A UNIFORM ONTOLOGY FOR SOFTWARE INTERFACES

Final Report

Period Covered: 2/28/2001- 2/28/2002
NASA Grant NAG 1-2271
W&M Accounting No. 301511

Principal Investigator: Stefan Feyock
Computer Science Department
College of William & Mary
Williamsburg, VA 23187

1 RESEARCH GOALS

It is universally the case that computer users who are not also computer specialists prefer to deal with computers[1] in terms of a familiar ontology, namely that of their application domains. For example, the well-known Windows ontology assumes that the user is an office worker, and therefore should be presented with a "desktop environment" featuring entities such as (virtual) file folders, documents, appointment calendars, and the like, rather than a world of machine registers and machine language instructions, or even the DOS command level.

The central theme of this research has been the proposition that the user interacting with a software system should have at his disposal both the ontology underlying the system, as well as a model of the system. This information is necessary for the understanding of the system in use, as well as for the automatic generation of assistance for the user, both in solving the problem for which the application is designed, and for providing guidance in the capabilities and use of the system.

Having the ontology and model available makes it possible to provide a number of facilities, including the following:

> Automatic generation of application- and system-level help with a uniform look and feel
>
> Assistance in searching for solutions of the application problem, including automatic recognition of tool applicability

[1] We will use the term "computers" to refer to the whole of the computer system as perceived by the user: hardware, entailing processor, monitor, and peripherals, and software, from the operating system to programming tools to applications.

Provision of guidance on how to apply these facilities, given the present state of the computation and the interactive session.

Recognition and retrieval of known precedents whose relevance is determined on the basis of similarity at the ontological level

2 RESEARCH APPROACH

2.1 Ontology Specification

For any complex system, in particular computer/software systems, the view and the underlying implementation will in most cases have radically different ontologies. Given that the intent is to shield the user from the underlying complexity of widely diverse systems, it is necessary to provide a top-level interface that is based on an ontology that is powerful, conceptually elegant, and capable of expressing a wide variety of systems.

Formal mathematical is an obvious candidate that fulfills all of the above requirements for specifying ontologies. Unfortunately general mathematical notation can prove daunting for non-mathematician users, even those trained in another technical discipline. We have therefore decided to use the mathematical modeling and specification language Z [3]. The Z notation is, in terms of comprehensibility, quite accessible technically trained users. It is related to relational database notation, and its data representation system is easily translated to simpler notations such as the entity-relationship model.

2.2 The Specification Language Z

Since the intent of this project was to model a file system, the file system was first modeled in the mathematical modeling and specification language Z [3]. The Z notation serves both as implementation guide for the system implementor, and as view of the system for user while the user is interacting with the system.

3 TEST BED DOMAIN: THE UNIX FILE SYSTEM

For our research the test bed of choice was the Unix file system, in particular the creation of a responsive, user-friendly interface to this system capable of providing the user with extensive goal-oriented assistance. This system well illustrates the relationship of applications, models, and ontologies, and has a level of complexity comparable to the ones that are of interest to the aviation community (as well as being more readily available than actual aircraft).

To summarize, the application, or rather the user's view of his application, is modeled in

Z, which provides an ontology consisting of abstract sets and function on these sets. This view level is in turn implemented in terms of an underlying system based on an unfamiliar ontology from which the user is to be shielded. For our research this underlying system is based on MOPs [1], which feature a frame-based universal ontology whose details must be hidden from the user.

3.1 The Underlying Implementation

The implementation of the Z-specified user interface described above was in terms of Schank's MOP[2]-based programming system [1] This system provides a powerful AI-based programming capability, which is indispensable for determining the intent of the user, inferring from this the user's needs, and then providing the required services, including

natural language understanding capabilities

case-based reasoning (so that the system can learn and improve its capabilities on the basis of experience.

a powerful knowledge representation system

Unfortunately Schank's MOP-based reasoning system, as well as the ontology on which is tis based, is comprehensible only to the specialist who has devoted extensive study to this programming system and methodology. Furthermore, the MOP-based system's ontology is quite general, and therefore its ontology is not specific to any particular application area, in particular the application area whose ontology is familiar to the user.

It is therefore necessary to present a *view of the application ontology* that hides the system's implementation and presents the user with only the familiar application ontology.

* The user-oriented ontology of our test bed is an abstraction (aka simplification) of the Unix file system.

* How the view works with/translates to implementation (DMAP as natural language understanding system)

[2] Memory Organization Packet

4 FUNDAMENTAL DEFINITIONS

In the discussion below we will treat as isomorphic the concepts of *sequence* (mapping from *1..n* into some set *S*) and lists with elements from S. In our context such lists will also be referred to as *paths*.

Given this convention, we will model the file hierarchy as a *set of lists* (paths). This amounts to using *lists* (rather than just atoms) as the names of files or directories. Atomic names are then converted into list names by concatenating them onto the content of the environment variable "." (synonym: *CURRENT_DIRECTORY*). We will accordingly treat as equivalent the string notation $\backslash n_1 \backslash n_2 \backslash \ldots \backslash n_k \backslash$ and the list notation $(\backslash\ n_1\ n_2 \ldots\ n_k)$.

Definition: If *s1* and *s2* are lists (in general: sequences) over a set *A* of atoms, then *s1* is said to be a *prefix* of *s2* iff *s2* == *s1* || *s* for some sequence *s* over *A*. If *s* != *( )*, then *s1* is said to be a *proper prefix* of *s2*.

4.1 Representation Of Trees As Sets Of Full Paths:

Let T_A be a tree with nodes labeled by members of some set A. We can represent any path in tree T_A as a list {n1,...,nk} of the node labels on that path, beginning with the node closest to the root[3].

Special case: ( ) represents the empty path.

We will say that a path p is a *full path* if p is a path in T_A from the root to a leaf of T_A.

Let *S* be a set of lists (in general: sequences) over a set *A* of atoms, having the property that if $s \in S$, then no proper prefix of $s \in S$. Equivalently, if $s \in S$, then *s* is not a proper prefix of any other *s'* in *S*.

We will call sets of lists *S* having this property *prefix-free*.

It is evident that if *S* is a set of lists representing the set of all *full* (root-to-leaf) *paths* of some tree *T*, then, since no proper prefix of a full path of a tree *T* can itself be a full path of *T*, *S* must be prefix-free.

Conversely, as can easily be shown by an inductive proof, if *S* is a prefix-free set of lists, then *S* is the set of all full paths of some tree *T*.

[3] We will use "\" to denote "root." In list notation this becomes (\ ).

Thus any tree T over a set A can be represented as a set of lists over A, namely the set of full paths of T.

Let R be some binary relation. Then:

$domain(R) = \{x \mid x\ R\ \_\}$

$range(R) = \{x \mid \_\ R\ x\}$

xR *denotes* $\{y \mid xRy\}$

Ry *denotes* $\{x \mid xRy\}$

$\oplus$ denotes the *override* operator, defined as follows:

Let f and g be functions, i.e., (in Z's notation) sets of pairs of the form { x |-> y }.
Then $f \oplus g =def= (f \setminus \{x \mapsto y : (\exists z)[x \mapsto z \in g]\}) \cup g$.

It is evident that, in database terms, $\oplus$ is an *update* operation on relation f:
if $x \mapsto z$ occurs in g, then update any tuples of the form $x \mapsto y$ occurring in f to $x \mapsto z$.

4.2 Notational Conventions and Auxiliary Definitions

We will use the following notations and definitions in the discussion below:

Let *ipath* (implied path) denote the full expanded path name,
to wit "." || <file spec>, of the file.

Let DIRECTORY denote the path set representing the file hierarchy.

Let "~" (synonym: HOME) denote the list name of the user's home directory,
and "." (synonym: CURRENT_DIRECTORY) denote the current directory.

5 Z DEFINITION OF THE UNIX FILE SYSTEM

5.1 Data Structure Definitions In Z

```
[CHAR]                          // The set of characters
SPACE  == {' '}                 // SPACE is the space character
EOF == {eof}                    // EOF is the end-of-file character
COLON == {:}
SLASH == {/}
```

```
PERIOD== {.}
WORD == fseq1(CHAR \ {BLANK, EOF,  SLASH,  COLON})

[PERSON]
[FILE]       // set of abstract entities representing files

FILE_TOKEN subset FILE;  // FILE_TOKEN is the set of existing files
(i.e., in-use file tokens)
TEXT == fseq1(CHAR \ {EOF})    // TEXT is a sequence of any CHAR but EOF
FILE_TEXT ==TEXT || <EOF>      // FILE_TEXT is the text in text files

USER subset PERSON;
GROUP subset POWERSET(USER);  // a group is a set of users
FILE_NAME subset WORD;  // file names are finite non-empty seqs of CHAR
(except EOF)
PATH_NAME subset fseq(FILE_NAME);  // pathnames are lists (possibly
empty) of filenames

 | DIRECTORY subset PATH_NAME;
+-----------------------------------------------
 | Constraint: DIRECTORY is a tree over FILE_NAME


PERMISSIONS subset {1 .. 9} -> {0,1}  // permissions are bit strips of
length  9
FILE_TYPE == {d, -}   // is this file a directory or not?
BYTES == N

user_id: USER      //  the current user
Groups: GROUP x OWNER  // Groups is an m:n relation

path2token_map: PATH_NAME -> FILE _TOKEN  //  could be m:1 (remember
links?)
token2permission_map: FILE _TOKEN -> PERMISSIONS   //  1:1 into; each
file has a permission
token2type_map: FILE _TOKEN -> FILE_TYPE  //  n:1; every file is either
a directory or regular file
token2owner_map: FILE _TOKEN -> USER  // n:1; every file has an owner
token2content_map: FILE_TOKEN -> FILE_TEXT;
name2path_map: FILE_NAME -> PATH_NAME
mask: USER -> PERMISSION      // default permissions associated with
user

name2path_map(file_name) == CURRENT_DIRECTORY || file_name
name2token_map(file_name) == path2token_map(name2path_map(file_name))

 | file2group_map: FILE_TOKEN -> GROUP
+-------------------------------------------------
 | Constraints: file2group_map(f) in  Groups (token2owner_map(f))
 | // the file's group must be a group the owner belongs to.

// some useful macro functions:
owner(f: FILE_NAME) ==
token2owner_map(path2token_map(name2path_map(f)))
permissions(f: FILE_NAME) ==
token2permissions_map(path2token_map(name2path_map(f)))
type(f: FILE_NAME) == token2type_map(path2token_map(name2path_map(f)))
```

```
new_FILE_TOKEN( ) == /*return*/ f, where f is_in FILE - FILE_TOKEN
new_file(fname?: FILE_NAME, type?: FILE_TYPE, permissions?:
PERMISSIONS} ==
{       let new_path = CURRENT_DIRECTORY || fname?;
        let new_token = new_FILE_TOKEN( );
        DIRECTORY' = DIRECTORY + {new_path!};
        path2token_map' = path2token_map ⊕ {new_path -> new_token!}
        token2owner_map' = token2owner_map ⊕ {new_token -> user_id}
        token2permission_map' = token2 permission _map ⊕ {new_token -
>permissions?}
        token2type_map' = token2owner_map ⊕ {new_token -> type?}
}
```

5.1.1 The Size Attribute For Files

```
BYTES : N       // N is the set of natural numbers
file_size_map: FILE_TOKEN -> BYTES   // file_size_map returns the size
of a file
```

5.1.2 Date Information For Files

```
DAY == 1..31
MONTH == {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}
YEAR: N             // YEAR is a natural number
HOUR == {00, 01, 02, 03, ... , 22, 23}
// HOUR is a number from 00 to 23 with leading zero pattern
MINUTE == {00, 01, 02, ... , 58, 59}
// MINUTE is a number from 00 to 59 with leading zero pattern
TIME == HOUR || COLON || MINUTE>

OLD_DATE: MONTH || DAY || YEAR>
// DATE is a MONTH concatenated with a DAY concatenated with a YEAR.

NEW_DATE:  MONTH || DAY || TIME

DATE: NEW_DATE ∪ OLD_DATE      // DATE is a NEW_DATE or an OLD_DATE.

CURRENT_DATE: NEW_DATE         // CURRENT_DATE is a NEW_DATE

last_date_map : FILE_TOKEN +> DATE  // date of last access
```

5.2 Operator Definitions In Z

Actually, the Z specification system does not define operators per se; instead, it allows the definition of predicates that give the relationship between two states of the system being specified. This allows operators to be defined as functions that map a "before" state, specified by a set of preconditions, into an "after" state specified by a set of postconditions. In the following definitions *Effect* describes this state-to-state mapping.

```
| chmod (f?: FILE_NAME, new_permissions?: PERMISSIONS)
+-----------------------------------------------
| Preconditions:
| owner(f?) = USER_ID   // must own the file to change permissions
| Effect:
| let ftoken == name2token_map(f?);
| token2permission_map' =
|   token2permission_map ⊕
|   {(ftoken -> token2permission_map(ftoken)) V new_permissions?}
| // do a logical OR of f?'s permissions and new_permissions?


| compress(f?: FILE_NAME)
+-----------------------------------------------
| Effect:
| let f = f? || ".z"
| new_file(f,  // name of new (compressed) file
|          -,  // type of new  (compressed) file
|          permissions(f)
|             // permissions of  compressed file: same as of old file
|          )
| file_size_map(ftoken) < file_size_map( (FILE_TOKEN) f? )
|        //  cast f? to TOKEN_TYPE


| cp(f?, fcopy?: FILE_NAME)
+-----------------------------------------------
| Effect:
| new_file(fcopy?,  // name of new file
|          f,       // type
|          mask(user_id),  // default permissions associated with user
|          )
| token2content_map((FILE_TOKEN) fcopy?) :=
|                      token2content_map((FILE_TOKEN) f?)


| ls(name?: FILE_NAME, name_list!: fseq[FILE_NAME])
|  // name_list! is a list of file names
+-----------------------------------------------
| Effect:
| name_list! = {n : CURRENT_DIRECTORY || name? || n is_in DIRECTORY}
|  // Note: name? Could be the empty sequence,
|  //in which case the current directory is returned


| mkdir(fname?: FILE_NAME)
+-----------------------------------------------
| new_token!: FILE_TOKEN;
| new_path!: PATH_NAME;  // these two are local variables
```

```
|  // Create a new empty directory named fname
|  //under the current directory
| new_file(fname?,
|          d,  // type
|          mask(user_id),  // default permissions associated with user
|         )


| mv(old_fname?, new_fname?: FILE_NAME)
+-------------------------------------------------------
| let old_path_name = name2path_map(old_fname?);
|     // get the full pathname of the old file
| let new_path_name = name2path_map(new_fname?);
|     // construct the full pathname of the new file
| let token = path2token_map (old_path_name);
|     // get the file token itself
| DIRECTORY' = DIRECTORY \ {old_path_name} + { new_path_name };
| path2token_map' =
|      path2token_map \ {old_path_name |-> token} U
|      { new_path_name |-> token};


| pwd(current_directory!)
+-------------------------------------------------------
| current_directory! = CURRENT_DIRECTORY


| rm(fname?: FILE_NAME)
+---------------------------------
| Precondition: file2type_map(fname?) == -
|
| let path_name = name2path_map(fname?);
|   // get the full pathname of the file
| let token = path2token_map (path_name);
|   // get the file token itself
| DIRECTORY' = DIRECTORY \ {path_name};
| path2token_map' = path2token_map \ {path_name |-> token};
| if {p : path2token_map (p) == token}== {}
| then FILE_TOKEN' = FILE_TOKEN \ {token}
|    // if no more links to this file, then remove it.


| rmdir(dname?: FILE_NAME)
+---------------------------------
| Precondition: file2type_map(dname?) == d
|    // dname? must be a directory
|
| let path = name2path_map(dname?);
|    // get the full pathname of the directory
| let token = path2token_map (path);   // get the file token itself
| DIRECTORY' = DIRECTORY \ {p : path is a prefix of p};
|    // remove path & all its children
| path2token_map' = path2token_map \ {path |-> token};
```

```
| if {p : path2token_map (p) == token}== {}
| then FILE_TOKEN' = FILE_TOKEN \ {token}
|     // if no more links to this directory, then remove it.
```

6 THE MOP IMPLEMENTATION LEVEL

6.1 Designing An Interface For A File System

The most basic and important part of a computer's task is editing, modifying, and executing files. Thus, the first part of creating a natural language interface is to create a natural language interface with the file system.

6.2 Organizing A File System

The following section describes the logic used to organize the file system interface using Case-Based Reasoning architecture. The First task is to organize the main MOPs.

6.2.1 Determining Mops

The first step in organizing the actual MOPs is to determine the answer to these questions:
- What needs to be recognized?
- How will it be recognized?
- What will be done once it is recognized?

This particular interface needs to recognize file system commands and questions about files. This involves commands, file names, and file attributes, as well as queries about each of these entities. A further important type of question concerns system actions and reactions (and lack of these), such as *why did this command not log me out?* Each item to be recognized or concept was designed to have its own branch (i.e., MOP) within the hierarchy. This enables checking abstractions of the MOPs recognized when determining the task of the interface.
For example:
If the abstraction of the concept entered is a command, then a command is executed.

If the abstraction of the concept entered is a question, then the answer is determined and printed.

6.2.1.1 Organizing Files

An abstract MOP *m-file* is defined for files and their attributes using the defmop macro. It executes before interaction begins, and creates the hierarchy of MOPs. While files are described first within the paper, the attributes must be described first within the code. The definition of a file, as well as a directory

(which formally is a specialization of a file), is shown below.

```
(defmop m-file (m-root)
(userid m-userid)
(group m-group)
(size m-size)
(userread m-access)
(userwrite m-access)
(userexecute m-access)
(groupread m-access)
(groupwrite m-access)
(groupexecute m-access)
(worldread m-access)
(worldwrite m-access)
(worldexecute m-access) )

(defmop m-directory (m-file)
      (userid m-userid)
      (group m-group)
      (size m-size)
      (userread m-access)
      (userwrite m-access)
      (userexecute m-access)
      (groupread m-access)
      (groupwrite m-access)
      (groupexecute: m-access)
      {worldread: m-access}
      (worldwrite: m-access)
      (worldexecute: m-access))
```

Only the abstraction MOP *m-file* of a file is defined before the code executes. The MOPs describing individual files are created upon entering the interface. The files are read by means of an *ls -l* command piped to a file and read back. Each occurrence of a file specification causes the creation of an instance MOP, as well as the slots for the attributes added to the file instance using *ADD-ROLE-FILLER*, defined in [1]. After each file is read in and added, a new defphrase function is executed to define a trigger for the file in case it is referenced during interaction with the user. The example below illustrates adding a new file MOP, adding three of the many slots, and, finally, the definition of the trigger of the MOP. The trigger is set to be the name of the file.

```
(defun new_file  (pathname
                  filename
                  useraccess
                  groupaccess
                  worldaccess
                  userid
                  group
                  size
                  month...)

   (setq filemop (convertstr_symbol filename))
     (cond ((not (mopp filemop))
            (new-mop filemop '(m-file) 'instance nil))
           (t (delete-key (mop-table 'slots)
```

```
 filemop)

; add userread

(let ((tuserread (convertstr_symbol (concatenate 'string "i-m-" (subseq
useraccess 0 1))))) (cond ((not (mopp tuserread))
(new-mop tuserread '(m-access) 'instance nil)))
(add-role filler 'userread filemop tuserread))

                                                        ; add userid
slot

(let ((tuserid (convertstr_symbol (concatenate 'string "i-m-"
userid))))
  (cond ((not (mopp tuserid))
         (new-mop tuserid '(m-userid) 'instance nil)))
  (add-role-filler 'userid filemop tuserid))

; add size slot

(let
    ((tsize (convertstr_symbol
      (concatenate 'string "i-m-" (write-to-string size)))))
    (cond ((not (mopp tsize))
           (new-mop tsize '(m-size) 'instance nil)))
(add-role-filler 'size filemop tsize))

; define the phrase that will activate the mop

(let ((symfile (convertstr_symbol filename)))
  (fdefphrase filemop (list symfile)))
filemop)
```

Note that if the instance of a file MOP is already defined, all slots for that file are deleted. This enables updating of the slots with new sizes, new access, etc.

Note also that file MOP is not set to "*i-m*" concatenated with the filename, while all other instances do begin with "*i-m*". During many sections of the code it is necessary to use the filename as an argument. Not using the "*i-m*" relieves the code of finding the substring of the filename MOP.

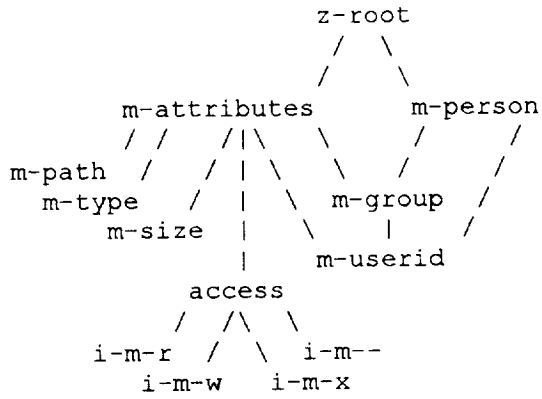
1.3.1.2 Organizing File Attributes

The file attributes are themselves MOPs, and fillers within the slots describing the files point to the file's attributes. The *defmops* below show how the attributes are defined before the interface begins to execute.

```
(defmop m-attributes (m-root))
(defmop m-person (m-root))
(defmop m-group (m-attributes m-person))
(defmop m-userid (m-attributes m-person m-group))
(defmop m-size (m-attributes))
(defmop m-type (m-attributes))
(defmop m-access (m-attributes))
```

```
(defmop i-m-r (m-access))
(defmop i-m-w (m-access))
(defmop i-m-x (m-access))
(defmop i-m- (m-access))
(defmop m-path (m-attributes))
```

Chapter 11 of Inside Case-Based Reasoning uses a standard prefix "m-" to define abstract concepts and a prefix "i-m-" to show an instance. This notation is carried into the project, with the exception of the filenames as mentioned in the note in 5.3.1.1. Below shows how we can view the MOP attributes as a hierarchy.



Note that the MOPs for group and userid have two parents. MOPs may have many different abstractions. While this feature is not used in this particular interface, questions could relate to the users as people as well as to an attribute of a file.

The *defphrase* macro is executed to create triggers for attributes:

```
(defphrase m-size size)
(defphrase m-size big)
(defphrase m-size little)
(defphrase m-size kb)
(defphrase m-size kilobytes)
```

When any of the above words - size, big, etc. - is mentioned, the *m-size* concept is referenced. When any of the above words is referenced in relation to a specific file, the role *size* in the mop representing that file can be accessed. The role names should describe the abstract attribute referenced to enable efficient searching for a file attribute value. The slots can be looped through until a role matches the attribute in question.

6.2.1.2 Organizing Command MOPs

The commands are defined with DEFMOPs before execution of the interface.

```
; commands with no arguments
```

```
(defmop ls (m-noarg-command)
(defmop pwd (m-noarg-command))
(defmop lpq (m-noarg-command))

; commands with arguments

(defmop lpr (m-one-command) (file m-file))
(defmop cd (m-one-command) (path mpath))
(defmop mkdir (m-one-comnmnd) (directory m-directory))
(defmop emacs (m-one-command) (file m-file))
(defmop compress (m-one-command) (file m-file))
(defmop uncompress (m-one-command) (file m-file))
(defmop rmdir (m-one-command) (directory m-directory))
(defmop rm (m-one-command) file mfile))
(defmop mv (m-two-command) (file m-file))
(defmop cp (m-two-comman4) (file m-file))
(defmop chmod  (m-four-command)
(file m-file) (group m-group)
(operator) (access))
```

Again the organization comes down to recognition. After a MOP is recognized, it is determined to be a command by checking its parent (abstraction) and then executed with the arguments also recognized within the command. Below are some of the defphrases used to trigger the command MOPs.

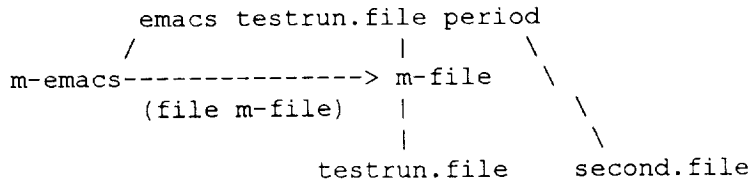
```
; defphrases

(defphrase ls  ls period)
(defphrase ls  dir period)
(defphrase ls  list period)
(defphrase ls  folder period)

(defphrase pwd pwd period)
(defphrase pwd current directory period)
(defphrase pwd which directory period)
(defphrase pwd where am I period)

(defphrase lpq  lpq period)
(defphrase lpq  printer queue period)
(defphrase lpq  queue)

(defphrase lpr   lpr (file) period)
(defphrase lpr  print (file) period)
(defphrase lpr  hard copy (file) period)
(defphrase lpr  print file (file) period)

(defphrase cp  cp (file) period)
(defphrase cp  cpy (file) period)
(defphrase cp  copy (file) period)

(defphrase mkdir  md period)
(defphrase mkdir  make directory period)
(defphrase mkdir  make folder period)
(defphrose mkdir  new directory period)
(defphrase emacs  emacs (file) period)
(defphrase emacs  emacs (file) period)
```

(defphrase emacs edit (file) period)

Commands are broken into three categories: commands with no arguments, commands with all defined/recognizable arguments (one-arg) , and those with some unrecognizable arguments (two-arg) Commands without arguments such as ls or lpq are simply recognized and executed.

Commands with arguments such as *emacs* or *lpr* must have the file/argument entered. The argument is defined with the syntax (file): the role file within a slot modifying the command, instead of an exact word. The abstract version of the particular command has a role file and a filler pointing to mfile. Any MOP triggered under m-file will satisfy the trigger for the role (file).

(defmop emacs (m-one-command) (file m-file))
(defphrase emacs emacs (file) period)



The triggers *emacs, test.file,* and *period* will be recognized, *emacs* will be executed, and the argument for file *(test.file)* used. *test.file* is recognized only because it was defined as described in the FILE section above.

Commands such as *mv* and *cp* require the user to enter a word or file destination that cannot be recognized by the parser. Concepts are recognized only if they have a trigger that has previously been defined by *defphrase* or *fdefphrase*. When executing *MV* and *CP* the destination file is new and unrecognizable by the system. When a word does not trigger any MOPs and a command requiring a destination is a possible target, the unrecognized word is stored as a destination for subsequent use.

6.2.1.3 Execution of the Command

There are five types of interactions that the parser expects to recognize: commands with no arguments, commands with one argument, two arguments, four arguments, and questions regarding attributes. Each of these interactions has code defined to perform the command and answer the user. First the code for these interactions is written as a LISP function, then the name of the function is defined as a MOP. Each type of interaction has a slot with the name of the function to be called. *get-filler* is used to call the function. The one-argument interaction is defined as follows:

(defun onearg-function (mop pred)
(setq command (pred-base pred))
(funcall command
(nstring-downcase (subseq (symbol-name (slot-filler (car (mop-slots mop)))) 4))))

(defmop onearg-function (m-function))

```
(defmop m-one-command (m-command) (getcode onearg-function))

(defun pred->mop (pred)
(let ((mop (slots->mop (pred-slots pred)
                       (list (pred-base pred))
t))
    (start (pred-start pred)))
(format t "~&activating ~s" mop)
(cond ((get-filler 'getcode (pred-base pred))
(funcall (get-filler 'getcode pred-base pred)) mop pred)))

((for (next-pred :in (get-triggered-preds mop start))
     :do (advance-pred next-pred mop))))
```

Each of the functions is passed the prediction and MOP in order to keep everything uniform. The prediction contains the (target phrase base start) where target is the MOP predicted, phrase is the part of phrase defined to trigger the MOP and waiting to be recognized, base is the MOP to be activated, and start is the position in the sentence where the index pattern first started being recognized. Prediction is a list defined in Inside Case-Based Reasoning (p358). The MOP is the specific interaction recognized. It is important to note that inheritance causes the filler for getcode to be retrieved from an ancestor of the MOP pased to it if the MOP does not have the code defined Thus, if the prediction is for emacs (test.file), a new MOP is created for this interaction, emacs (mfile) is the parent, m-one-command is the grandparent and role getcode is found with filler onearg-function.

6.2.1.4 Command History

The system understands "new concepts." Every time a file is deleted, copied, etc, the information about the command is stored. When the MOPs are activated for a command, a specific instance is instantiated into the command and is stored as a new item of information. If *rm testrun.lsp* is entered, then rm (file) becomes *rm m-file* which will translate to *i-m-rm testrun.lsp* which is a new specific instantiation now stored. When the user cannot find a file, there is a record of the deletion. This can be used to help solve user problems such as not finding a file. If a command does not have arguments, nothing new is learned and a new instance is not stored. Only a new instantiation of a command with a different file is stored.

It is desired that this system answer questions pertaining to files. The most obvious question is about a file's attributes. Thus there is a branch from z-root for the concept of attribute questions and defphrases for recognizing questions.

```
; mops that relate to questions about attributes of files

(defmop attribute-questions (m-root) (getcode attribute-question-
function))
(defmop question-size (m-attribute-questions)
        (file m-file)
(verb is)
```

```
(size m-size)
        (describe kilobytes)
)
```

The above MOP shows how questions about the size of a file are defined under m-attribute-questions. Note that when defining names of MOPs, any important words that need to be recognized should be last for getting substrings of known length (e.g., m-question-size). The prefix can be a set length, then the important words found by using the substring function. This feature is used in many situations including attribute questions. Below are defphrases for recognizing a questions pertaining to the size of a file.

```
; defphrases relating to direct attribute questions

(defphrase m-size size)
(defphrase m-size big)
(defphrase m-size little)
(defphrase m-size kb)
(defphrase m-size kilobytes)

(defphrase m-question-size how (size) (file) period)
(defphrase m-question-size what (size) file) period)
(defphrase m-question-size (size) file) period)
```

Given the question *(c '(how big is testrun.file))*, the interface should fetch the size of *testrun.file* and tell the user the size of *testrun.file*.

The system will recognize "how" and activate any MOPs starting with "how". "big" will activate the concept m-size as well as continue the activation of m-question-size. "is" will be ignored, but will not affect the activation of m-question-size. "*Testrun.file*" will activate *m-file* and continue *m-question-size*. The period is automatically put at the end of all interactions to signal the end. We have the following MOP:

```
(m-question-size (file i-m-testrun.file) (size m-size))
```

When the question is recognized, the slots are looped through to answer the question and the file attribute matching the target of the question is found using routines provided by *Inside Case-Based Reasoning*. The following code performs this task:

```
(defun attribute-question-function (mop pred)
(let ((answer (convertstr_symbol (subseq (symbol-name (pred-base pred))
1 1)))
       (file (role-filler file mop)))
for (slot :in (mop-slots mop))
:do (cond ((equal (slot-role slot) answer)
(print (subseq (svmbol-name (role-filler answer file)) 4)))
(t (print (subseq (symbol-name
(slot-filler slot)) 4)))))))
```

First, the answer is set to the last part of the question MOP's title : *m-question-size*; thus the target is size. File is set to the MOP for

the filename entered: *testrun.file*. The slots for the question are looped through. If the slot does not have role the same as answer (target of the question), then the filler (file-name) is printed. Slots are stored in the desired order for printing. If the slot's role is the same as the target of the question, then the slot-filler for that role within the file is found to print the answer for the file. Currently the question is saved for debugging purposes. Because the answer can be re-derived at any time, it is desirable that the new MOP recognized be deleted right after the question is answered.

6.2.1.5 Problem Questions

This is the most difficult part of the system. The interface should be able to determine why certain problems occur, such as a file not being found. The user types in a question about a current problem relating to files and receives answers or help to determine the source of the problem. The problem questions are again entered as MOPs, and corresponding trigger questions for problems are entered using *defphrase*. Below is a common type of problem query, triggered by queries such as "where is file *testrun.lsp*?".

```
; functions for finding a file problem

(defun find-file-problem-function (mop pred)
for (slot :in (mop-slots (pred-base pred)))
:do (cond ((or (equal (slot-role slot) 'M-FILE)
(equal 'slot-role slot) 'GETCODE))
((and (not (equal (Slot-role slot) M-FILE))
(not (equal (slot-role slot) 'GETCODE)))
(print funcall (slot-filler slot)
(slot-role slot)
(slot-filler (mop-slots mop))))

;  example: funcall calls the code for find-file-past-function for mv
or rm


(defun find-file-past-function (mopname file)
(setq found nil)
(for (mop :in (mop-specs mopname))
:do (cond ((equal file (get-filler 'FILE mop))
                  (setq found mop))
)) found)

; found is set to the mop containing the filename
; example: I-M-RM.125 for a rm of file TESTRUN. LSP

;  defmop relating to access of file mop problem

(defmop m-mistakes (m-root))
(defmop m-error-typing (m-mistakes) (file m-file))
(defmop m-error-rm (m-mistakes) (file m-file))
(defmop m-error-mv (m-mistakes) (file m-file)
(defmop m-error-cd (m-mistakes) (file m-file)
(defmop m-error-access (m-mistakes) (file m-file)
```

```
(defmop m-problem-answers (m-root))
(defmop m-find-file-problem (m-problem-answers)
      (getcode find-file-problem-function)
      (m-file m-file)
      (rm find-file-past-function)
      (mv find-file-past-function)
      (file find-file-past-function)
)

; defphrases relating to problem - access of file
; defphrase m-find-file-problem where is file))
```

After the system determines the MOP is a find-file problem, it would loop through the slots for m-find-file-problem, executing the code listed as the filler. The filename and the MOPs that will have their descendants searched are passed to the code. The code is stored in the filler of the slots. The code to search for MOPs that will solve the problem is executed and found is set to the specific MOP solving the problem.

7 BIBLIOGRAPHY

[1] *Inside Case-Based Reasoning*, Riesbeck, Christopher, and R. Schank, Lawrence Erlbaum Associates, 1989

[2] *Common LISPcraft*, Wilensky, Robert, Norton Books, 1986

[3] *The Way of Z*, Jacky, Jonathan, Cambridge University Press,1996

[41 *Understanding Z*, J.M. Spivey, Cambridge University Press, 1988

[5] *PROLOG Programming for Artificial Intelligence, 3rd ed.*, Bratko, Ivan, Addison-Wesley, 2001

[6] *Artificial Intelligence*, 3rd ed., Winston, Patrick H., Addison-Wesley, 1992