Representation of Command Language Behavior for an Operating System Consultation Facility

Stephen J. Hegner
Department of Computer Science and Electrical Engineering
Votey Building
University of Vermont
Burlington, VT 05405

(802)656-3330
hegner%uvm.edu@csnet-relay
..!{decvax,ihnp4}!dartvax!uvm-gen!hegner

This paper appeared in *Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications*, San Diego, 14-18 March 1988, pp. 50-55.

ABSTRACT

Yucca-II is a consultation system which is designed to provide the UNIX user, through a natural language interface, with detailed expert advice on the use of the UNIX command language. The most salient architectural feature of Yucca-II is its sharp division into a natural language front end and a formal knowledge base back end, coupled by a formal dynamic query language. This paper describes the knowledge representation techniques used in the back end to represent the appropriate aspects of command language behavior. The emphasis is upon the particular problems encountered in the representation of command language behavior which are not easily addressed within more general knowledge representation frameworks. The most significant include the display object problem, which mandates distinction between an object (such as the contents of a directory) and a display of same, the problem of representing actions whose occurrence is more significant than the details of what they do (such as putting pageheaders on a file), and the problem of representing the interconnective behavior of a large number of commands and options in a succinct fashion.

1. Introduction

Yucca-II is a consultation system for the UNIX[1] operating system. Its primary goal is to provide expert help to users who have some general understanding of the nature of operating and file systems, but who lack detailed knowledge of the behavior of UNIX. The capabilities of Yucca-II go beyond those of the traditional operating system help facility in several ways. First, it provides a natural language interface. As such, it does not force the user to specify a command name or keyword, but rather allows him to communicate with the consultation system in the same way that he would with a human consultant. Second, it is fundamentally designed to answer *process queries* [Scra75], such as "How do I print a file with pageheaders?" or "What happens if I try to delete an unwritable file?". In this way it is distinguished from help systems such as Cousin ([Haye82a], [Haye82b]), which can only provide help in situations in which the command name is known. Third, it is able to provide detailed help. Thus, a user asking "How do I print a file with line numbers on the laser printer?" will be given a specific command sequence to execute, rather than just manual pages describing the *cat* and *lpr* commands. Fourth, Yucca-II has limited ability to understand queries within the context of other operating systems. For example, a TOPS-20[2] user asking about on-line expiration time will be told that UNIX does not support automatic archiving of inactive files. Finally, Yucca-II is able to answer static as well as process queries. For example, queries such as "What is a pipe?" or "What are i-nodes?" can be answered. The most salient architectural feature of Yucca-II is the clear-cut separation of the process of understanding a query from that of solving it. This is manifest in Figure 1 below.

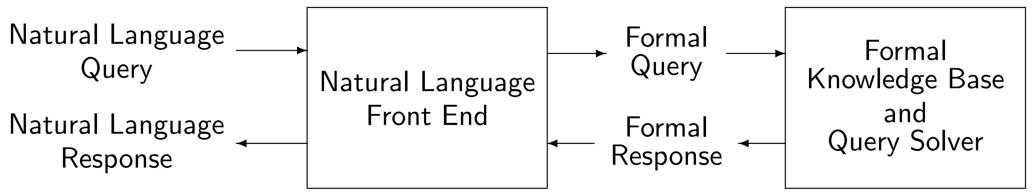
Natural Language Query
Natural Language Response
Natural Language Front End
Formal Query
Formal Response
Formal Knowledge Base and Query Solver

Figure 1: Overall architecture of Yucca-II.

The *natural language front end* provides the understanding component of the system. It translates the user's natural language query into a formal query in the language *Osquel*, and generates a natural language version of the formal response to the query. The *formal knowledge base and query solver*, on the other hand, embodies a completely formal representation of UNIX properties and behavior. In some ways, this architecture is similar to that employed in natural language interfaces to database systems, such as PLANES

[1]UNIX is a trademark of AT&T Bell Laboratories.

[2]TOPS-20 is a trademark of Digital Equipment Corporation.

[Walt78], TEAM [Gros83], and IRUS [BaMS86]. However, these are interfaces to preexisting databases, while Yucca-II involves the integrated design of both components.

There are at least two advantages to such a separation of understanding and solution. First, the formalization of the query provides an unambiguous statement of exactly how the query was understood, quite independently of how it is to be solved. This is extremely useful in support of paraphrasing the query meaning to the user. Second, distinct knowledge representation methods, optimized for each task, may be employed, so that a less optimal compromise solution is unnecessary.

Yucca-II is being implemented in Common Lisp, and is a collaborative effort. The natural language front end is being designed and implemented by researchers at New Mexico State University. The development of the query language and formal knowledge base and query solver (the *back end*) is being led by the author at the University of Vermont. This paper describes only the design of the back end; the front end is described elsewhere ([McKe86], [McWi87]).

From a knowledge representation point of view, Yucca-II is distinguished by its customized and deep model of UNIX. Whereas the UC system [WiAC84] [WMAC86] uses a general cognitive knowledge framework in which UNIX concepts are expressed, and the systems CMS-HELP [YuLo84] and TVX [BiCa85] employ rule-based frameworks to model operating system concepts for VM/CMS[3] and VAX/VMS[4], respectively, the knowledge representation and inference methods employed in Yucca-II are specially tailored to model the behavior of operating systems. Indeed, it is a major thesis of our work that to answer *detailed* queries efficiently, such a specialized knowledge representation methodology is necessary. An effort which shares this philosophy is the Programmer's Apprentice project ([Rich81] [Wate85]); however, that system is designed to provide detailed interactive help on complex programming problems, while Yucca-II is designed to provide complete solutions to relatively simple command-language queries.

Yucca-II is an outgrowth of two earlier systems, UCC [DoHe82] and Yucca [HeDo84]. UCC was a prototype system, designed to identify the needs of an operating system consultation system. It combined a relatively straightforward ATN-based front end with shallow representation of UNIX command semantics. Yucca was an attempt to extend UCC by implementing a much more sophisticated back-end knowledge base. However, as development proceeded it became apparent that to support a sophisticated back end, a much more sophisticated front end would also be necessary. In addition, many shortcomings of the original Yucca back-end were uncovered in the process. As a result, both components are being completely redesigned from scratch in Yucca-II.

The back-end knowledge base is divided into two main components. The *static component* (which is described in Section 3) comprises a formalization of the basic objects present in an operating system, such as files, directories, users, print queues, and the like. Complete knowledge of this component is shared with the front end, since it is necessary for the

[3]VM/CMS is a trademark of IBM Corporation.

[4]VAX/VMS is a trademark of Digital Equipment Corporation.

expression of queries in Osquel (which is discussed in Section 2). In a loose sense, this component corresponds to the data definition language (DDL) of a traditional database system. The *dynamic component* (which is described in Section 4) contains descriptions of command behaviors, which are formalized as dynamic statements using the static component as a base. The knowledge contained in the dynamic component is invisible to the front end, save through formal responses to queries. Thus, in a loose sense, the dynamic component corresponds to the actual database in a traditional database system. The query solution methodology is discussed in Section 5.

2. The Formalization of Process Queries

In a traditional relational database management system, (nonprocedural) query solution is tantamount to binding free variables in a well-formed formula in a first-order language [Codd72]. Analogously, in process queries, query solution becomes the problem of binding free variables in a well-formed formula in a dynamic logic. To illustrate by example, the formal query in Osquel corresponding to "How do I print a file with pageheaders on the laser printer?" is shown in Figure 2 on the following page.

In this query, `P` is the *precondition*, `F` the *action*, `Q` the *postcondition*, and `A` the *actor*. The precondition `P` declares the existence of two mutable object instances, a plain file `#I` and a printer queue `%laser-print-queue`. The variable `#I` is *external*, meaning that it is a generic instance, having no more properties than those bound by the query. The variable `%laser-print-queue` is a *global constant*, identifying the print queue of the system laser printer. Also asserted is the fact that the contents of file `#I` is a `visible-byte-sequence`, which means that it consists of printable characters. The postcondition `Q` declares the existence of three mutable objects. The file `#I` is declared with the same version number (`0`) as it is in `P`. This is an explicit frame axiom which states that the file is not to change in the transition from `P` to `Q`. Although there are implicit frame axioms present, they may be overridden in certain cases in which one change "triggers" another; an explicit frame axiom prevents such overriding. The print queue version is asserted to be `1`. Thus, the constitution of the print queue in `Q` may be different from that asserted in `P`. A new object instance, a print queue entry `#PQE`, is also asserted in `Q`. The first "`=`" statement asserts that version `1` of the print queue is the same as version `0`, save that `#PQE` has been inserted. The second asserts that `#PQE` has `%user` as owner. (`%user` is a global constant identifying the current user.) The final "`=`" statement asserts that the contents of `#PQE` is that of the file `#I`, but with pageheaders added.

The declarations of `F` and `A` are straightforward; `F` may be an arbitrary command sequence, and `A` must be the current user.

The *secondary* entries in `P` and `Q` are used to identify any additional conditions which must be met in order that the solution be valid, including the overriding of implicit frame axioms. In a solution to this example query, `?SPP` might be bound to a statement that the file `#I` be readable by the actor.

```
(query:
  (dynamics: ( (((state: P) (action: F) (state: Q)) (actor: A)) ) )
  (query-variables: ?SPP ?SPQ ?CS)
  (external-variables: #I)
  (local-variables: #PQE)
  (define: P
   (AND:
     (instance: (class: plain-file)
                (identification: ((name: #I) (version: 0))) )
     (instance: (class: visible-byte-sequence)
                (identification:
                  (retrieve: record-entry:
                               (field: contents)
                               (source: ((name: #I) (version: 0))))))
     (instance: (class: print-queue)
                (identification: ((name: %laser-print-queue) (version: 0))))
     (secondary: ?SPP)))
  (define: Q
   (AND:
     (instance: (class: plain-file)
                (identification: ((name: #I) (version: 0))))
     (instance: (class: print-queue)
                (identification: ((name: %laser-print-queue) (version: 1))))
     (instance: (class: print-queue-entry)
                (identification: ((name: #PQE) (version: 1))))
     (= (retrieve: entire-value:
                   (source: ((name: %laser-print-queue) (version: 1))))
        (enqueue: (retrieve: entire-value:
                     (source: ((name: %laser-print-queue) (version: 0))))
                  (retrieve: entire-value:
                     (source: ((name: #PQE) (version: 1))))))
     (= (apply-filter: (name: paginated)
                       (constraint:
                         %paginated-standard-pageheaders)
                       (argument: (retrieve: record-entry:
                                    (field: contents)
                                    (source: ((name: #I) (version: 0))))))
        (retrieve: record-entry: (field: contents)
                                 (source: ((name: #PQE) (version: 1)))))
     (= (retrieve: record-entry:
                  (field: owner)
                  (identification: ((name: #PQE) (version: 1))))
        (retrieve: entire-value:
                  (source: %user)))
     (secondary: ?SPQ)))
  (define: F
     (instance: (class: command-sequence) (identification: ?CS)))
  (define: A
     (instance: (class: user) (identification: %user))) )
```

Figure 2: Example query: “How do I print a file with pageheaders on the laser printer?”

Queries are classified by primary unknown. The above query is an *unknown action* query, because the action `F` is the primary unknown. Unknown precondition, postcondition, and actor queries are also possible, although unknown action queries are given primary solution support. *Static queries*, as mentioned in the introduction, are also supported in a straightforward manner, but are not elaborated here.

3. The Database of Static Objects

In principle, the organization of the static object database is along the lines of a traditional object hierarchy, as enforced by systems such as KL-ONE [BrSc85]. In particular, there is type classification with a formal generalization hierarchy and property inheritance, and distinction between object classes and individual instances. However, the concepts modelled within an operating system framework have much more specific and formalizable structure than do those of the general nature which KL-ONE was intended to subsume, and so our framework is far more specialized. In the current version of the static database, there are definitions for well over 100 different UNIX object classes, describing the overall system, users, terminal connections, processes, files (including directories and devices), byte sequences (file contents), commands, printers and print queues, and basic notions such as time and byte contents. Specific examples occurring in the query in the previous section include `plain-file`, `visible-byte-sequence`, `print-queue`, and `print-queue-entry`. In addition, there is a basic collection of extant global object instances, describing items which are always presumed to be present. Two examples occurring in the example query in the previous section are the current user (denoted `%user`) and the laser printer queue (denoted `%laser-print-queue`). The basic syntactic structure of an object class definition is given in Figure 3 below.

```
(object-class: <object-name>
   (hierarchy: <hierarchical-descriptor>)
  [(structure: <structural-descriptor>)]
  [(filters: <filter-descriptor>+)]
  [(generation-structure: <generation-structure-descriptor>+)]  )
```

Figure 3: Object class definition.

Here we have used the common syntactic notation that `[<foo>]` denotes at most one occurrence of a `<foo>`, and `<foo>+` denotes the juxtaposition of one or more `<foo>`'s.

A `<hierarchical-descriptor>` is a just a formula which defines the position of the object class within the hierarchy. A `<structural-descriptor>` gives the "data type" description of the object class. There are eight basic constructors which are available: `sequence-of:`, `set-of:`, `queue-of:`, `stack-of:`, `array:`, `record:`, `union:`, and `enumerated:`; each structural descriptor is defined in terms of one of these descriptors. Attributes declared in a `<structural-descriptor>` may be domestic or foreign. *Domestic* attributes are those present in the extant operating system UNIX; examples for a file include `owner` and `protection`. *Foreign* attributes are not directly applicable to UNIX, but are commonly used in other operating systems with which users may be familiar. File `visibility` and `on-line-expiration` are examples based upon TOPS-20. The presence of these foreign attributes provides a limited ability to formally express concepts not native to UNIX. Various properties of the systems TOPS-20, VAX/VMS, and MS-DOS[5] are modelled in this fashion.

The structural descriptor of an object class identifies the intrinsic properties of objects of that class. However, some properties do not fit well into such a role. For example, consider the property "has pageheaders" of a byte sequence. The direct characterization of this property, in terms of primitive properties of a byte sequence, is very complex, and dependent upon the exact meaning of "pageheader", which may vary from situation to situation. In Yucca-II, such properties are modelled using filters. A *filter*, which is defined by the syntactic category `<filter-descriptor>`, is an operator which is applied to an object to yield a new object of the same class. The action of the filter is not modelled by explicitly changing the values of some of the object's attributes; rather, it is recorded by stating that the filter has been applied. For example, in the query of the previous section, the filter *paginated* is applied, with the *option package* `%paginated-standard-pageheaders`, to version `0` of `#I` to yield version `1` of `#PQE`. Note that more than one filter may be applied, and the order in which they are applied is significant. For example, applying the `number-lines` filter (which produces an output in which each input line, blank or otherwise, is numbered) after the `pageheaders` filter will certainly produce a result distinct from application in the reverse order.

The `generation-structure:` attribute is relevant to macro expansions, and will be discussed shortly.

One of the most difficult knowledge representation problems encountered in our earlier system Yucca is the *display object problem*, which arises because it is necessary to distinguish an object from its display. As a specific example, consider the problem of responding to the query "How do I display the users currently logged on the system?" The display is not really a set of users, but rather a byte sequence describing the collection of users. This type of identification, which is made automically by humans, must be explicitly represented in a formal consultant. In Yucca-II, such displays are defined through the use of *macro meta-objects*. They are literally object macros; upon binding formal to actual parameters and expanding, an (ordinary) object is obtained. Rather than present the general syntax of macros, which is quite complex, we illustrate how an expansion of the macro `display-set-attributes` (Figure 4) yields an object of type `collective-user-listing` (Figure 5).

In Figure 4, the `external-parameters:` define the parameters which must be bound in the actual expansion. In this example, there is no constraint on the base type. The `source-parameters:` define the actual structure to be displayed, in this case, a set of the objects identified by the external parameter. The `generators:` entry specifies how the attributes of the source object induces attributes of the display object. The `special-attributes:`

[5]MS-DOS is a trademark of Microsoft Corporation.

```
(object-class-macro: display-set-attributes
   (hierarchy: (ISA: visible-byte-sequence (condition: declared)))
   (external-parameters: ((name: individual-source)
                          (local-identifier: ?IS)
                          (structure-class: any)))
   (source-parameters: ((name: display-source)
                        (structure-class: (set-of: (base: ?IS)))))
   (generators: (individual-attribute-inclusion:
                 (macro-expand: (display-record-attributes:
                                ((parameter: individual-source)
                                 (binding: ?IS))))))
   (special-attributes: ((name: sort-criterion)
                         (values: generic-sort-criteria))
                        ((name: sort-direction)
                         (values: direction))
                              ...         ) )
```

Figure 4: Example of a macro.

are additional attributes not defined in the generation.

A `collective-user-listing` is a specific macro expansion of the above, defined as in Figure 5 below.

```
(object-class: collective-user-listing
  (hierarchy: (ISA: (macro-expand: display-set-attributes
                                   (external-parameter-bindings:
                                     (individual-source: user)))
                                   (condition: declared)))
  (generation-structure: (special-attributes:
                           ((name: display-format)
                            (values: user-listing-display-format)))) )
```

Figure 5: A macro expansion of the example of Figure 4.

Note that the entire declaration is contained within the hierarchical description. The individual source variable `?IS` is bound to object class `user`, and an object which defines a display of a set of users is obtained. The `generation-structure:` entry identifies additional display attributes which are specific to this particular object, and are not generally relevant to the `display-set-attribute` macro. In this example, there is a special class `user-listing-display-format` which identifies particular formats (the UNIX *w* command format or the *f* command format) of the user listing, of which one must be selected.

4. The Database of Dynamic Objects

The overall design of the dynamic objects knowledge base is based upon the premise that most, if not all, interesting command language behaviors can be described as the interconnection of a set of primitive behaviors. Accordingly, there are two basic components; a collection of *simple dynamic object classes*, and an *interconnection calculus*. By interconnecting simple dynamic object classes, general dynamic object classes are obtained. At present, the only interconnection strategy supported is simple serial interconnection. Thus, every dynamic object class has a unique representation as a finite sequence $(D_1, D_2, .., D_k)$ of simple dynamic object classes. For example, a solution to the query posed in Section 2 has as one solution the object class `(pr-starter, pr-input-definition, pr-command-main, pr-finisher, pipe, lpr-starter, lpr-input-definition, lpr-command-name, lpr-option-laser, lpr-finisher)`.

All object classes have a common structure, which is represented by the template depicted in Figure 6 below.

```
(dynamic-object-class: <object-class-name>
   (hierarchy: <hierarchical-description>)
   (preconditions: <static-formula>)
   (postconditions: <static-formula>)
   (action: <static-formula>)
   (actor: <static-formula>) )
```

Figure 6: Structure of object classes.

A `<hierarchical-description>` is similar to that found in the static object database; it determines the position of the object class within the class hierarchy. A `<static-formula>` is a formula defined in the first-order language underlying the static-object database; the formalism involved is very similar to that described for queries in Section 2. Formally, the interconnection calculus allows the interconnection of object class `A` to object class `B` if and only if, after appropriate unification, the postcondition of `A` implies the precondition of `B`.

For each UNIX command, there is an associated *command manager*, whose purpose it is to collect, in a single module, all of the relevant information about the syntax and semantics of that particular command. Each such manager has three components, the object class definitions, the local interconnection syntax, and the global interconnection categorization. The *object classes definitions* for a given manager contain the formal descriptions for each simple dynamic object class associated with that command. There are three types of such object classes. First, for each command, there is the *main object class*, which defines the central behavior of the command. It is always included in any interconnection reprsenting the behavior of that command. For the *pr* command, this class is `pr-command-main`. Second, there are *option object classes* associated with each command. These classes express the behavior of command options; particular options are realized by including the associated class in an interconnection. For the *pr* command, the associated option object classes in the current version are `pr-drop-header-trailer`, `pr-change-pageheader`, `pr-change-pagelength`, and `pr-change-startpage`. Finally, for each command there are several *support object classes* which perform various housekeeping tasks, including initialization of defaults (which may subsequently be changed by option classes). For the *pr* command, there are three such classes, `pr-starter`, which must appear as the first class of any interconnection for *pr*, `pr-finisher`, which must appear as the last, and `pr-input-definition`, which controls whether the command version modelled uses standard input or user-supplied files for input. These modules are always mandatory in an interconnection for the associated command.

The *local interconnection syntax* for a given command provides two services. First, it supplies a template for how the simple object classes associated with the given command must be arranged in any interconnection, including the exact position for option object classes which are included. Thus, no searching or planning ever need be done to decide how to do local interconnection of object classes associated with a particular command; once the associated simple classes are selected, the interconnection is completely determined. Second, the local interconnection syntax provides the link between command-line syntax and class interconnection, including textual parameters, such as a user-supplied header for the *pr* command.

The *global interconnection categorization* provides nonlocal information on the nature of the input and output "ports" of the command, and provides assistance to the more global planning mechanism, which is further discussed in the next section.

In addition to a command manager for each modelled command, there are also several *auxiliary command managers.* In the current version, these include units for two kinds of interconnection, `pipe` and `sequence`, as well as two types of command line redirection, `redirect-input` and `redirect-output`.

5. Inference and Query Solution

There are two main modules which manage the query solution process, the *command synthesizer* and the *simulator*. The former addresses unknown action queries, while the latter addresses all other dynamic queries.

The problem of solving an unknown action query is basically one of plan generation. The preconditions and postconditions are known, as are the behaviors of the building blocks and the way in which they interconnect. The problem is to produce an appropriate plan (*qua* interconnection). Since solving such queries is at the very heart of Yucca-II, it is essential that they not only be solved correctly, but efficiently as well. General research on the subject of planning and the management of dynamic knowledge, such as may be found in the recent anthologies [Brow87] and [GeLa86], provides a wealth of conceptual information, but little on methods for constructing *efficient* general purpose planners. For this reason, we have developed a highly specialized architecture for the command synthesizer, which takes full advantage of the unique properties of operating system dynamics. This architecture is depicted in Figure 7.

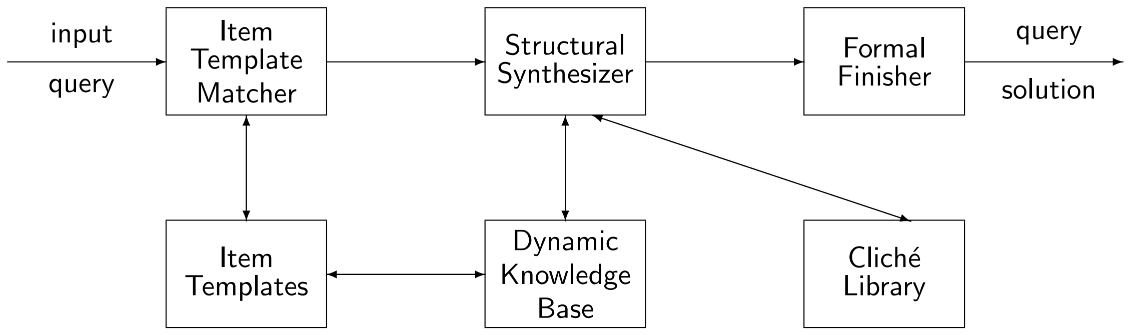

Figure 7: Architecture of the command synthesizer.

The first step in the solution of an unknown action query is the identification of a small set of candidate dynamic object classes which will participate in the interconnection which is to be the ultimate solution. This function is rendered by the *item template matcher*. The *item templates* form an inverted index into the dynamic object class database, indexing commands and command options by the static object class attributes (including filters) which they alter. For example, there is a reference from the `paginated` filter of the `visible-byte-sequence` object class (in the static knowledge base) through an item template to the `pr-command-main` object class of the *pr* command in the dynamic knowledge base. (The static knowledge base is ubiquitous within the command synthesizer, as it is throughout the entire back end, and is not explicitly depicted in Figure 7.) The item template matcher performs an initial examination of an unknown action query and, using the item templates as a guide, identifies a collection of simple dynamic object classes which may be useful in a solution to the query. For the example query given in Section 2, the simple command object classes `pr-command-main`, `lpr-command-main`, and `lpr-option-laser` would be selected.

The output of the item template matcher is then passed on to the *structural synthesizer*, which is the very heart of the planning unit. The processing of this unit is broken into two steps. In the first step, complete but isolated single command interconnections are assembled from the dynamic object classes delivered by the item template matcher. For the example query, two command interconnections would be generated at this step. The first is a "bare" *pr* command definition, consisting of the interconnection `(pr-starter, pr-input-definition, pr-command-main, pr-finisher)`, and the second is the interconnection defining printing on the laser printer using the *lpr* command, and consists of the interconnection `(lpr-starter, lpr-input-definition, lpr-command-name, lpr-option-laser, lpr-finisher)`. The information necessary to build these interconnections is contained entirely within the associated command managers; as noted in the previous section, no planning is necessary. In effect, at this first step, a set of potential building blocks which may be incorporated within the final solution is constructed.

In the second step, these building blocks are interconnected into what will become a solution to the query. The efficiency of this step rests primarily on the existence of a library of *clichés*, which are parameterized skeletons of commonly occurring command interconnections. The cliché `filter-input-file-and-print` is depicted in Figure 8.

```
(define-cliche:
   (name: filter-input-file-and-print)
   (nondefault-components:
     (name: initial-filter
      (categories: (AND: file-input stream-output)))
     (name: print-module
      (constraint: (AND: stream-input enqueue-output))))
   (interconnection-constraint:
     (pipe-couple:
           (arguments:(output-line: initial-filter)
                      (input-line: print-module)))) )
```

Figure 8: An example cliché.

Each such cliché has a number of slots for `nondefault-components` which must be filled by the command building blocks assembled in the first step. These blocks have *interconnection categories* which are identified in the command manager for the associated command, as noted in the previous section. In the cliché of Figure 8, the `initial-filter` component must be in both the category `file-input` (which says that it takes a single file as its primary input) as well as in the category `stream-output` (which states that it sends its result to the standard output). Similarly, the `print-module` component must be in the category `stream-input` (which mandates that it take its primary argument from standard input) as well as in the category `enqueue-output` (which states that it places its output in a queue). It is important to note that the valid interconnection categories for isolated commands are determined not only by the base command itself, but by the way in which arguments are supplied. For example, *lpr* is in the category `stream-input` only if the file to be printed is *not* supplied as a command-line argument, but rather is taken from standard input. The precise conditions are recorded in the command manager, and are checked after the isolated command is assembled.

Although more than one cliché may be selected at this step, only those which provide a match on all nondefault slots will be selected, and this will typically be a very small number. The structural synthesizer generates all match possibilities on each selected cliché. It is significant to note that the resulting interconnection will involve not only the slot fillers delivered in the matching process, but the cliché-supplied classes as well. In the example of Figure 8, the cliché interconnects its two arguments with a `pipe` dynamic object class. In the solution of the running example query, the cliché depicted in Figure 8 is the only one selected, and only one slot filling scheme is possible; namely, the `initial-filter` slot is bound to the assembled *pr* command, and the `print-module` slot is bound to the assembled *lpr* command.

The interconnections generated by the structural synthesizer are passed on to the final unit in the query solution process, the *formal finisher*. This role of this unit is to take the command interconnections supplied by the structural synthesizer and actually generate the bindings on the formal variables of the input query. The technique is basically one of pattern matching via unification. The format of the input query is matched to that of the solutions. A search for a unifier is made such that the preconditions of the query imply the preconditions of the solution, and the postconditions of the solution imply the postconditions of the query. (This may involve filling the secondary precondition and postcondition slots of the query with assertions required to make the match.) If more than one interconnection was passed along, the most suitable one must be selected. This is accomplished via a "best match" process, in which the solution with the weakest secondary preconditions and postconditions is selected. In the example query, only one unification is possible. There are no secondary postconditions, but the secondary precondition that the input file be readable is passed back with the solution. Finally, the binding of solution command syntax to the associated action variable is determined by referring back to the local interconnection syntax components of the appropriate command managers.

It should be emphasized that the query solving mechanism is in reality a very specialized planning mechanism. It is *hierarchical* in the sense of [Sace77], in that the initial "plan" conceived by the item template matcher contains only a fragments of what will eventually become the total plan. The structural synthesizer then adds further refinement, and finally the formal synthesizer produces the correct plan. It is also at least implicitly *nonlinear* in that in the initial phase (the item template matcher) commands and options are selected without regard for the order in which they are to be interconnected. That refinement takes place in the next step (the structural synthesizer).

Our use of a library of clichés is inspired by a similar technique employed in the Programmer's Apprentice ([Rich81], [Wate85]). Although that system makes ultimate use of such clichés in a somewhat different manner than do we, the notion that *parameterized versions* of commonly used plans should be stored away in a library and retrieved when needed is a crucial one, which contributes substantially to the efficiency of dynamic query solution in Yucca-II.

Unknown action queries are given primary support by the back end, in the sense that the the knowledge representation and inference techniques are primarily designed for such queries. Unknown precondition, postcondition, and actor queries are supported to the extent that the propositional semantic representation recaptures the needed behavior. The simulator module supports the solution of such queries by directly simulating the behavior of the command sequence in question. (For lack of space, we eschew further description of the simulator.) We hope to eventually provide strong support for such queries also, but for now, the emphasis is on unknown action queries.

6. Conclusions and Other Directions

To provide sophisticated on-line help for an operating system such as UNIX, an equally sophisticated knowledge representation and inference system is necessary. A framework for representing such knowledge has been presented, using a propositional representation for command semantics and a highly customized plan generation mechanism for the solution of dynamic queries. Compared to similar systems which embed the modelling and planning mechanism into a general knowledge representation and inference framework, we believe that our system will offer the advantage of the ability to answer *detailed* queries correctly and more rapidly, with a modest commitment of system resources.

We further conjecture that the knowledge representation techniques employed in Yucca-II may be used not only for query solution support, but for the specification and documentation of operating system commands in the first place. Indeed, one conclusion that has repeatedly been reached in this investigation is that if the command semantics had been formally specified in the first place, not only would the definition and implementation of the consultation system have been much easier, but the commands themselves would have been much more understandable. Accordingly, we are beginning a parallel investigation aimed at the formal specification, implementation, and documentation of UNIX commands.

References

[BaMS86] Bates, M., M. G. Moser, and D. Stallard, “The IRUS transportable natural language database interface,” *Expert Database System, Proceedings of the First International Workshop*, Benjamin/Cummings, edited by L. Kerschberg, 1986, pp. 617-630.

[BiCa85] Billmers, M. A., and M. G. Carifio, “Building knowledge-based operating system consultants,” *Proc. Second Conf. on Artificial Intelligence Applications*, Miami Beach, 1985, pp. 459-454.

[BrSc85] Brachman, R. J., and J. G. Schmolze, “An overview of the KL-ONE knowledge representation system,” *Cognitive Science*, **9**(1985), pp. 171-216.

[Brow87] Brown, F. M. (editor), *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop*, Morgan Kaufmann, 1987.

[Codd72] Codd, E. F., “Relational completeness of data base sublanguages,” in *Data Base Systems*, edited by R. Rustin, pp. 65-98.

[DoHe82] Douglass, R. J., and S. J. Hegner, “An expert consultant for the UNIX operating system: bridging the gap between the user and command language semantics,” *Proc. Fourth CSCSI/SCEIO Conf.*, Saskatoon, 1982, pp. 119-127.

[GeLa86] Georgeff, M. P., and A. L. Lansky (editors), *Reasoning about Actions and Plans, Proceedings of the 1986 Workshop*, Morgan Kaufmann, 1986.

[Gros83] Grosz, B. J., “TEAM: a transportable natural language interface system,” *Proc. 1983 Conf. on Applied Natural Language Processing*, pp. 39-45.

[Haye82a] Hayes, P. J., “Uniform help facilities for a cooperative user interface,” *Proc. 1982 NCC*, Houston.

[Haye82b] Hayes, P. J., “Cooperative command interaction through the Cousin system,” *Proc. Intl. Conf. Man/Machine Systems*, U. of Manchester Institute of Science and Technology, 1982.

[HeDo84] Hegner, S. J., and R. J. Douglass, “Knowledge base design for an operating system expert consultant,” *Proc. Fifth CSCSI/SCEIO Conf.*, London, 1984, pp. 159-161.

[McKe86] Mc Kevitt, P., Building embedded representations of queries about UNIX, Technical Report, Computing Research Laboratory, New Mexico State University, December 1986.

[McWi87] Mc Kevitt, P., and Y. Wilks, "Transfer semantics in an operating system consultant: the formalization of actions involving object transfer," *Proc. Tenth IJCAI*, Milan, 1987, pp. 569-575.

[Rich81] Rich, C., "A formal representation for plans in the Programmer's Apprentice," *Proc. Seventh IJCAI*, Vancouver, 1981, pp. 1044-1052.

[Sace77] Sacerdoti, E. D., *A Structure for Plans and Behavior*, Elsevier North-Holland, 1977.

[Scra75] Scragg, G. W., "Answering process questions," *Proc. Fourth IJCAI*, Tbilisi, 1975, pp. 435-442.

[Walt78] Waltz, D. L., "An English language question answering system for a large relational database," *Comm. ACM*, **21**(1978), pp. 526-539.

[Wate85] Waters, R. C., "The Programmer's Apprentice: a session with KBEmacs," *IEEE Trans. Software Engineering*, **11**(1985), pp. 1296-1320.

[WiAC84] Wilensky, R., Y. Arens, and D. Chin, "Talking to UNIX in English: an overview of UC," *Comm. ACM*, **27**(1984), pp. 574-593.

[WMAC86] Wilensky, R., J. Mayfield, A. Albert, D. Chin, C. Cox, M. Luria, J. Martin, and D. Wu, UC – A progress report, Report No. UCB/CSD 87/303, Computer Science Division of EECS, University of California at Berkeley, July, 1986.

[YuLo84] Yun, D. Y. Y., and D. Loeb, "The CMS-HELP expert system," *Proc. Intl. Conf. on Data Engineering*, Los Angeles, 1984, pp. 459-466.