses that are consistent with all training examples). Version space collapse is one indicator that the target procedure does not lie within the bias expressed in this version space; in future work, we plan to dynamically extend the bias to consider additional hypotheses in this event.

The CommandOutput version space is updated using the current command and the output of the previous command. Each argument in the current command is matched against the character string representation of the previous command's output, and if it matches, then those hypotheses that explain how to extract that match from the previous command are retained in the version space. For example, suppose the user entered the command `kill 3777` after observing this output from a `ps` command:

```
PID  TTY          TIME CMD
3770 pts/10   00:00:00 zsh
3777 pts/10   00:00:00 ps
```

The argument 3777 could be explained by the extent starting at character 59 and ending at character 63, counting from the beginning of the string. It can also be explained as the extent starting from row 3, column 1 and ending at row 3, column 5. Another explanation is that it is the whitespace-delimited token that occurs three tokens before the token `ps`.

These hypotheses are contained within the Index, RowCol, and Token version spaces, respectively. (Technically, there is one Index/RowCol space for the starting location of the extent, and one each for the ending location of the extent.) On receipt of a new training example, these version spaces are updated to contain only those hypotheses consistent with the example. For example, if a subsequent `ps` listing had more lines of output, it is likely that the index-based hypotheses would become inconsistent, leaving only those hypotheses that explain all the observed examples.

In this way, the version space is capable of learning incrementally from multiple examples. A version space trained on one or more examples can be executed to play back the same procedure, assuming it hasn't collapsed due to insufficient bias. This is done one command at a time. For each command, the Stub version space must output a string representing the command stub, such as `vi`. Each argument version space is provided with the output of the previous command's execution, and any extraction hypotheses are applied to that string, in order to produce candidate argument values. For example, suppose an `ls` command produces the output string:

```
total 720
-rw-r--r-- tlau  19:05 aaai.sty
-rw-r--r-- tlau  10:23 discuss.tex
-rw-r--r-- tlau  10:25 introduction.tex
-rw-r--r-- tlau  13:37 learn.tex
-rw-r--r-- tlau  10:24 paper.tex
```

A hypothesis in the Extent version space might extract characters 172-182 from this string, and be joined with the command stub to produce the resulting command `vi paper.tex`. Different hypotheses could select different extents from the output string, resulting in a choice of hypotheses being produced and presented to the user for execution.

Hypotheses in the version space are given a probability in order to rank them for the user. Hypotheses in each of the leaf-node version spaces are given a probability distribution, such that the probabilities of all the hypotheses in a leaf-node version space sum to one. Each version space in a union is given a weight; when combining hypotheses from the members of the union, they are weighted by the space from which they were drawn. The probability of a hypothesis in a join is the product of the probabilities of the hypotheses contributing to the joint hypothesis. In this way, the probability of each hypothesis in the complete version space can be determined.

Discussion

Our experience with the SMARTshell system has led us to formulate several desiderata for machine learning algorithms that incorporate human input and enable the user to take control at specific points during the execution of the learned process. One of the largest barriers to the adoption of automated systems for system administrators is trust that the system will do the right thing in the right situation. Our desiderata thus reflect the need to establish trust between the human operator and the adaptive system. Specifically, we believe that:

- The system must be able to learn incrementally as users provide examples;
- Learning must happen in real-time, so that the results are immediately accessible to the user;

- The system must be able to explain its inferences, such as a user-understandable representation of the proposed hypothesis; and
- The user should be able to understand the the learning algorithm at a high level.
- The user and the system ought to be able to take turns performing steps in the procedure.

These desiderata will certainly inform our future work in the area of automating tasks through programming by demonstration, and may apply to other areas where supervisory control of machine learning systems is required.

Conclusions and open questions

In summary, we have described an approach to learning shell scripts by demonstration. We have cast the problem as a machine learning problem and formalized it using the version space algebra framework. Our SMARTshell system provides an interface for users to interact with the system to demonstrate new examples, refine the current hypothesis, and execute the procedure either automatically or under the user's control.

Many open questions remain, such as:

- **How do we evaluate the effectiveness of a supervised adaptive system?** One obvious metric for programming by demonstration systems is to evaluate how long it takes humans to complete a task both with and without help from the system. Are there any other metrics that could be used? How do we quantify the benefit of a supervised system, compared to a fully-automated system?
- **How should users interact with AI systems?** In this work we focused on example-based communication between human and system, but others have considered declarative policy-based interaction. What other possibilites exist, and what are the benefits and drawbacks of each one?
- **How do we establish trust between the human and the system?** Trust goes both ways. The user needs to trust that the automated system is controlling the process adequately; supervisory control may be a means for increasing trust by enabling the user to oversee the system's performance. However, in learning and adaptive systems, the system needs a certain amount of "trust" that the human is acting correctly and rationally. Can learning systems detect and recover from human fallibility?
- **How do we combine incremental, supervised learning with distributed, collaborative learning across many experts?** People often learn from each other. An adaptive system may be able to increase its knowledge by learning from multiple experts and combining their results together to form a single model that exploits the best knowledge from each of the experts. How do we retain the benefits of an fast, incremental approach and extend the learning algorithm to incorporate data collected asynchronously from other humans' experience?

References

Kandogan, E., and Maglio, P. P. 2003. Why don't you trust me anymore? Or the role of trust in troubleshooting activities of system administrators. In *CHI 2003 Workshop: System Administrators are Users, Too*.

Lau, T.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53(1-2):111–156.

Lau, T.; Domingos, P.; and Weld, D. S. 2000. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 527–534.

Mitchell, T. 1982. Generalization as search. *Artificial Intelligence* 18:203–226.