The Linux Plan Corpus: Documentation

Nate Blaylock
Institute for Human and Machine Cognition
Pensacola, Florida, USA

August 2010

1 Introduction

This document describes the Linux Plan Corpus, a collection of Linux sessions collected from students, faculty and staff of the Department of Computer Science at the University of Rochester in 2003 in a manner similar to that in Lesh's Unix corpus ([LE95, Les98]). Each session is labeled with a parameterized, top-level goal that is being pursued in the session. We have used this corpus in training and evaluating our goal recognition algorithms [BA05, BA04, BA03].

2 Files in the Distribution

The distribution contains three files:

- `linuxCorpus.pdf`: this document
- `raw.txt`: a file containing the unprocessed data from the corpus (the only change made to this data was the replacement of usernames with numbers to deidentify the data).
- `preprocessed.txt`: contains only the successful sesssions, converted to goal and action schema format, as outlined below.

3 Corpus Data Collection

The data collection methodology is detailed in [Bla05]. For convenience, we reprint/summarize it in this document.

The goals of the Linux corpus collection were threefold. First, we wanted to increase the size (i.e., number of plan sessions) of the Unix corpus. Second, we wanted to increase the complexity of the recognition task by adding more goal schemas. And finally, we wanted to increase the variety of goals by allowing multiple parameter values for each goal schema.

We first describe how the Linux corpus was gathered and then how it was post-processed. We then make some general observations on the resulting corpus.

3.1 Data Collection

Data for the Linux corpus was gathered from volunteer students, faculty and staff in the University of Rochester's Department of Computer Science. Volunteers were instructed to run a program installed on the local network, which led them through a collection session. This was advantageous, as it required no human supervision and could be run by users at their own convenience. Multiple concurrent users were also supported. Users were able to run the script as many times as they wished, in order to contribute more plan sessions.

3.1.1 User-Specific Data

On a first run by a user, the program gathered general data about him (as detailed below) and created an account for him. The user was then shown a set of general instructions about what the experiment was about and how the plan session should proceed. In particular, users were told that they would be given a task in Linux to perform, and that they should perform it using only the command line of the shell they were currently in. In addition, they were asked to avoid using certain constructs such as pipes or scripting languages, as we wanted to keep the mapping of one command string to one actual domain action. The actual instructions given can be found in Appendix A.

3.1.2 Goal and Start State Generation

At the start of each session, a goal and start state were stochastically created. Each goal schema in the domain was given an a priori probability, and the program used

these to stochastically choose a goal schema for the session. Each goal schema had associated with it a list of possible parameter values for each parameter position, and one of these was chosen randomly for each parameter position, giving us an instantiated goal.

Goals were similar to those used in the Unix corpus, including goals like "find a file that ends in '.txt'" and "find out how much filespace is free on filesystem /users." A list the goal schemas found in the Linux corpus can be found in Appendix B.

For each session, we generated a new start state — a new directory structure and the files within it.[1] A particular challenge was to ensure that the generated goal was achievable. Instead of trying to do this on a case-by-case basis given a generated goal, we decided to guarantee that any generated goal would be possible in any start state.

To do this, we first settled on a static set of files and directory names from which all start states were generated. The set was carefully coordinated with the set of goal schemas and possible parameters. For example, one of the possible instantiated goals was "delete all files which contain more than 40,000 bytes." To make this achievable, our static file set included several files which were larger than 40,000 bytes.

For a given session, we first created a small subset of the directory tree which ensured that all goals were possible. The remaining part of the tree was then generated randomly given the set of remaining directory names and files.

3.1.3 The Plan Session

Once the goal and start state were generated, the user was to be presented with the goal. We followed Lesh in presenting the goal as natural language text to the user. We associated a template with each goal schema which was instantiated by substituting variables with the values of the corresponding schema parameter values. Appendix B shows the Linux goal schemas and their corresponding English templates.

The goal was displayed to the user and he was given a shell-like prompt in which to input commands. The user's commands as well as their results (the

[1] It appears that the Unix corpus used a static start state for each session. We chose to generate random start states to avoid a possible learning effect in the corpus. Users who participated in multiple plan sessions may have learned the directory structure, which could have made certain tasks e.g., finding files, much easier. Eventually, one may want to model a user's knowledge of the environment, but we chose to leave this to future research.

output to `stdout` and `stderr`) were recorded and stored in the corpus. In addition, the system supported several meta-commands which were not directly recorded in the corpus:

- `success` — used to indicate that the user believes that they have successfully completed the task.
- `fail` — used to end the session without successful task completion.
- `task` — used to redisplay the session goal at any time.
- `instruct` — used to redisplay the general instructions at any time.
- `help` — used to display general help with the system.

The system continued recording commands and results until the user used the `success` or the `fail` command.

3.1.4 Data Recorded

For each plan session, the following data was recorded and is available in the raw version of the corpus:

- *Time*: date and time the session began.
- *User ID*: a unique number that identifies the user.
- *Linux level*: the user's reported proficiency in Linux between 1 (lowest) and 5 (highest).
- *User status*: whether the user was an undergraduate student, graduate student, or other.
- *Goal*: the instantiated goal schema for the session.
- *Goal text*: the actual text that was presented to the user.
- *Reported result*: whether the user reported success or failure for the session.
- *Directory structure*: the directory tree generated for the session (actual files used were static for each session and are also available).
- *Commands and results*: each issued command along with its result (from a merged `stdout` and `stderr` stream).

	Original	Post-processed
Total Sessions	547	457
Failed Sessions	86	0
Goal Schemas	19	19
Command Types	122	43
Command Instances	3530	2799
Ave Commands/Session	6.5	6.1

Table 1: Contents of the Linux Corpus

3.2 Corpus Post-processing

After the experiments, we performed various operations in order to transform the raw corpus into something we could use for training and testing our goal recognizer.

First, we excluded all sessions which were reported as failures, as well as sessions with no valid commands. Although such data could possibly be useful for training a recognizer to recognize goals which will not be accomplished by the user alone, we decided to leave such research for future work.

We also converted issued Linux commands into parameterized actions. Unlike actions in many domains used in plan recognition, Linux commands do not nicely map onto a simple set of schemas and parameters. To do the mapping, we defined action schemas for the 43 valid Linux command types appearing in the corpus. This allowed us to discard mistyped commands as well as many commands that resulted in errors. More details about this conversion process as well as the list of action schemas themselves can be found in Appendix C.

Table 1 gives a comparison of the original and post-processed versions of the corpus.

The post-processed corpus had 90 less plan sessions (86 failed and 4 where the user reported success but did not execute any successful commands!) The drastic reduction in command types (from 122 to 43) is mostly due to mistyped commands which either did not exist or which were not the intended command (and therefore not used with the right parameters).[2] The removal of unsuccessful commands was the main contributor to the drop in average commands per session.

[2] A frequent example was using the command `ld` instead of the (supposedly) intended `ls`.

3.3 General Comments

As discussed above, the Linux corpus was gathered semi-automatically from humans. As a consequence, it contains mistakes. A frequent mistake was typographical errors. The post-processing step described above helped ameliorate this somewhat — as it was able to detect incorrectly typed commands (at least in cases where the mistyped command wasn't also a successful command). However, it only checked the command itself, and not its parameters. This lead to cases of the user using unintended parameters (e.g., `ls flie` instead of `ls file`).

Another phenomenon that we were not able to automatically detect was the user's lack of knowledge about commands. For example, one user, upon getting the task of finding a file with a certain name tried several times in vain to use the command `grep` to do so, where the command he was likely looking for was `find`.[3]

Finally, another source of noise in the corpus is that the users themselves reported whether they had accomplished the task successfully. We have seen several cases in the corpus where a user apparently misunderstood the task and reported success where he had actually failed. Overall, however, this does not appear to have happened very often.

4 Conclusion

This document has described the Linux Plan Corpus.

References

[BA03] Nate Blaylock and James Allen. Corpus-based, statistical goal recognition. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1303–1308, Acapulco, Mexico, August 9–15 2003.

[BA04] Nate Blaylock and James Allen. Statistical goal parameter recognition. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning*

[3] The command `grep` is used to find text in a file or set of files, not to find a file in a directory tree.

and Scheduling (ICAPS'04), pages 297–304, Whistler, British Columbia, June 3–7 2004. AAAI Press.

[BA05] Nate Blaylock and James Allen. Recognizing instantiated goals using statistical methods. In Gal Kaminka, editor, *IJCAI Workshop on Modeling Others from Observations (MOO-2005)*, pages 79–86, Edinburgh, July 30 2005.

[Bla05] Nathan J. Blaylock. Towards tractable agent-based dialogue. Technical Report 880, University of Rochester, Department of Computer Science, August 2005. PhD thesis.

[LE95] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *IJCAI95 - Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1704–1710, Montreal, Canada, 1995.

[Les98] Neal Lesh. *Scalable and Adaptive Goal Recognition*. PhD thesis, University of Washington, 1998.

A Instructions Given to Users in the Linux Corpus Collection

We are studying how people perform tasks in Linux. We will give you a series of tasks to complete. In each case, we will record the commands you use (and their results). By continuing, you agree to let us do this recording and use it for further study and/or publications. It will in no way be used to personally identify you.

Each task should take no more than a few minutes at most. You are free to do as many tasks as you like and you may quit at any time.

INSTRUCTIONS

You will be given a task to complete in Linux. When you have successfully completed the task, use the command 'success' to indicate so. If, at any time, you wish to give up, use 'fail'. Note: the system is not actually checking to see if you accomplished the task or not. It just believes you when you say 'success' or 'fail'. Use the command 'help' if you ever need any.

You may perform the task any way you like. However, please follow the following rules:

1. Do everything in the current shell. Don't invoke new shells (tcsh, rsh, etc.) or do stuff in another program (like emacs). It prevents the program from monitoring your activity.

2. Don't use scripts (awk, perl, sh, ...)

3. Use one command per line, don't use pipes '|' or commands with other commands embedded in them, (e.g., with ';' or backticks '`'). Also, it's ok to use 'find' but not 'find -exec'

4. For each session, you will be assigned a randomly generated directory called: /u/blaylock/Experiment/Playground/username_time (where 'username_time' will be your username and the current time). Please stay within that subdirectory tree (i.e., pretend like /u/blaylock/Experiment/Playground/username_time is /)

5. Use only standard programs. Don't use any shell scripts/programs installed in personal user accounts.

6. Arrows, command completion, and command editing don't work. Sorry.

Remember, there is nowhere you need to put an 'answer' for the task. Simply type 'success' if you accomplished the task, or 'fail' if you are giving up.

The current directory is `dirname`, please treat this as your root directory.

B Goal Schemas in the Linux Corpus

There were 19 goal schemas used in the Linux corpus. Table 2 shows each schema, along with the template used to generate its English description and its assigned a priori probability.

Note that the English description includes parameters in the form $1, $2, etc. which correspond to the first, second, etc. parameter in the goal schema. In the corpus collection, these variables were instantiated with the value of the actual parameter and then the text was shown to the subject.

Goal Schema **English Description**	**Prob.**
`find-file-by-attr-name-exact(filename)` find a file named '$1'	0.091
`find-file-by-attr-name-ext(extension)` find a file that ends in '.$1'	0.055
`find-file-by-attr-name-stem(stem)` find a file that begins with '$1'	0.055
`find-file-by-attr-date-modification-exact(date)` find a file that was last modified $1	0.055
`compress-dirs-by-attr-name(dirname)` compress all directories named '$1'	0.055
`compress-dirs-by-loc-dir(dirname)` compress all subdirectories in directories named '$1'	0.055
`know-filespace-usage-file(filename)` find out how much filespace file '$1' uses	0.073
`know-filespace-usage-partition(partition-name)` find out how much filespace is used on filesystem '$1'	0.055
`know-filespace-free(partition-name)` find out how much filespace is free on filesystem '$1'	0.036
`determine-machine-connected-alive(machine-name)` find out if machine '$1' is alive on the network	0.036
`create-file(filename,dirname)` create a file named '$1' in a (preexisting) directory named '$2'	0.073
`create-dir(create-dirname,loc-dirname)` create a subdirectory named '$1' in a (preexisting) directory named '$2'	0.036
`remove-files-by-attr-name-ext(extention)` delete all files ending in '.$1'	0.036
`remove-files-by-attr-size-gt(numbytes)` delete all files which contain more than $1 bytes	0.018
`copy-files-by-attr-name-ext(extention,dirname)` copy all files ending in '.$1' to a (preexisting) directory named '$2'	0.018
`copy-files-by-attr-size-lt(numbytes,dirname)` copy all files containing less than $1 bytes to a (preexisting) directory named '$2'	0.018
`move-files-by-attr-name-ext(extention,dirname)` move all files ending in '.$1' to a (preexisting) directory named '$2'	0.091
`move-files-by-attr-name-stem(stem,dirname)` move all files beginning with '$1' to a (preexisting) directory named '$2'	0.073
`move-files-by-attr-size-lt(numbytes,dirname)` move all files containing less than $1 bytes to a (preexisting) directory named '$2'	0.073

Table 2: Goal Schemas in the Linux Corpus

C Action Schemas in the Corpus

We discuss here some of the issues in converting raw Linux command strings into parameterized actions. We first discuss some of the general issues encountered and then discuss the action schemas themselves.

C.1 General Issues for Conversion

The following describes some of the general difficulties we encountered in mapping the Linux domain onto actions. It is important to note that our goal in this project was not to write a general-purpose action-description language for Linux, rather to test a theory of goal recognition, thus some of our choices were pragmatic rather than principled.

Flags Linux uses command flags (e.g., `-l`) in two different ways: to specify unordered parameters and to change the command functionality. The former is fairly easy to handle. The latter, however, is more difficult. It would be possible to treat each command/flags combination as a separate command. However, many commands have various flags, which may be used in various combinations, which would likely lead to a data sparseness problem.

We currently just ignore all command functionality flags. The action schema name used is just the 'command name' of the Linux command (e.g., `ls` from `ls -l -a`). One option would be to form a sort of multiple-inheritance abstraction hierarchy of commands and their flags (e.g., `ls -l -a` inherits from `ls -l` and `ls -a`), although we leave this to future work.

Optional Parameters Treating various modes of commands as one command expands the number of possible parameters for each command. For example, `find` can take the `-size` parameter to search for a file of a certain size, or `-name` to search for a certain name. These parameters can be used together, separately, or not at all.

To deal with this problem, each action schema has a parameter for each *possible* parameter value, but parameter values are allowed to be blank.

Lists of Parameters Many commands actually allow for a list of parameters (usually for their last parameter). The command `ls`, for example, allows a list of filenames or directory names. Rather than handle lists in special ways, we

treat this as multiple instances of the action happening at the same timestep (one instance for each parameter in the list).

Filenames and Paths As can be seen below in the action schemas, many commands have both a `path` and a `prepath` parameter. Because our parameter recognizer uses the action parameter values to help predict goal's parameter values, it is necessary that the corresponding value be found in the action parameter where possible. Paths were especially difficult to handle in Linux because they can conceptually be thought of as a *list* of values — namely each subdirectory name in the path as well as a final subdirectory name (in the case that the path refers to a directory) or a filename (in the case it refers to a file). In a complex path, the parameter value was often the last item in the path.

As a solution, we separated each path into a `path` and a `prepath`. The `path` was the last item on the original path, whereas the `prepath` contained the string of the rest of the original path (even if it had more than one subdirectory in it).

As an example, consider the command `cd dir1/dir2/file.txt` which contains a complex path. In this case, it would translate into the following action in our corpus: `cd(dir1/dir2,file.txt)`. This way, the argument `file.txt` becomes a separate parameter value, and thus accessible to the parameter recognizer.

Wildcards How to handle wildcards (* and ?) was another issue. In Linux, filenames containing wildcards are expanded to a list of all matching file and directory names in the working directory. However, that list was not readily available from the corpus. Furthermore, even if there is not match to expand to, we would like to be able to tell that a command `ls *.ps` is looking for an extension `ps`. Our solution was to simply delete all wildcards from filenames.

Current and Parent Directories The special filenames `.` and `..` refer to the current working directory and its parent, respectively. The referents of these were not readily available from the corpus, and leaving them as `.` and `..` made them look like the same parameter value, even though the actual referent was changing (for example when a `cd` was executed).

For each goal session, we rename `.` and `..` to `*dot[num]*` and `*dotdot[num]*`, where `[num]` is the number of `cd` commands which have been executed thus far in the current plan session. This separates these values into equivalence classes

where their real-life referent is the same. Of course this doesn't handle cases where a later `cd` comes back to a previous directory.

C.2 The Action Schemas

There were 43 valid command types used in the Linux corpus which we converted into the action schemas listed below. Each action schema lists the name of the command as well as its named parameters.

- `cal()`
- `cat(prepath,path)`
- `cd(prepath,path)`
- `clear()`
- `compress(prepath,path)`
- `cp(dest-prepath,dest-path,source-prepath,source-path)`
- `date()`
- `df(prepath,path)`
- `dir(prepath,path)`
- `du(prepath,path)`
- `echo(string)`
- `egrep(pattern,prepath,path)`
- `fgrep(pattern,prepath,path)`
- `file(prepath,path)`
- `find(prename,name,size,prepath,path)`
- `grep(pattern,prepath,path)`
- `gtar(dest-prepath,dest-path,source-prepath,source-path)`

- gzip(prepath,path)
- info(command)
- jobs()
- less(prepath,path)
- ln(dest-prepath,dest-path,source-prepath,source-path)
- ls(prepath,path)
- man(command)
- mkdir(prepath,path)
- more(prepath,path)
- mount()
- mv(dest-prepath,dest-path,source-prepath,source-path)
- pico(prepath,path)
- ping(machine-name,machine-path)
- pwd()
- rlogin(machine)
- rm(prepath,path)
- rsh(machine,command)
- ruptime()
- sort(prepath,path)
- tar(dest-prepath,dest-path,source-prepath,source-path)
- touch(prepath,path)
- tree(prepath,path)
- uncompress(prepath,path)

- `vi(prepath,path)`
- `which(command)`
- `zip(dest-prepath,dest-path,source-prepath,source-path)`