A Step Toward an Intelligent UNIX Help System: Knowledge Representation of UNIX Utilities

Bryan So
Larry Travis

Technical Report #1230

April 1994

A Step Toward an Intelligent UNIX Help System: Knowledge Representation of UNIX Utilities

Bryan So (so@cs.wisc.edu)

Larry Travis (travis@cs.wisc.edu)

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, WI 53706
USA

April 19, 1994

ABSTRACT

Because of its wide availability and open architecture, UNIX™ provides a suitable and convenient environment for command user interface research. We believe a formal, yet practical, knowledge representation scheme for UNIX utilities is a necessary tool to build higher-level online help and active help systems. We propose a hierarchy of knowledge levels to represent knowledge of UNIX: static objects, command syntax, command synopsis, command semantics and UNIX semantics. We describe how to implement the first three levels using taxonomic classification technology. We illustrate their usefulness by showing how they can be used to correct some non-trivial mistakes made by novice users. The ultimate goal of this research is to realize a UNIX command user interface (UNIX shell) that is knowledgeable enough to give intelligent advice to the user. By intelligent advice, we mean advice of a kind previously available only from human experts.

1. The UNIX Interface

Because of its wide availability and open architecture, UNIX provides a suitable and convenient environment within which the human-computer interaction researchers can work on improving command user interfaces. A user interacts with UNIX by typing commands through a program known as the *shell.* The chief function of the shell is to dispatch, or *run*, any of the hundreds of utility programs available in UNIX [Kernighan81]. The designers of UNIX have emphasized the principles of simplicity and elegance. Therefore, the traditional shell is small and lacks features. For example, while the user interfaces of most operating systems have a command to list the contents of a directory, the UNIX shell does not. In order to list the contents, one must run a separate program (also called *command* or *utility*) dedicated to this purpose. In line with the same principle, most UNIX commands are small in size, terse in input and output format, and algorithmically straightforward. Some benefits of this design are that it is elegant, concise, and even easy to learn and use — for mathematically-oriented programmers. The drawback, however, is that it is dreadfully difficult for the novice to learn. The burden is made heavier by cryptic command names and other accidental undesirable features. Many design deficiencies pointed out years ago have never been repaired [Norman81, Norman85].

Understandably, there are some attempts to tame the UNIX system for beginners and even some more ambitious attempts to provide online assistance for experienced users. These attempts take the following approaches:

- Shell enhancements. This approach aids the user by augmenting the traditional shell with features. Filename completion, history, and alias were added in the C Shell to reduce cognitive load for general users [csh89]. Familiar programming notations have been added to help programmers. Visual line editing and automatic spelling correction have been added in a new version of the C shell [tcsh91]. More recent attempts correct syntactic and pragmatic mistakes by using contextual evidence [Eide91]. This approach is limited by its syntactic nature. Most often, the user's input is matched with templates and the advice given by the system is canned. There is no help for problems of a semantic nature.
- Graphical user interfaces (GUIs). Research and development in graphical interfaces for UNIX has skyrocketed as the cost for powerful graphics hardware plummets. GUIs, like the X

Window System, have become part of many UNIX distributions. All of the commercial UNIX GUIs (such as Tab Window Manager and Motif) and many research prototypes (such as Xerox's Rooms [Henderson86] and Silicon Graphics' 3D File System Navigator[1] [Tesler92]) are visualization tools for the UNIX file system. We would like to see more direct manipulation capabilities[2] in these systems. In particular, we want to see more graphical user interfaces like that of [Borg90] that intuitively capture the UNIX pipe and I/O redirection mechanisms.

- Online help systems. These systems assist users through a question and answer format. The simplest of all is "man", a standard UNIX command that displays the usage of any UNIX command. In this case, the question is always "how is command *x* used?". The answer printed to the question "man *x*" is the manual pages for the command *x*. It should be noted that the manual pages are intended to be read by experienced users and are not easily comprehensible by novices. Research in online help systems takes on several forms, including guidelines for online documentation authoring [Horton90], information storage and retrieval techniques [McCune85, Gordon88], information organization techniques (such as hypertext and multimedia systems) [Conklin87, Barrett88, Barrett89, Nielsen90], natural language understanding and generation [Wilensky84], and modeling users to create an adaptive help system [Jones88, Kass88, Nessen89]. The major weakness of this approach is the question answering format. Often, a novice user knows so little that he/she does not know enough to ask the right question in the right terminology. Also, unless the online help system volunteers to offer help, the novice user may be working with some ineffective procedures for a long time while remaining completely unaware that a more effective procedure is possible and could be asked for.

- Active help systems. The kind of help provided by the above-mentioned systems is called *passive help* because the user has to take initiative and ask for advice [Fischer85]. An *active help*

[1] As seen by millions in the movie, Jurassic Park.

[2] "visibility of the objects and actions of interest; rapid, reversible, incremental actions; and replacement of complex command-language syntax by direct manipulation of the object of interest" [Shneiderman92, p.183]

system, in contrast, runs in parallel with the user, keeping track of his/her actions and taking initiative to give advice.

As an illustration, let's pretend a novice user types the following command line in UNIX:

```
compress big-file | mail -s A Compressed File smith@cs
```

The intention of the user is to send the user smith@cs a file named "big-file". He/She has learned to use the -s option of the mail command to specify a subject line that reads "A Compressed File". He/She has also learned to use "compress" to keep the message short. Furthermore, he/she knows the pipe (|) will join these two separate commands as one. However, there are several mistakes made by the user. A human expert might rewrite the above command line as:

```
compress -c big-file | uuencode big-file | mail -s "A Compressed File" smith@cs
```

More importantly, the expert would explain to the novice the reasons behind the modification. The spirit of an active help system is to simulate such a human expert, to give online advice as previously only a human expert would.

There is little research on active help systems in general, and active UNIX help systems in particular. Most of the current research emphasizes high-level, cognitive problems such as user modeling and plan recognition [Quilici88, Shrager82, Jerrams-Smith89, Cesta91, Woodroffe88, Jones88a]. We believe, however, those researchers will appreciate a set of lower-level knowledge representation tools upon which these high-level solutions can be built. Concerning his Yucca-II system (a passive UNIX help system), Hegner writes,

> We have found many times that had the command semantics of UNIX been formally specified in the first place, before any implementation, not only would the design of the consultant [have] been much simpler, but much anomalous behavior could have been avoided, and so many of the commands themselves would have been much more understandable, and the process of documentation would have been much more systematic. [Hegner87, p.46]

This paper investigates the plausibility of a consistent knowledge representation formalism for UNIX utilities. Such a formalism must satisfy the following goals:

1. it is realizable in the current UNIX environment;
2. it does not limit itself to passive help systems;
3. whenever possible, there is an easy conversion process from existing documentation to the representation formalism.

Although we are not defining the semantics of UNIX as rigorously as [McDonald90], we do provide a less ad hoc knowledge specification scheme before the implementation stage. We also illustrate how we could make use of this knowledge representation formalism to create an active help system and how would it correct erroneous command lines (such as the one displayed above). Section 2 describes the nature of knowledge in UNIX and in our representation scheme. Section 3 introduces a practical representation system that is the basis of our formalism. In Sections 4 to 6, we describe the knowledge representation scheme in three levels of detail. Section 7 exemplifies use of the knowledge. Section 8 is a summary.

The ultimate goal of this research is to realize a UNIX shell that is knowledgeable enough to give intelligent advice to the user. The completed shell will be able to induce the intention of the user and supply meaningful advice rather than canned responses. For example, the user enters the following sequence of commands:

```
crypt <letter >letter.crypt
mail another_user <letter.crypt
```

The system will reason that the user intended to send a letter to another user secretly. However, the user made a mistake because it is unreliable to send binary data using "mail". So the following advice is given:

> It is unreliable to send binary data using mail. You may use the xsend command. If your system does not support xsend, you may use uuencode to encode the encrypted letter first. The encoded letter can be later decoded by using uudecode.

2. Knowledge Levels

Knowledge in our scheme is represented at different levels of abstraction, where concepts at higher levels make use of concepts defined at lower levels. The numerous benefits in this approach include clarity, conciseness, incrementality and maintainability. Because UNIX is written by programmers for programmers, the interface is so well structured that it is natural to describe it in terms of such hierarchical knowledge levels. For example, all input/output devices are treated as files (i.e. streams of characters) regardless of their actual instantiation which may be, for example, a disk drive, a modem, a printer or a network port. The same does not necessarily apply to knowledge representation of the less structured medical diagnosis domain, for instance.

We have identified five levels of abstractions for UNIX. Respectively, they are the static objects level, the command syntax level, the command synopsis level, the command semantics level and the UNIX semantics level.

- Level 1: Static Objects

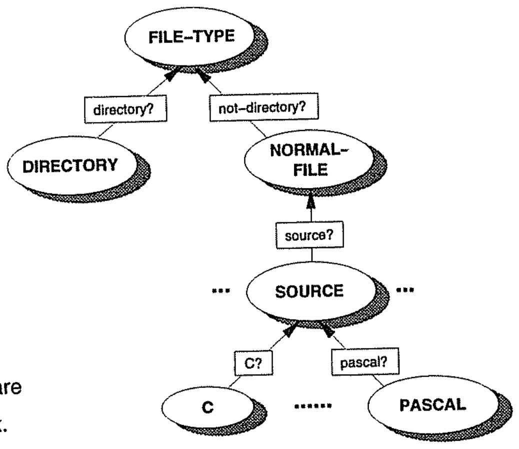
  At this level, operating system objects are declared and organized in a subsumption hierarchy [Woods91]. Some operating system objects include identifiers, files, filenames, user names and commands[3]. They are arranged in a hierarchy according to their relationships in UNIX. Figure 1 shows a small part of the FILE-TYPE hierarchy. It shows that C and PASCAL are two kinds of SOURCE, which is a kind of TEXT, which is a kind of NORMAL-FILE. Concepts under the same parent are mutually exclusive. Each concept has an associated membership test that decides whether an object is an instance of that concept.

- Level 2: Command Syntax

  At this level, each UNIX command is briefly described. This description includes the command name, valid options, input/output file types, and certain intrinsic properties, such as whether the command is destructive or not.

[3] We do not represent the individual commands yet, only that there is a generic concept called COMMAND.

Figure 1: A part of the FILE–TYPE hierarchy



Ovals represent concepts. Arrows indicate subclass relationships. Membership tests are in boxes associated with each subclass link.

- Level 3: Command Synopsis

  At this level, each UNIX command is defined in greater detail than for Level 2. Specifically, this level simulates the "Synopsis" description in the UNIX User's Reference Manual. These details include the numerous usages for each command, the types of arguments of each usage, the meaning of each option ... etc. Recall that one of our goals is to be able to map existing documents to our formalism. This synopsis knowledge level contributes directly toward this goal.

- Level 4: Command Semantics

  At this level, the semantics of each UNIX command is defined. The command is described in terms of simple programming constructs, such as loops and conditions, and operation primitives, such as file operations (*read*, *write*, *remove*...). We intend that the knowledge engineer will be able to map existing source codes into this formalism.

- Level 5: UNIX Semantics

  At this level, UNIX semantics such as I/O redirection and the pipe mechanism are introduced. Until now, UNIX commands have been treated as stand alone objects only. Within Level 5, one can model complex tasks by means of combinations of commands.

This separation of knowledge levels is similar to Doane's UNICOM cognitive theory of UNIX expertise. A knowledge base so constructed displays an incremental nature of expertise which is defined as: first, a thorough awareness of static objects; second, a syntactic understanding of the utilities; third, a semantic understanding of those utilities; and finally, the knowledge to put them together [Doane92]. As a beneficial side effect, a shell that utilizes this scheme can be constructed incrementally, with progressively more knowledge at each stage.

In many ways, Levels 1, 2 and 3 are logically separated from Levels 4 and 5. First, they are the syntactic parts of the representation scheme. Second, they are analogous to one major part of the shell: parsing a single command. Third, they can all be represented in a single taxonomic classification module. This paper emphasizes the representational issues of these three levels and demonstrates the usefulness of the scheme before introducing any semantic level. Levels 4 and 5 will be treated in a separate paper.

3. Taxonomic Classification

Since our knowledge representation scheme is based on classification-based technology, it is useful to give a brief introduction to the latter's history and development. Further, it is necessary to understand the features and terminology of a particular classification system, C-CLASSIC, in order to understand our formalism.

3.1. Terminology

Major expert systems constructed in the 80's were mostly object-based. Object-based representations are more structured and easier to maintain than rules. A number of variations of object-based technologies have been used in different systems, such as frame-based [Minsky85], access-oriented [Stefik89] and object-oriented [Stefik89a]. More recently, another object-based technology has evolved which combines the strength of objects, rules and logic. It is called classification-based technology [Mac Gregor91, Brachman83].

Knowledge in a classification-based system is encoded in two kinds of languages, namely a *terminological language* and an *assertional language*. The terminological language is used to define structural components of abstract concepts while the assertional language is used to assert the existence of instantiations of these abstract concepts (concepts and instantiations are analogous to definitions of records and variable values of those records in a programming language). There are three types of objects manipulated by the languages: concepts, roles and individuals.

- Concepts

  A *concept* represents a class of objects in the real world. In a classification system, a concept is defined by a set of necessary and sufficient conditions expressed in well-defined logical operators. A new concept is automatically inserted in a multiple inheritance taxonomy of concepts such that the most specific concepts that subsume it become its immediate parents, and the most general concepts that it subsumes become its immediate descendants. A concept C_1 *subsumes* a concept C_2 if and only if all instances described by C_2 are also described by C_1. This automated process is called *classification*.

  Most systems also allow one to define *primitive concepts*. Primitive concepts have only necessary conditions and represent objects that are either too complex or too vague to define precisely. They are usually at the upper levels of the taxonomy from which more specific concepts are derived. At the root of the taxonomy is a primitive concept CLASSIC-THING, where all other concepts descend.

  In this document, concept names are typeset in capital letters. As an example, let us define two similar concepts: IDENTIFIER and COMMAND.

IDENTIFIER	**isa**	CLASSIC-THING		
	hasa	*name*	**isa**	STRING
			exactly	1
COMMAND	**isa primitive**	CLASSIC-THING		
	hasa	*name*	**isa**	STRING
			exactly	1

The two definitions are identical except IDENTIFIER is a defined concept while COMMAND is a primitive concept. The first definition says an identifier has a name which must be a string and the converse is true. That is, anything that has a name (which must be a string) is classified as an identifier. The second definition says a command has a name which must be a string, but (being a primitive concept) the converse is not necessarily true. As it turns out, IDENTIFIER subsumes COMMAND because any instance that is a COMMAND must also satisfy the condition for being an IDENTIFIER.

- Roles

  One concept is related to another by means of *roles*. A role in a classification system is analogous to an attribute in a frame-based system or a "has-a link" in a semantic network, except that a role may have more than one value filled in. These values are called *fillers*. One may use the terminological language to define the properties of a role. For example, a role may be defined to have fillers that are instances of STRING only (*value restriction*), or it may be restricted to have a certain number of fillers (*number restriction*). Sometimes a role can be defined as a specialization of another role. In the example above, *name* is a role relating IDENTIFIER and COMMAND to STRING (i.e., a filler of *name* must be a string). The example also states that there must be one and only one filler for *name* in each instance of IDENTIFIER and COMMAND. Roles are typeset in italics in this document.

- Individuals

  An *individual* is an instantiation of a concept. It usually represents an extensional object in the physical world. In older semantic network systems, there was no assertional language. As a result, the user had to explicitly specify the concept to which an individual belonged. For instance, in the assertion "Peter isa BOY", Peter is an individual and BOY is a concept. With an assertional language, a system can do various kinds of automated reasoning, depending on the expressive power of the language. A user can now assert "Peter isa PERSON", together with other role information (such as *age* and *gender*), and let the system classify the individual, Peter, under the most specific concept, BOY.

As a more relevant example, the following may be two individuals in a UNIX shell:

Command001	**hasa**	*name*	**fills**	"mail"
Command002	**hasa**	*name*	**fills**	"mail"
	isa	COMMAND		

Both individuals are classified under IDENTIFIER. Command002 is further classified under COMMAND according to told information. Notice there is not enough information to classify Command001 under COMMAND because having a *name* is only a necessary condition of a COMMAND individual, not a sufficient one. Individuals in this document are typeset with capitalized first letter and often with an integer suffix.

3.2. Classification-based Systems

Ronald Brachman formulated a classification-based language, KL-ONE, in his Ph.D. dissertation [Brachman77, Brachman85]. KL-ONE sired a large family of classification-based systems, commonly known as the KL-ONE Family [Woods90]. Some well-known descendants of KL-ONE are KL-TWO, NIKL, KRYPTON, LOOM and CLASSIC. The chief driving force behind this research current is the inherent difficulty when balancing the expressive power of the language against its efficiency of inference.

Besides efficiency, there is another interesting problem. Nature precludes an inference procedure that is both expressive and complete (i.e., all logical consequences of a knowledge base can be proven). There is no common consensus that favors a limited but complete language or an expressive language which may fail to find a valid solution at some critical moment. Consequently, one branch of the KL-ONE family consists of systems with powerful, expressive but incomplete languages, while a second branch consists of small, elegant but trusted systems. CLASSIC belongs to the second branch. According to [Mac Gregor91, p.390],

> The term language chosen for the CLASSIC system [Borgida89, p.390] represents the first system that is, roughly speaking, "as expressive as possible" while preserving computational tractability and completeness of inference.

3.3. C-CLASSIC

The original CLASSIC is written and embedded in Common Lisp [Resnick91]. The system we have employed to represent UNIX utilities, C-CLASSIC[4], is a version of CLASSIC that is written and embedded in the C programming language [Weixelbaum93].

Figure 2 (adapted from [Brachman91]) shows a representative part of C-CLASSIC's language grammar. We shall explain the specifics of the language as they come up in the following sections. It is here helpful, however, to remark on the following features:

- The terminological language and assertional language are identical since <individual-expr> has the same definition as <concept-expr>.
- Role expression in C-CLASSIC is limited. In other languages, it is possible to derive specialized roles from a parent role. In C-CLASSIC, a <role-expr> is simply a <symbol>.
- The operator **test-c** is used to invoke procedural membership tests coded in the C language (CommonLisp for CLASSIC). This hybrid strategy (mixing procedural and declarative representation) is usable in small language like C-CLASSIC to accomplish complex tasks like

[4] Much of the following description of C-CLASSIC also applies to CLASSIC, so we'll simply use the name C-CLASSIC instead of the clumsy phrase "CLASSIC and C-CLASSIC".

Figure 2: The C-CLASSIC grammar

<concept-expr>	::=	THING | CLASSIC-THING | HOST-THING |
		<concept-name> |
		(**and** <concept-expr>$^{+}$) |
		(**all** <role-expr><concept-expr>) |
		(**at-least** <positive-integer><role-expr>) |
		(**at-most** <non-negative-integer><role-expr>) |
		(**exactly** <non-negative-integer><role-expr>) |
		(**fills** <role-expr><individual-name>$^{+}$) |
		(**test-c** <fn><arg>*) |
		(**one-of** <individual-name>$^{+}$) |
		(**primitive** <concept-name><concept-expr>) |
		(**disjoint-primitive** <concept-name><concept-expr><index>)
<individual-expr>	::=	<concept-expr>
<concept-name>	::=	<symbol>
<individual-name>	::=	<symbol> | <cl-host-expr>
<role-expr>	::=	<symbol>
<cl-host-expr>	::=	<string> | <number>
<fn>	::=	a function in the host language (C language) with three-valued logical return type
<arg>	::=	an expression passed to a test function
<index>	::=	<symbol>

determining whether a PATHNAME "/usr/smith/mbox" is the name of an EXISTING-FILE. The disadvantage is that automatic classification is limited because there is no way for the system to tell whether one procedural test subsumes another.

- A procedural test returns a three-valued logical constant: TRUE (when the tested instance proves to be a member), FALSE (when it is proves to be a non-member) or MAYBE (if there is not enough evidence to prove either). C-CLASSIC is a monotonic reasoning system and does not work under a closed-world assumption.

There are three types of rules in C-CLASSIC. Figure 3 shows the rule syntax. All three kinds are forward-chaining. A *simple rule* has an antecedent concept and a consequent concept. Whenever an individual is classified under the antecedent concept, it is also classified under the consequent concept. The truth maintenance component will check for inconsistencies. Should the individual

Figure 3: C-CLASSIC rules

<antecedent>	::=	<concept-name>
<consequence>	::=	<concept-name>
<rule>	::=	<simple-rule> | <computed-future>
<simple-rule>	::=	(<antecedent> <consequence>)
<computed-future>	::=	(<antecedent> <computed-concept>) |
		(<antecedent> <computed-filler>)
<computed-concept>	::=	(**computed-concept** <fn> <arg>*)
<computed-filler>	::=	(**computed-filler** <fn> <role> <arg>*)

contradict the definition of the consequence, an error will occur.

A *computed-concept* is like a simple rule, except a procedural function is called which must return a concept as the rule consequence. This is useful if the consequence cannot be fixed in advance, but has to be computed based on the values of some role fillers.

A *computed-filler* is used to compute a filler for a role if the filler cannot be fixed in advance. Whenever an individual is classified under an antecedent concept of a computed-filler, the associated procedural function is called to provide an individual or a vector of individuals to be filled into the indicated role. This allows the system to derive new information based on given data. For example, the *suffix* role of a FILENAME may be computed to be ".c" given a FILENAME "classic.c".

Collectively, computed-concept and computed-filler are called *computed-futures*.

4. Level 1: Static Objects

The process of knowledge base construction involves the major tasks of of knowledge acquisition and knowledge formalization. There is a large literature devoted to these subjects [Gonzalez93]. This paper does not emphasize techniques of knowledge base construction for UNIX (i.e., how to do it in an elegant and cost-effective way); instead it explores what to represent and what the final constructed knowledge base should look like. It also highlights some obstacles and some difficult decisions that may stand in the way, so they may be avoided or remedied. This section discusses issues in representing level-one, static objects. The next two sections discuss level-two and level-three representations.

4.1. What Are They?

Static objects are essentially primitive objects being manipulated by users. They are, in other words, operands of some basic user operations. Therefore, the variety of static objects to represent also depends on the operations encoded in levels 2 and 3. Static objects are usually nicely separated into groups with respect to the subsystems of an operating system. For example, the file system has different kinds of files as static objects; the process manager has different kinds of processes; an editor manipulates objects like chapters, paragraphs and words. But there may be miscellaneous objects, such as identifiers, etc., that do not belong to any specific groups. In Level 1, concepts are created that represent classes of these static objects. Table 1 lists the static objects involved in the file system component of a knowledge-based UNIX shell.

Table 1: File system static objects in a knowledge-based shell

System	Category	Concept	Example Individual
file system	name objects	FILENAME	"classic.c"
		RELNAME	"bak/classic.c"
		PATHNAME	"/usr/smith/classic/classic.c"
	file objects	FILE-TYPE	a file type "C"
		FILE-MODE	readable, writable
		EXISTENCE	the concept of an existing file
		FILE	a file with inode no. 20507

Static objects are organized semi-automatically by C-CLASSIC in a taxonomic hierarchy, as mentioned in Section 2. Many of the higher-level objects will be primitive concepts (i.e., defined with necessary, but not sufficient, conditions). Under a primitive concept, FILE-TYPE, for example, one could find a large hierarchy of more specific file types, of which Figure 1 shows a part. Details on organizing such a hierarchy and the definitions of the concepts will follow shortly.

Notice that some of the concepts (called abstract concepts) are only semi-realistic or not realistic at all. For example, FILE-TYPE is an abstract concept. There is no real, material tag stamped on a C source file that says “C”, although one may guess it from the filename suffix “.c”. (But this relationship between filename suffix and file type is not guaranteed.) The concept EXIST (and its mirror image, NON-EXIST) is also an abstract concept. It is also used as a tag to denote that a file exists (or does not exist).

The number of static objects to encode depends entirely on the intended application. The same scheme may be applied to a knowledgeable UNIX shell or a knowledgeable word processor, though in each case, the knowledge encoded will greatly differ.

4.2. Level 1 Experience

This section describes our effort in constructing a simple level-1 knowledge base for the UNIX file system component. It describes some methodology we found useful, some difficulties encountered and the solutions.

FILE

FILE is the most important concept in UNIX. Each individual of FILE is either an existing file in the file system or an instantiation of a device treated as a file, such as the keyboard or the terminal.

In order to represent files, we need to find out what attributes a file has. This is not difficult. By consulting the UNIX man-page for the entry “stat”, we found the following description:

stat() obtains information about the file named by path.

...

A stat structure includes the following members:

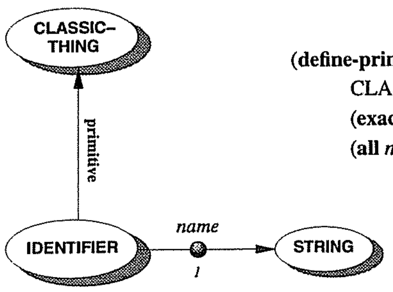
```
dev_t       st_dev;       /* device file resides on */
ino_t       st_ino;       /* the file serial number */
mode_t      st_mode;      /* file mode */
nlink_t     st_nlink;     /* number of hard links to the file */
uid_t       st_uid;       /* user ID of owner */
gid_t       st_gid;       /* group ID of owner */
dev_t       st_rdev;      /* the device identifier (special files only)*/
off_t       st_size;      /* total size of file, in bytes */
time_t      st_atime;     /* file last access time */
time_t      st_mtime;     /* file last modify time */
time_t      st_ctime;     /* file last status change time */
long        st_blksize;   /* preferred blocksize for file system I/O */
long        st_blocks;    /* actual number of blocks allocated */
```

Each member of this structure becomes a role of the FILE concept. For instance, *st_size* becomes a role of FILE that has a value restriction of INTEGER (which is what off_t is), and will be filled in by the system with the size of the file. The exact meaning of each role need not concern the reader. For each FILE individual, a forward-chaining rule is set up to invoke a procedural method that fills each of the above roles with the appropriate value. The values can be conveniently extracted by using the "stat" system call. This all illustrates how we apply existing documents and system facilities to aid knowledge engineering.

IDENTIFIER

In a command user interface, a user communicates to the computer by typing in keywords. All the computer sees at the top-most level are lexical identifiers. These are later classified into more specific concepts. The following is the definition of the class IDENTIFIER:

Figure 4: Definition of IDENTIFIER



```
(define-primitive IDENTIFIER (and
        CLASSIC-THING
        (exactly 1 name)
        (all name STRING)))
```

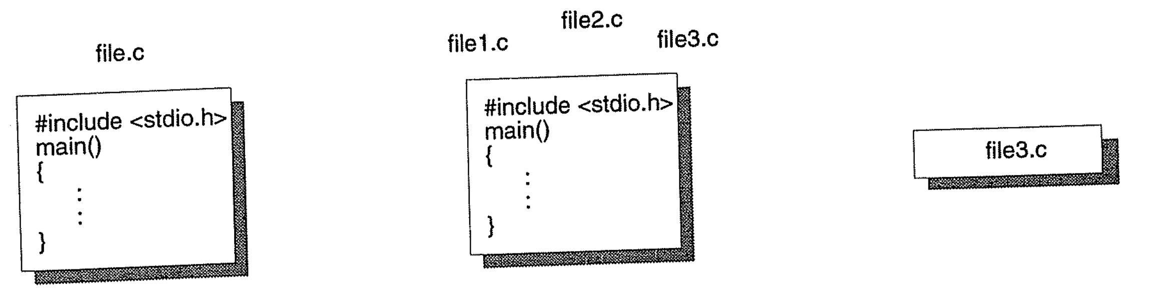
Translation: IDENTIFIER is a primitive concept under the most general concept, CLASSIC-THING. The small circle indicates that IDENTIFIER has a role *name* that has exactly 1 filler of type STRING, a predefined concept.

FILENAME and PATHNAME[5]

Representations of some static objects may be deceptively obvious. Novice UNIX users may believe a filename is a role of a file because it is so obvious that files and filenames go hand in hand. This is, however, a clumsy representation for three reasons. First, it would be difficult to represent a filename of a non-existing file. Second, it would be difficult to find the corresponding FILE individual given a filename. Third, the scheme that we use is cleaner if we allow a FILE individual to have more than one filename, a situation found in UNIX but not in some operating systems (e.g., MSDOS). See Figure 5.

[5] To simplify matters, we will use filename and pathname interchangeably despite their subtle differences. See FNAME below for a thorough discussion.

Figure 5: Relationship between FILE and FILENAME



Novice user's model: one file, one filename

Correct UNIX model: one file, many filenames

And... an independent filename

Because filenames and pathnames can have independent existence, a better scheme is to represent them as concepts rather than roles:

(**define-concept** FILENAME (**and** IDENTIFIER (**test-c** filename? *name*)))

(**define-concept** PATHNAME (**and** IDENTIFIER (**test-c** pathname? *name*)))

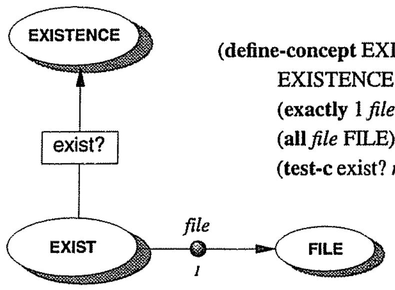
where filename? and pathname? are procedural predicates (written separately in C) which perform tests to verify the legality of the *name* filler. For example, in UNIX a legal pathname begins with a "/", has a certain length limit and must be composed of a certain set of characters. Given an identifier, C-CLASSIC is able to classify whether it appears as a filename, a pathname or simply an identifier. The intention of the user does not always concur with C-CLASSIC's classification, though. For the command line "mail userA", the identifier "userA" will be classified under FILENAME, because it is a legal filename judging solely from its appearance, while the user intents it to be a USERNAME. Fortunately, there is no conflict. Given a correct definition for the concept USERNAME, the identifier "userA" will be classified under both FILENAME and USERNAME and will be disambiguated when the right context arises.

Other than *name*, the role inherited from IDENTIFIER, there are some other roles related to a filename. The most important one may be *file*. This role is consulted when the system wishes to find out the corresponding file object for a given filename. However, *file* is not present in the above definitions for FILENAME and PATHNAME. This is because it is not an essential property. Rather it is an *incidental* property that exists only for filenames of existing files. The following section on EXISTENCE elaborates on this.

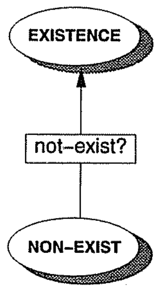
EXISTENCE

In a UNIX interface, an individual of FILE is always an existing file. An individual of FILENAME, however, can be a filename with or without an associated file. As a result, the role, *file*, is present in some individuals of FILENAME and absent in others. To represent this knowledge, a forward chaining rule invokes a procedural test to classify a FILENAME individual under either the concept EXIST or the concept NON-EXIST (see Fig. 6).

Figure 6: EXIST and NON-EXIST



```
(define-concept EXIST (and
        EXISTENCE
        (exactly 1 file)
        (all file FILE)
        (test-c exist? name)))
```



```
(define-concept NON-EXIST (and
        EXISTENCE
        (test-c not-exist? name)))
```

EXISTENCE is a primitive concept that serves as the root of EXIST and NON-EXIST. “Exist?” and “not-exist?”[6] are procedural tests to test the existence of a FILENAME individual. When an individual gets classified under EXIST, it inherits the *file* role. An additional rule fills it with the respective FILE individual. As an example of a procedural test, the source code for “exist?” is listed in Figure 7.

Figure 7: source code to the procedural test "exist?"

```
int exist(cl_object individual, int n, cl_object *arguments)
{
     char *name;
     struct stat buffer;
  /* get the pathname */
     name = cl_ind_attr_string(individual, arguments[0]);
     if (!name)
         error();
  /* test for existence */
     if (stat(name, &buffer) != -1)
         return cl_TEST_TRUE;
     else
         return cl_TEST_FALSE;
}
```

FILE-TYPE and FILENAME-TYPE

Every FILE individual has a file type, of which there are several kinds:

- input file formats, e.g., the Pascal compiler takes input from files with PASCAL type, which has ".p" as filename suffix by convention.
- output file formats, e.g., the "compress" command creates an output file of type COMPRESSED, which has ".Z" as filename suffix.
- system types, e.g., DIRECTORY, NORMAL-FILE, SYMBOLIC-LINK.
- collective types. These are not actual file types, but are useful as generalizations of logical groups of file types. For example, SOURCE is a collective type for C, PASCAL, FORTRAN, ... With these, we can easily express the ideas that "a C compiler reads files of type C and compilers, in general, read files of type SOURCE".

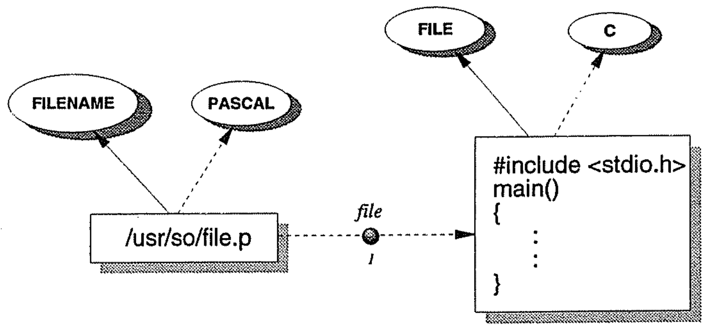
When considering a new concept candidate, such as FILE-TYPE, the first question we must ask is whether it is a concept at all. It seems to be equally plausible to represent file types as a role of

[6] C-CLASSIC does not support a **not** logical operator. It would of course have been more efficient to be able to express (**not** (**test-c** exist? *name*)) rather than to use a second test "not-exist?".

FILE. If file types were implemented as a role of FILE, however, the syntax of C-CLASSIC requires an individual to act as fillers of such a role. This may cause some confusion because it requires us to imagine that there is an individual, say Pascal-ness, for the concept PASCAL; and then to use it as a filler. (We had best avoid the paradoxes of medieval philosophy in our knowledge representation!) Therefore, we chose to implement file types as abstract concepts that do not have associated concrete individuals.

A deeper look at FILE-TYPE tells us that there should be another concept, FILENAME-TYPE. Merely by looking at the filename of a file yields one file type (e.g., ".p" suffix signifies PASCAL) while an in-depth look into the file may yield another file type (e.g., a file named "file.p" may be a C program inside, but the file is unconventionally or incorrectly named). Figure 8 demonstrates our representation scheme. A FILENAME individual has both FILENAME and PASCAL as its parents, while its role *file* has a filler that is an individual under both FILE and C. A problem may arise when this individual is given as input to a command that requires its input to be a PASCAL type. This is elaborated in a later section.

Figure 8: A filename and its associated file may have different types.



An intuitive implementation of FILE-TYPE and FILENAME-TYPE is to create a taxonomy as in Figure 1. Each link has an associated procedural test to test a FILE or FILENAME individual for membership. However, this scheme is expensive when implementing collective types. Take SOURCE as an example. Suppose there are three file types under SOURCE, namely C, PASCAL and FORTRAN with the membership tests C?, pascal? and fortran?. Due to the disjunction restriction of C-CLASSIC[7], we have to supply SOURCE with a membership test, source?. We have no choice but to write a procedure to call C?, pascal? and fortran?. If any returns TRUE, then the individual is a SOURCE. Thus, a PASCAL individual invokes pascal? twice, once in SOURCE and once in PASCAL. The situation propagates as the taxonomy gets more complex.

Our solution is to declare a primitive hierarchy of file types, i.e., a hierarchy without any membership tests, only taxonomic relationships between concepts are defined. Then, we use a computed-concept to ascertain the proper file type (usually at the leaf-level of the hierarchy) of a FILE individual in one pass. As the hierarchy does not contain specific membership information, it is also used as FILENAME-TYPE.

Existing facilities for file type identification include the UNIX "file" command and the UNIX *magic number* mechanism [magic87]. The varieties of file types identified using these facilities are still limited. However, if FILENAME-TYPE, as well as the history of command usage, is taken into account, the result is satisfactory. A more organized research effort in file type classification is the Rufus project from the database fraternity [Messinger91].

FNAME

In UNIX, FILENAME and PATHNAME are treated as the same kind of object most of the time, even though they are distinct concepts in our representation. A command that accepts filename arguments almost always accepts relative (such as "../../file") and absolute (such as "/usr/smith/file") pathnames as well. Therefore, it will be useful to define a concept (called FNAME) that is the union of FILENAME, RELNAME[8] and PATHNAME. Intuitively, FNAME ought to be defined as:

[7] We would like to express SOURCE as (**or** C PASCAL FORTRAN), but we can't.

(**define-concept** FNAME (**or** FILENAME RELNAME PATHNAME))

Unfortunately, C-CLASSIC does not support the disjunctive operator **or**. The alternative is to define explicit rules so that FNAME is inferred for the various kinds of filenames:

(**define-rule** rule001 FILENAME FNAME)
(**define-rule** rule002 RELNAME FNAME)
(**define-rule** rule003 PATHNAME FNAME)

With the definition of FNAME, many explicit computations common to both FILENAME and PATHNAME mentioned above can be united. For example, the following rules, previously defined for both FILENAME and PATHNAME, are defined for FNAME:

(**define-rule** rule004 FNAME EXISTENCE)
(**define-rule** rule005 FNAME (**computed-filler** *suffix* get-suffix))
(**define-rule** rule005 FNAME (**computed-concept** find-filename-type))

It is also useful to define concepts like NAME-of-EXIST-FILE, NAME-of-NON-EXIST-FILE, NAME-of-EXIST-DIRECTORY and NAME-of-NON-EXIST-DIRECTORY when the need arises.

Static Individuals

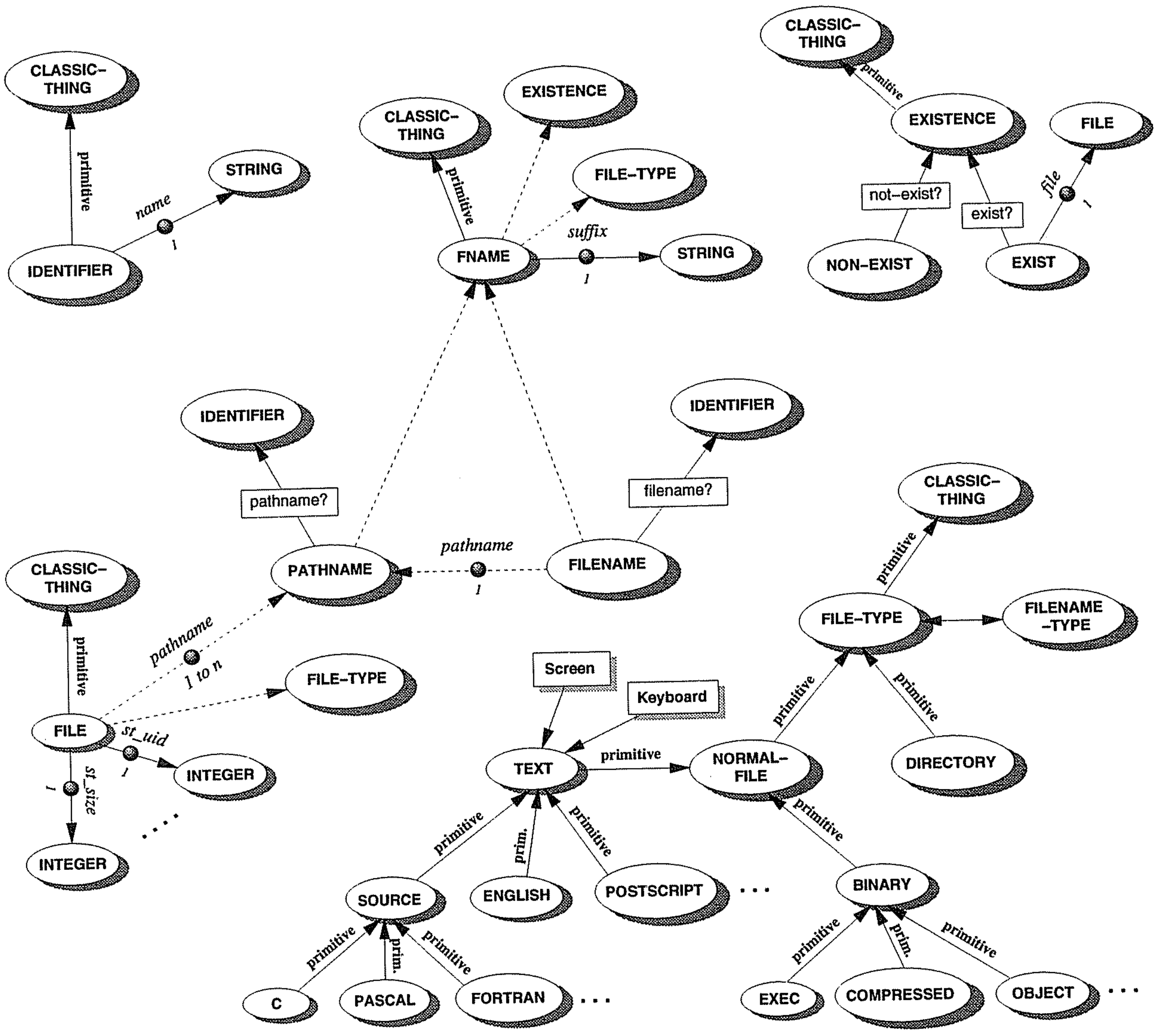
Besides concepts, there are some real objects in UNIX. These should be represented as individuals of the static concepts. For example, users frequently make use of the keyboard as input device, so we should define an individual Keyboard to be an instance of TEXT. Similarly, we should define an individual, Screen, for the output device.

Figure 9 displays the static concepts and individuals mentioned in this section.

8 RELNAME is not defined in this document. Its definition is a generalization of FILENAME because a filename "f" can be viewed as "./f".

Figure 9: Concepts discussed in Section 4.

A semantic network should not be disconnected. The following diagram looks like a disconnected graph solely due to the effort to circumvent space limitations.



5. Level 2: Command Syntax

In one viewpoint, each individual command (also called "utility") can be regarded as a static object. However, commands are special static objects which can benefit from a different level of representation. Some unusual attributes of commands are:

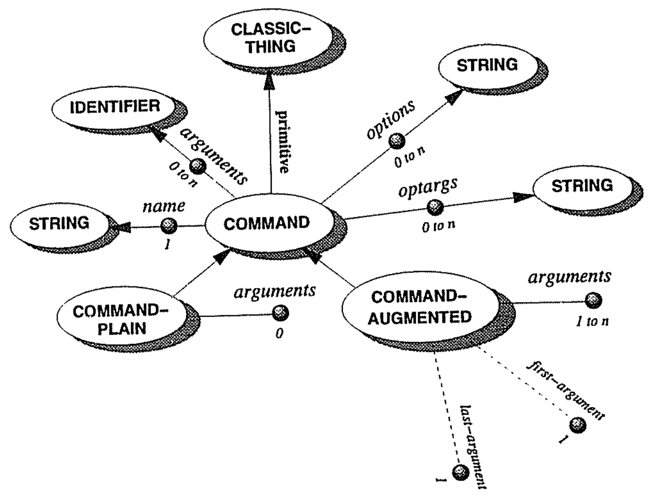
- They are manipulated directly by users. In a sense, they are closer to the surface than certain static objects, like files, in a command user interface.
- They have some standard roles, many of which are unique to commands. For example, most basic commands have *input* and *output* as roles. In UNIX, there is a common kind of role, *options*.
- They spawn a variety of categorical concepts used to classify the commands into meaningful categories based upon their functions. These concepts are abstract descriptions, therefore should not be at the same level as the static objects. For example, the concepts of destructiveness (DESTRUCTIVE) and reversibility (REVERSIBLE).

At this level of representation, we encode the knowledge of the general concept of COMMAND. Moreover, we encode the syntactic knowledge of each individual command. By syntactic, we mean to encode the command invocation format and the categorical concepts to which each command belongs (without describing the meaning of each category to the system).

5.1. Generic COMMAND

Every UNIX command is represented as a C-CLASSIC concept. COMMAND is defined as the root concept of these commands. There are two variants of COMMAND. COMMAND-PLAIN is the class of commands that do not take any arguments, while COMMAND-AUGMENTED contains those with arguments. In addition to the *arguments* role, other roles are *name*, the symbolic name of the command; *options*, the flags to alter a command's behavior, usually entered as "-*x*" where *x* is a character or number; and *optargs*, the auxiliary arguments to the options. Since the first and the last arguments of a command often have special meanings, we define two incidental roles, *first-argument* and *last-argument*, to be filled in by explicit rules. See Figure 10.

Figure 10: Generic COMMAND

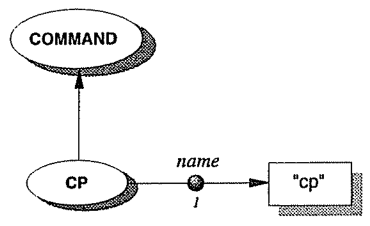


5.2. Specific Commands

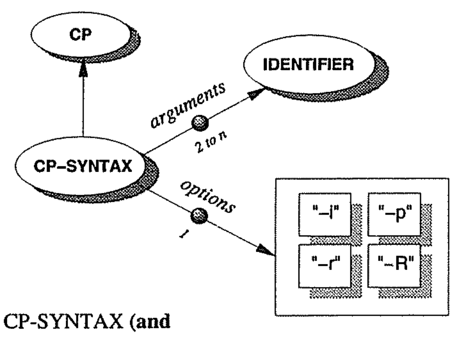
Using COMMAND as a template, each UNIX command is represented by two concepts: Γ and Γ-SYNTAX, where Γ is the command name. Within Γ, we define the symbolic name of the command. Within Γ-SYNTAX, we further define the command with respect to the roles in the COMMAND template. This separation of representation captures the nature of each command from general to specific, for the same reason that we have a hierarchy of knowledge levels.

Figure 11 illustrates the syntax representation of the UNIX command "cp" (used to make copies of files). It is shown that a correct usage of "cp" has at least 2 arguments. These arguments are IDENTIFIERs entered by the user. If the user enters one or more options, these options must be one of "-i", "-p", "-r" or "-R". This information can be found in the UNIX man-pages.

Figure 11: Syntax representation of "cp"



```
(define-concept CP (and
    COMMAND
    (fills name "cp")))
```



```
(define-concept CP-SYNTAX (and
    CP
    (at-least 2 arguments)
    (all arguments IDENTIFIER)
    (all options (one-of "-i" "-p" "-r" "-R"))))
```

This level of representation corresponds to the parsing of a command line by the UNIX shell. The only knowledge here is that of the command components: command name, options[9], option arguments and arguments, which are all that is needed at this level.

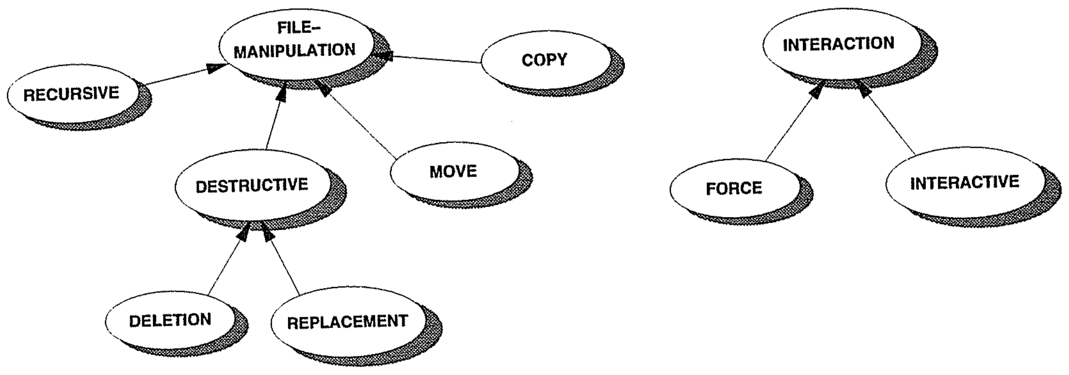
5.3. Categorical Concepts

UNIX utilities can be logically grouped into classes of commands. At this level, we further define a taxonomy of command classes by encoding only symbolic concepts. While these classes have no meaning to the system yet, the knowledge engineer should have a mental model of the semantics of each class. At higher levels, the definition of each class is elaborated to include semantics.

Figure 12 shows two hierarchies of command classes. One is for file manipulation, the other is for human-computer interaction.

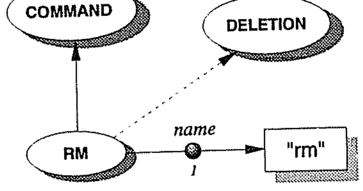
[9] In reality, the shell does not distinguish options from non-options. They are all transferred to the command as arguments. But the use of options is standard across all UNIX utilities, so our distinguishing options from arguments is justified.

Figure 12: Command classes



The technique of abstraction is useful in creating command classes. There are some similar or even identical actions performed by a group of commands, but not by others. For example, many file manipulation commands accept the options "-f" and "-i". The former indicates the command should carry out its operation with force, i.e., it does not confirm with the user even when data are being destroyed. The latter indicates that confirmation should be requested. So, the INTERACTION hierarchy is created.

With these categorical concepts, the command "rm" (remove a file) should have the following syntax (i.e., with DELETION as an incidental property):



```
(define-concept RM (and
        COMMAND
        (fills name "rm"))
 DELETION)
```

6. Level 3: Command Synopsis

UNIX comes with an online documentation facility, activated with a "man" command which displays command usage (called "man-pages"). A man-page for a command is divided into several standardized sections. Typically, one will find the following information in a man-page:

- NAME. The command name and a one-line description.
- SYNOPSIS. Usage(s) of the command and how it(they) should appear in the command line.
- DESCRIPTION. Verbal description of the command usage.
- OPTIONS. A description of how each option modifies the command's behavior.

Of lesser importance are sections such as SEE ALSO, BUGS, AUTHOR and EXAMPLES. At this level of representation, we are most interested in encoding the knowledge from the SYNOPSIS and OPTIONS sections.

6.1. Variants of Command Concepts

The following is the SYNOPSIS section of the command "cp" (from UNIX Programmer's Manual, SunOS 4.1):

cp [**-ip**] *filename1 filename2*
cp **-rR** [**-ip**] *directory1 directory2*
cp [**-iprR**] *filename ... directory*

It is shown that there are three variants of "cp". Each variant accepts different options and argument types (square brackets indicate optional flags or arguments). Each of them is represented as a specialization concept of CP-SYNTAX, and is given a unique concept name.

It is also useful to further refine the synopsis definitions in a simple way. Command usage can differ slightly depending on whether the directory or file given refer existing or non-existing entities. For example, the DESCRIPTION section of the man-page says:

> In the second form, **cp** recursively copies *directory1*, along with its contents and subdirectories, to *directory2*. If *directory2* does not exist, **cp** creates it and duplicates the files and subdirectories of *directory1* within it. If *directory2* does exist, **cp** makes a copy of the *directory1* directory within *directory2* (as a subdirectory), along with its files and subdirectories.[10]

Therefore, the second form needs to be represented by two variant concepts depending on the existence of *directory2*. The same goes for the first form. When *filename2* exists, it should be treated as a REPLACEMENT concept (i.e. replacing an existing file); otherwise, it is a CREATION concept (i.e. creating a new file). For the third form, a "-r" or "-R" option must be given if in place of any of the *filename* arguments a directory name is given; otherwise, it is an error. We represent this by two concepts, one which allows directories in the argument list, and one which does not. The following is the expanded synopsis, augmented with the corresponding concept names:

CP-PLAIN	**cp** [**-ip**] *old-filename new-filename*
CP-DESTRUCT	**cp** [**-ip**] *old-filename1 old-filename2*
CP-DIR1	**cp** **-rR** [**-ip**] *old-directory new-directory*
CP-DIR2	**cp** **-rR** [**-ip**] *old-directory1 old-directory2*
CP-GROUP	**cp** [**-iprR**] *old-filename1 old-filename2 ... old-directory*
CP-GENERAL	**cp** **-rR** [**-ip**] *old-filename old-directory1 ... old-directory2*

6.2. Argument Concepts

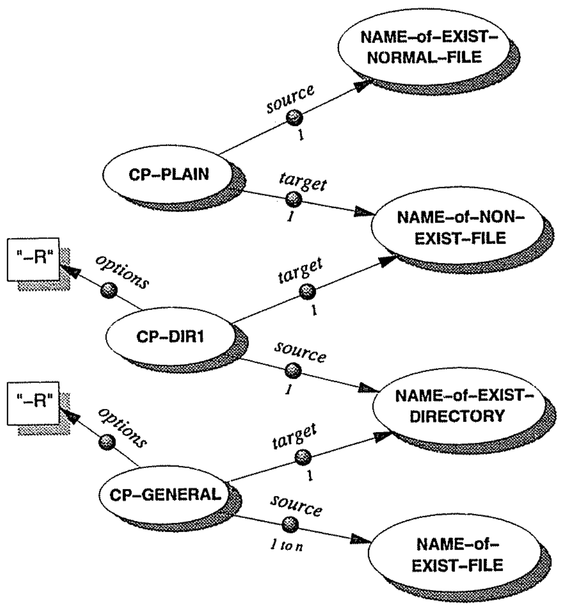
In the previous section, we defined the roles *arguments*, *first-argument* and *last-argument*. Now, the *arguments* role is partitioned into more meaningful roles if the command we are encoding has such logical partitioning. Then, for each command concept, we restrict the values of these roles to the classes of static objects to which they belong.

10 man-page for "cp", SunOS 4.1

In "cp", the arguments are partitioned into two roles: *target* and *source*. The last argument is the *target* and all others are *source*. This is a common terminology for copying. We define explicit rules to fill these roles automatically whenever there is a new instance of the copy command.

At the end of Section 4, we mentioned it is useful to define static concepts like NAME-of-EXIST-FILE and NAME-of-EXIST-DIRECTORY. They come into play when it is time to define *source* and *target* for "cp", as well as arguments of other file manipulation commands. Figure 13 shows a few of these concepts graphically. Each of them is a specialization of CP-SYNTAX. The system is capable of inferring this relationship even if the knowledge engineer does not explicitly state it. Notice the number restriction of *arguments* is different for CP-GENERAL, in accordance with the synopsis.

Figure 13: Several variations of "cp".



```
(define-concept CP-PLAIN (and
        (exactly 1 source)
        (all source NAME-of-EXIST-NORMAL-FILE)
        (exactly 1 target)
        (all target NAME-of-NON-EXIST-FILE)))

(define-concept CP-DIR1 (and
        (exactly 1 source)
        (all source NAME-of-EXIST-DIRECTORY)
        (exactly 1 target)
        (all target NAME-of-EXIST-FILE)
        (fills options "-R")))

(define-concept CP-GENERAL (and
        (at-least 1 source)
        (all source NAME-of-EXIST-FILE)
        (exactly 1 target)
        (all target NAME-of-EXIST-DIRECTORY)
        (fills options "-R")))
```

6.3. Inferring Command Classes from Options

Options (also called flags or switches) of a UNIX command modify the behavior of the command. There are a great variety of options in UNIX. Some options do so little that they are imperceptible to users who don't understand their internal details; some options modify a command tremendously. Some commands do not accept options at all; some commands have so many options that they are considered as undesirable features: "BUGS – *Indent* has even more switches than *ls*"[11], "BUGS – There are many flags that are not documented here. Most are not useful to the general user."[12].

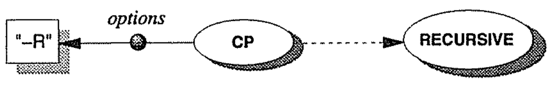
Even though there are a large variety of options, it is possible to find similar options across different commands. They may appear to be different, but have similar function, nevertheless. For instance, due to the recursive nature of the UNIX directory structure, there is an option to cause a file manipulation command to traverse a directory tree recursively. The option "-r" for "rm" (remove), "-R" for "ls" (list files) and either "-r" or "-R" for "cp" (copy) instructs the command to perform its function recursively if one of its arguments is a directory. However, the use of options is not consistent in UNIX. Specifying "-r" to "lpr" (line print) removes the target file upon completion of printing; and to "ls" reverses the order of the listing.

These similar functions are exactly those categorical concepts we defined in Section 5.3. Besides categorical convenience, there are advantages of economy from extracting common functions amongst different commands and setting up stand-alone concepts for them. When we define semantics at the higher levels, it is more economical to define these stand-alone concepts and let the commands inherit semantics from them.

To infer a command class from an option, we create forward chaining rules, such as the following. The incidental concept RECURSIVE is asserted for any instantiation of CP that has "-R" as

[11] man-page for "indent", UNIX 4.3BSD
[12] man-page for "mail", UNIX 4.3BSD

an *options* filler.



```
(define-concept CP-RECURSIVE
        (and CP (fills options "-R"))
    RECURSIVE)
```

6.4. More On File Types

A command often accepts arguments of a fixed type. Sometimes this is explicitly mentioned in the man-page. The following is the man-page for SunOS' C compiler:

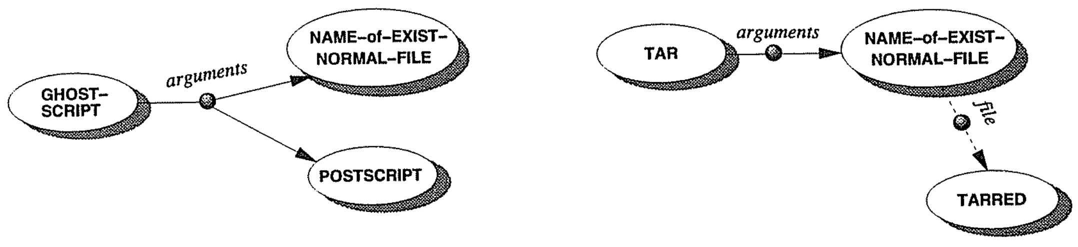
> In addition to the many options, **cc** accepts several types of filename arguments. For instance, files with names ending in **.c** are taken to be C source programs. They are compiled, and each resulting object program is placed in the current directory. The object file is named after its source file — the suffix **.o** replacing **.c** in the name of the object. In the same way, files whose names end with **.s** are taken to be assembly source programs. They are assembled, and produce expansion code template files; these are used to expand calls to selected routines in-line when code optimization is enabled.

"cc" is one of those commands that checks its arguments by looking at their filename suffixes. Since the arguments to "cc" are represented as individuals of the NAME concept, and each NAME individual has an associated FILE-TYPE determined by its suffix (see Figure 8), this FILE-TYPE can be used to check against the value restriction of the *argument* role in the CC concept. If there is an incompatibility between the role and the supplied individual, then the user must have made a mistake. For those commands that do not check for filename suffixes, but nonetheless require certain fixed file type, we may use the FILE-TYPE concept of the associated FILE of the arguments (see the right half of Figure 8). For example, "tar", the tape archive command, works only on archive files. However, an archive file does not always have a suffix; thus, it would be pointless to check the FILE-TYPE associated with the argument of tar (which is determined solely by its suffix). But we may use the FILE-

TYPE of the associated file, as long as that argument is the name of an existing file. Unfortunately, UNIX is not a strongly typed operating system. For a given FILE individual, we could only make an educated guess of its type. Unless there is a foolproof mechanism to recognize a file's type by looking at its content, this latter approach is only an approximation.

We can do a little better if we combine the FILE-TYPE of NAME, T_{name}, with the FILE-TYPE of FILE, T_{file}. For example, the heuristic, "if T_{name} and T_{file} are in the same path of the FILE-TYPE taxonomy, use the more specific one", works most of the time if the files are named correctly. However, when they are incompatible, each case must be considered individually. For instance, at least one version of the UNIX file type classification command, "file", judges a postscript file incorrectly to be "c-shell commands". In this case, T_{name} is more reliable because a postscript file is often ended with ".ps" or ".eps". On the other hand, "file" is usually accurate in judging "tar" files, regardless of their names. So, for the "tar" command, T_{file} should be used. See Figure 14.

Figure 14: Representing the argument types of two commands.



```
(define-concept GHOSTSCRIPT
      (all arguments
            (and NAME-of-EXIST-NORMAL-FILE
                  POSTSCRIPT)))
```

```
(define-concept TAR
      (all arguments
            (and NAME-of-EXIST-NORMAL-FILE
                  (all file TARRED))))
```

The lesson we learn here is that strong typing of a file system would be extremely useful in supporting intelligent help systems, and the typing rules would likely benefit from following ideas found in strong typing of programming languages. But our goal here is not to design a new operating system. Rather it is to support the use of an existing one, warts and all.

6.5. Input and Output

We discussed the difficulties in modeling file types in the previous section. There is one solution we have not yet mentioned. A help system may keep a record of file types by modeling the input and output of a command.

In UNIX, there are many commands that follow the simple input-process-output computation model, where both input and output are files. This simplicity is partly the result of the "keep-it-simple" philosophy of UNIX and partly the result of the pipe mechanism that originates in UNIX. The pipe mechanism works through a class of commands called *filters*. A filter takes an input file from a device called *stdin*, which is normally the keyboard, and writes its output to a device called *stdout*, which is normally the screen. More than one filter can be chained in series in a command line. When this is done, the stdout of the first command becomes input to the stdin of the second, the second's stdout becomes the third's stdin, etc. There are also ways to redirect the stdin of the first filter and the stdout of the last filter to files. This makes it possible to apply several simple commands sequentially to solve a complex task.

For these filters, the input and output file types can be modeled in the same manner as we model the arguments, only that the roles *input* and *output* are used. If a command completes successfully, the output file is guaranteed to have the type described by *output*. This information is kept by a truth maintenance component embedded in the help system.

There are also commands that may not be used conventionally as filters, but take input from stdin and send output to stdout, nonetheless. The same mechanism should be used to model their input and output files. For example, there is no ordinary way to tell whether a file is encrypted by the command "crypt", unless the system keeps track of this information.

7. Using The Knowledge

We have introduced a knowledge representation formalism for UNIX utilities in the previous sections. There are many ways to make use of this knowledge in the context of user support. This section presents an ideal UNIX shell that exemplifies how the formalism can be applied to aid users.

The front-end of the shell is an ordinary UNIX shell, such as "tcsh". The back-end is an expert system which has knowledge about UNIX encoded in the above form, as well as inference mechanisms described in the following sections. It acts as a knowledgeable source for the ordinary shell to consult. The front-end takes commands in the normal fashion. The difference from an ordinary shell is, this new one does not always blindly follow the user's command. Instead, it will consult with the back-end to see if there is need to correct or augment the command.

7.1. Initialization

When the system starts up, it searches the file system for every existing object that is modeled. A C-CLASSIC individual for each such object is created. For instance, a FILE individual is created for each existing file. Rules are fired to fill in properties such as file size and inode number. Ideally, later changes to these properties will be reported to the back-end immediately[13]. Then further rules classify it under the FILE-TYPE taxonomy. When the knowledge base is encoded in the way described above, these rules are spontaneous and hidden. A simple C-CLASSIC statement such as

(define-individual FILE0019 **(and** FILE **(fills** *pathname* "/usr/smith/file.c"**)))**

will start the process.

[13] Unfortunately, the knowledge retraction facilities in C-CLASSIC as well as most other practical KR systems are not versatile enough for an ideal implementation. Either we have to stick with a non-changing world, or we have to provide ad hoc facilities on a case by case basis, however unwillingly.

7.2. Processing A Simple Command Line

A command line is read in as usual (together with all up-to-date user support facilities such as history substitution, alias substitution and spelling correction). We know a simple UNIX command line is of the following form:

command-name [*arguments*...] [< *input*] [> *output*]

We create a new individual of COMMAND to represent this newly entered command line. The *name* role of the individual is filled with the **command-name** and the *arguments* role is filled with the arguments; each is represented by an IDENTIFIER individual. The corresponding *input* and *output* roles are filled if necessary. The classification process will start to classify this COMMAND individual under a specific command concept.

The arguments will be classified at different levels. At the static object level, each is classified under all possible static concepts that do not conflict with its appearance. In other words, if an argument looks like a filename, it will be classified under FILENAME. If it looks like a user's login name, it will be classified as such. C-CLASSIC does this by searching the descendant concepts of IDENTIFIER. It is not only possible but probable that an argument will be classified under many static concepts, even though the user most probably has intended only one of them. This is appropriate because, without higher-level knowledge, all we can do to guess its parent concept is judge by its appearance. At higher levels, the value restrictions for the *arguments* and other related roles will be used to reduce the ambiguity.

7.3. Type Checking

If we view a command invocation in UNIX as though it is a function call in a computer language, then the arguments to the command are analogous to the parameters passed to the function. Type checking is an effective means in correcting accidental syntax mistakes; therefore many modern programming languages incorporate this feature. Some UNIX commands also perform type checking on their arguments. Unfortunately, different commands do type checking in their own ways. Thus, even for the same kind of errors, different commands produce different error messages and follow different error recovery procedures. It would be much more consistent if such type checking were performed by the shell.

Figure 15: Processing a simple command line.

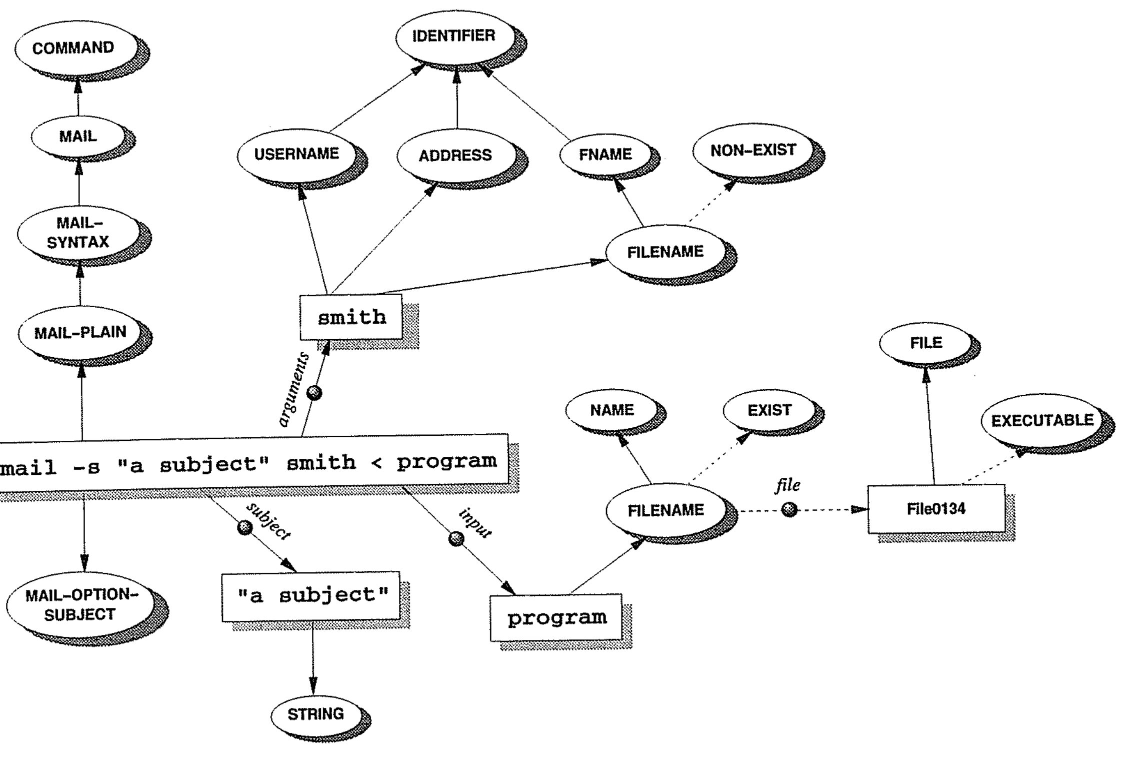
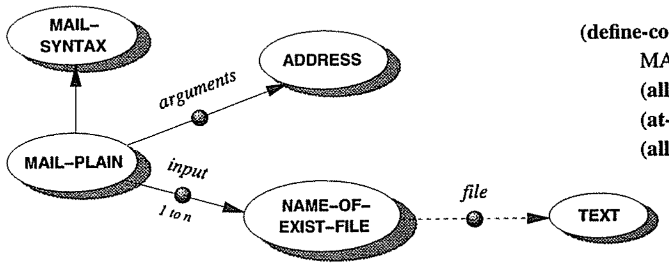


Figure 15 shows how a simple command is processed by the shell. The command line is

mail -s “a subject” smith < program

which should send the file named “program” to the user named “smith”. The entire command line is first classified under COMMAND and is later inferred to be under MAIL-PLAIN. The “-s” switch supplies a subject heading to the mail message. Because of this, it also gets classified under MAIL-OPTION-SUBJECT. An individual of STRING is automatically created to accommodate the subject itself. An IDENTIFIER individual is created for each argument. In this example, "smith" is the only argument. It is later recognized to be USERNAME, ADDRESS and NON-EXIST FILENAME. The last element of the command line is the input file, first classified as an IDENTIFIER, then as a FILENAME with an associated file. The *file* role is subsequently filled with the particular individual, say File0134, which is known to be an EXECUTABLE file.

Figure 16: Definition of MAIL-PLAIN



```
(define-concept MAIL-PLAIN (and
        MAIL-SYNTAX
        (all arguments ADDRESS)
        (at-least 1 input)
        (all input
                (and NAME-of-EXIST-FILE
                        (all file TEXT)))))
```

Type checking for a simple command line involves comparing the input command line to the command concept in the knowledge base. Figure 16 shows the definition of MAIL-PLAIN. Comparing the value restrictions of the roles in this diagram with the filled-in individuals in Figure 15, the system will be able to make two deductions:

- the IDENTIFIER individual "smith" is now known to be an ADDRESS. The knowledge that it is also a USERNAME and a FILENAME is superfluous.
- the *input* role for MAIL-PLAIN is restricted to be a NAME with an existing file that has TEXT as its file type. However, the actual command line has a file that has EXECUTABLE as its file type. C-CLASSIC will be able to find out EXECUTABLE is under BINARY, therefore not compatible with TEXT[14]. Thus, a type conflict. If the input file were classified as of C source code type, it would have been compatible with TEXT.

[14] TEXT is defined as a file that does not contain 8-bit bytes. BINARY is a file that consists of at least one 8-bit byte. Therefore, TEXT and BINARY are mutually exclusive.

The UNIX "mail" command actually takes any type of files as input. However, due to the inability of some mail servers to handle binary data, it is generally a good advice to, at least, inform the user that his data may not be transferred successfully. The above illustrates how a shell of the kind we have in mind can, by using our representation scheme, be enabled to provide such advice.

This section addresses only how to detect a type conflict. After a detection, there are many possible responses since there are different kinds of type conflicts. Some conflicts are fatal – there is no valid way to process that command. Some conflicts are only unconventional – the user really wants to carry that out. How to resolve a type conflict is a consideration of higher semantic levels.

7.4. Complex Syntax Checking

Since our representation so far deals with syntactic knowledge, many other kinds of syntactic problems can be detected, besides file type checking. The following is a complex command line:

```
compress big-file | uuencode | mail -s A Compressed File smith@cs
```

The user is trying to send a file to smith@cs. Realizing the file is large, he/she compresses the file before sending. He/She also realizes a compressed file must be turned into a text file by "uuencode" before sending. Despite the intuitive appearance of the command line, it has three hidden syntax mistakes which are, fortunately, detectable using our representation scheme. The syntax problems are:

- The command "compress" in the above form is disk-based. It compresses the file and renames it "big-file.Z". The above command line requires the output of "compress" sent to stdout, while "compress" in the above form does not generate any output unless the flag "-c" is given. This is detected by the *output* role of the concept COMPRESS-SYNTAX.
- The command "uuencode" requires at least one argument. It is used so the "uudecode" command knows what filename to use when decoding. This is detected by the number restriction of the *arguments* role of UUENCODE-SYNTAX.

- The "mail" command in the above form regards the single letter "A" as the subject and treats the remaining "Compressed", "File" and "smith@cs" as recipients. In this case, since the value restriction of *arguments* of MAIL-SYNTAX requires all the recipients to be a valid user address, and neither "Compressed" nor "File" is classified as USERNAME or ADDRESS (this is done by searching through a list of valid user names), the mistake is noted. If, by chance, there is a user whose login name is "Compressed" and one named "File", then even our representation cannot detect the problem.

The correct command line, which is much less intuitive, should be:

```
compress -c big-file | uuencode big-file | mail -s "A Compressed File" smith@cs
```

The above example demonstrates a non-trivial error correction task that might be difficult even for experienced users, but can be done by the syntactic levels of our proposed knowledge-based shell. Besides correcting the command line, the system will also explain the reasons behind the modification.

7.5. Abstracting Common Features

Another potential use of the knowledge representation scheme we have chosen is to make use of the inheritance feature to factor common features out of the utilities. Recall we create command concepts by abstracting similar features from the commands. Some features across commands are identical in nature. For example, the "-i" option in a destructive file manipulation command signals the command to prompt the user before any destructive action. The routine to prompt the user is duplicated in every command that supports this option. In order to reuse codes and to provide a more consistent interface, the next generation of UNIX commands would be well advised to take advantage of the knowledge base, such as we propose, with similar features residing in a higher knowledge level and being inherited by lower level concepts that use them.

Although this idea stems from object-oriented programming languages, it is not exactly the same. For example, the Smalltalk/V language supplies a PROMPTER class in which the application programmer uses to display a message and prompt for a yes/no answer. If we do not use the knowledge-based approach, but only use Smalltalk/V to enhance UNIX utilities, then the utility programmer still needs to instantiate an individual from the PROMPTER class when a utility receives a "-i" option. This may reduce code redundancy and improve consistency, but not as thoroughly as does our approach. In the knowledge-based approach, the utility programmer does not need to worry about any prompter routine at all. After he/she has finished programming the essential part of the utility, he/she declares his new utility to be DESTRUCTIVE, and declares the "-i" option to be INTERACTIVE. Then the knowledge-based shell will automatically run the prompter routine at the right time. Since this application requires major revision of UNIX utilities, we only point out the feasibility without an implementation plan.

8. Conclusion

This research began as an effort to create a knowledgeable mixed-initiative UNIX command shell that is different from question answering consultant programs or template-driven help systems. We found that the very foundation of such a command shell demands a formal knowledge representation scheme for UNIX utilities. As for much knowledge representation, we found it useful to arrange knowledge of UNIX in a hierarchy of knowledge levels. We have identified five levels of knowledge abstraction:

- Level 1: static objects.
- Level 2: command syntax.
- Level 3: command synopsis.
- Level 4: command semantics.
- Level 5: UNIX semantics.

Levels 1, 2 and 3 are the syntactic parts of entire representation scheme. They also lay down the building blocks for Levels 4 and 5 that deal with semantic properties of UNIX.

We have here described and exemplified our representation formalism for Levels 1 through 3. The formalism is based on classification technology. We have introduced the classification system, C-CLASSIC, and have shown how it is used to encode the syntactic knowledge of UNIX utilities. Finally, we have illustrated some possible uses of the knowledge representation in the context of user support.

Besides the examples given in the previous section, the most important contribution of the formalism is to lay down the basis for representing higher levels of UNIX knowledge. Levels 4 and 5 are discussed in a second paper. It will be seen that there are fundamental differences between syntactic knowledge and semantic knowledge, and that the two kinds of knowledge require some major extensions to C-CLASSIC. In particular, C-CLASSIC does not handle "part-of" relations, nor does it provide complex heuristic rule mechanisms.

References

Barrett88. Barrett, Edward (ed.), *Text, Context, and HyperText: Writing with and for the Computer,* MIT Press, Cambridge, MA, 1988.

Barrett89. Barrett, Edward (ed.), *The Society of Text: Hypertext, Hypermedia, and the Social Construction of Information,* MIT Press, Cambridge, MA, 1989.

Borg90. Borg, Kjell, ''IShell: A Visual UNIX Shell,'' in *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, End User Modifiable Environment, pp. 201-207, 1990.

Borgida89. Borgida, Alex, Brachman, Ronald J., McGuinness, Deborah L., and Resnick, Lori Alperin, ''CLASSIC: A Structural Data Model for Objects,'' *Proceedings of ACM-SIGMOD-89*, Portland, Oregon, 1989.

Brachman77. Brachman, Ronald J., *A Structural Paradigm for Representing Knowledge,* Ph.D. thesis, Harvard University, May, 1977.

Brachman83. Brachman, Ronald J., Fikes, Robert, and Levesque, Hector J., ''KRYPTON: A Functional Approach to Knowledge Representation,'' *IEEE Computer*, September, 1983.

Brachman91. Brachman, Ronald J., McGuinness, Deborah L., Patel-Schneider, Peter F., Resnick, Lori Alperin, and Borgida, Alexander, ''Living With CLASSIC: When and How to Use a KL-ONE-like Language,'' in *Principles of Semantic Networks: Exploration in the Representation of Knowledge*, ed. John F. Sowa, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

Brachman85. Brachman, Ronald J. and Schmolze, James. G., ''An Overview of the KL-ONE Knowledge Representation System,'' *Cognitive Science*, vol. 9, no. 2, pp. 171-216, 1985.

Cesta91. Cesta, Amedeo and Romano, Giovanni, ''Explanations in an Intelligent Help System,'' in *Human Aspects in Computing: Design and Use of Interactive Systems and Information Management*, ed. Hans-Jorg Bullinger, pp. 925-929, Elsevier, Amsterdam, 1991.

Conklin87. Conklin, Jeff, ''Hypertext: An Introduction and Survey,'' *IEEE Computer*, vol. 2, no. 9, pp. 17-41, Sept. 1987.

csh89. csh, UNIX man page, *csh – A Shell (Command Interpreter) With a C-like Syntax and Advanced Interactive Features,* UC Berkeley BSD 4.3, 1989.

Doane92. Doane, Stephanie M., ''Prompt Comprehension in UNIX Command Production,'' *Memory & Cognition*, vol. 20, no. 4, pp. 327-343, 1992.

Eide91. Eide, Eric, ''Using Context to Improve Command Language Interfaces,'' Master's Thesis Proposal, Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, 1991.

Fischer85. Fischer, Gerhard, Lemke, Andreas, and Schwab, Thomas, ''Knowledge-based Help Systems,'' *Proceedings of ACM SIGCHI'85*, ACM, 1985.

Gonzalez93. Gonzalez, Avelino J. and Dankel, Douglas D., *The Engineering of Knowledge-based Systems: Theory and Practice,* Prentice-Hall, Englewood Cliffs, NJ, 1993.

Gordon88. Gordon, Michael, ''Probabilistic and Genetic Algorithms for Document Retrieval,'' *Communications of the ACM*, vol. 31, no. 10, 1988.

Hegner87. Hegner, Stephen J., ''Knowledge Representation in Yucca-II: .Exploiting the Formal Properties of Command Language Behavior (draft version),'' in *Workshop on Knowledge Representation in the UNIX Help Domain (unpublished)*, ed. Robert Wilensky, December 1987.

Henderson86. Henderson, D. A. and Card, Stuart K., ''Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-based Graphical User Interface,'' *ACM Transactions on Graphics*, vol. 5, no. 3, pp. 211-243, July, 1986.

Horton90. Horton, W. K., *Designing & Writing Online Documentation: Help Files to Hypertext,* John Wiley & Sons, Inc., New York, NY, 1990.

Jerrams-Smith89. Jerrams-Smith, Jennifer, "An Attempt to Incorporate Expertise About Users Into an Intelligent Interface for Unix," *International Journal of Man-Machine Studies*, vol. 31, pp. 269-292, September, 1989.

Jones88. Jones, John and Millington, Mark, "Modelling UNIX Users with an Assumption-based Truth Maintenance System: Some Preliminary Findings," in *Reason Maintenance Systems and Their Applications*, ed. Gerald Kelleher, Ellis Horwood, Chichester, 1988.

Jones88a. Jones, John, Millington, Mark, and Ross, Peter, *Understanding User behaviour in Command-driven Systems,* pp. 226-235, Chapman and Hall Computing, London, 1988.

Kass88. Kass, Robert and Finin, Tim, "Modeling the User in Natural Language Systems," *Computational Linguistics*, vol. 14, no. 3, September, 1988.

Kernighan81. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Computer*, vol. 14, no. 4, pp. 25-34, Apr. 1981.

Mac Gregor91. Mac Gregor, Robert, "The Evolving Technology of Classification-Based Knowledge Representation Systems," in *Principles of Semantic Networks: Exploration in the Representation of Knowledge*, ed. John F. Sowa, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

magic87. magic, UNIX man page, *magic – file Command's Magic Number File,* UC Berkeley BSD 4.3, 1987.

McCune85. McCune, B. P., Tong, R. M., Dean, J. S., and Shapiro, D. G., "RUBRIC: A System for Rule-based Information Retrieval," *IEEE Trans. Software Engineering SE-11*, vol. 9, 1985.

McDonald90. McDonald, Christopher S., "An Executable Formal Specification of a UNIX Command Interpreter," in *Engineering for Human-Computer Interaction: Proc. of IFIP TC2/WG 2.7*, ed. Gilbert Cockton, North-Holland, 1990.

Messinger91. Messinger, Eli, Shoens, Kurt, Thomas, John, and Luniewski, Allen, "Rufus: The Information Sponge," *Research Report RJ 8294 (75655)*, IBM Almaden Research Center, Aug. 1991.

Minsky85. Minsky, Marvin, "A Framework for Representing Knowledge," in *Readings in Knowledge Representation*, ed. Hector J. Levesque, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1985.

Nessen89. Nessen, Erich, "SC-UM: User Modeling in the SINIX Consultant," *Applied Artificial Intelligence* , vol. 3, no. 1, 1989.

Nielsen90. Nielsen, J., *Hypertext & Hypermedia,* Academic Press, Inc., San Diego, CA, 1990.

Norman81. Norman, Donald A., ''The trouble with UNIX: the user interface is horrid,'' *Datamation*, vol. 27, 1981.

Norman85. Norman, Donald A., ''Four Stages of User Activities,'' in *Human-Computer Interaction - Interact '84*, ed. Brian Shackel, Elsevier Science Publishers (North-Holland), 1985.

Quilici88. Quilici, Alex, Dyer, Michael G., and Flowers, Margot, ''Recognizing and Responding to Plan-oriented Misconceptions,'' *Computational Linguistics*, vol. 14, no. 3, September, 1988.

Resnick91. Resnick, Lori Alperin, Borgida, Alex, Brachman, Ronald J., McGuinness, Deborah L., Patel-Schneider, Peter F., and Zalondek, Kevin C., *CLASSIC Description and Reference Manual For the COMMON LISP Implementation - Version 1.2,* AT&T Bell Lab., October, 1991.

Shneiderman92. Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction 2nd Edition,* Addison-Wesley, Reading, MA, 1992.

Shrager82. Shrager, Jeff and Finin, Tim, ''An Expert System that Volunteers Advice,'' *Proceedings of the 2nd Annual National Conference on Artificial Intelligence AAAI-82*, 1982.

Stefik89a. Stefik, Mark J. and Bobrow, Daniel G., ''Object-Oriented programming: Themes and Variations,'' in *AI Tools and Techniques*, ed. Mark H. Richer, Ablex Publishing Corp., Norwood, NJ, 1989.

Stefik89. Stefik, Mark J., Bobrow, Daniel G., and Kahn, Kenneth M., ''Integrating Access-Oriented Programming Into a Multiparadigm Environment,'' in *AI Tools and Techniques*, ed. Mark H. Richer, Ablex Publishing Corp., Norwood, NJ, 1989.

tcsh91. tcsh, UNIX man page, *tcsh – C Shell With File Name Completion and Command Line Editing*, 1991.

Tesler92. Tesler, Joel, *UNIX man page, fsn – File System Navigator,* Silicon Graphics Inc., 1992.

Weixelbaum93. Weixelbaum, Elia, *C-CLASSIC Reference Manual Release 1.3,* AT&T Bell Labs, April 21, 1993.

Wilensky84. Wilensky, Robert, Arens, Yigal, and Chin, David N., ''Talking to UNIX in English: An Overview of UC,'' *Communications of the ACM*, vol. 27, no. 6, 1984.

Woodroffe88. Woodroffe, Mark R., ''Plan Recognition and Intelligent Tutoring Systems,'' in *Artificial Intelligence and Human Learning: Intelligent Computer-aided Instruction*, ed. John Self, pp. 212-225, Chapman and Hall Computing, London, 1988.

Woods91. Woods, William A., ''Understanding Subsumption and Taxonomy: A Framework for Progress,'' in *Principles of Semantic Networks: Exploration in the Representation of Knowledge,* ed. John F. Sowa, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

Woods90. Woods, William A. and Schmolze, James G., ''The KL-ONE Family,'' *Technical Report TR-20-90*, Center for Research in Computing Technology, Harvard University, Aug. 1990.