Artificial Intelligence Review **14:** 181–228, 2000.
© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

Plan Realization for Complex Command Interaction in the UNIX Help Domain

STEPHEN J. HEGNER
Umeå University, Department of Computing Science, S-901 87 Umeå, Sweden
(E-mail: hegner@cs.umu.se)

Abstract. Yucca-* is a consultation system which is designed to provide the UNIX user, through a friendly interface, with detailed expert advice on the use of the UNIX command language. One of the principal design goals of this system is the ability to provide correct responses to technically complex queries whose solution may involve the interconnection of several commands, each with multiple options. The realization of such a goal requires two things. First, representation of dynamic knowledge about command behavior at a sufficient level of detail to support solution of the query is needed. Second, a planning mechanism capable of interconnecting such knowledge into a cohesive solution must be provided. This paper first develops the command dynamics representation techniques employed in Yucca-*. It then examines in detail the plan generation mechanism which is used to solve complex dynamic queries. Particular emphasis is placed upon those aspects of the problem which are unique to this particular domain.

Keywords: consultation system, operating system, planning

1. Introduction

1.1. *Overall goals*

Like the other systems described in this collection, Yucca-* is an intelligent help system for the UNIX[1] operating system. However, not all operating system consultants have identical goals; there are many points of emphasis upon which effort may be focused. Therefore, it appropriate to begin our description of Yucca-* with a statement of just *what* it is intended to do, before paying any attention to *how*.

The user community
The targeted users of Yucca-* are those persons who possess some familiarity with computer systems, but who lack the detailed knowledge necessary to accomplish a desired task. There are at least two common (overlapping) categories of such users.

1. The first category consists of users who are migrating from another system. That system may be another shared system, such as TOPS-20[2]

or VM/CMS,[3] or it may be an operating system for dedicated microcomputers, such as MS-DOS.[4] Such users will be aware, for example, that most operating systems have a command to print the contents of a file, but may be unaware that `lpr` is the specific command for accomplishing that task in UNIX, or that a combination of the `lpr` command and the `cat` command is necessary to print a file with lines numbered.

2. The second category contains occasional but not naïve users. These persons may use UNIX from time to time in the course of their work, but are not intense users, and thus find it easy to forget the details and intricacies of command syntax and semantics.

Key characteristic of this user community

1. *Type of information sought.* Perhaps the most succinct characterization of the members of this user community is that they possess the *conceptual* knowledge necessary to identify the information which they want, but lack the *implementation* knowledge of those concepts within the extant domain. They do not need to be told that information is stored in files, or that text editors are used to change the contents of files, but they may not understand the subtle but important differences between UNIX files and TOPS-20 files, nor may they know which editors are available under UNIX, and what their relative features are.
2. *Complexity of knowledge.* Because the targeted class of users is more sophisticated than true computing novices, so too will be their queries. Rather than simply wanting to know how to print a file, such a user may want to know how to print a file on the laser printer with pagination and with page headers.

Form of the utility

Yucca-* is designed to be an external utility, rather than an integral system component such as are Cousin (Haye 1982a), (Haye 1982b), (Hayes and Szekely 1983), SARA (Fenschel and Estrin 1982), and CONSUL (Mark et al. 1980). This means that it must run as a separate program on the existing system, to be invoked only when needed. Its installation will not alter the standard user interface in any way.

Historical influence on design goals

Yucca-* is the outgrowth of three earlier efforts, *UCC* (Douglass and Hegner 1982), *Yucca* (Hegner and Douglass 1984), each developed in Franz Lisp, and *Yucca-II* (Hegner 1988), a proposal which never reached fruition. Each of these earlier systems featured a natural-language interface to the user. It was our feeling in designing those systems that the user interface must mimic the human consultant insofar as possible, particularly requiring little or no learning on the part of the user, and that natural language was the best way to realize this end. While this may still represent the ideal, experience has lead us to the more pragmatic conclusion that a pure natural-language interface may not be optimal. The number of queries which could not be understood, or which were understood incorrectly, was too high to be acceptable in a working system. Due to the highly structured nature of knowledge about operating system behavior, it seems that, at least in some cases, alternate approaches to user interfaces, such as intelligent menus, may form a viable and even superior alternative, not only in functionality, but in terms of reduced design and implementation effort as well. It is clear that further work on the form of the user interface needs to be done.

Regardless of the form of the user interface, once the general class of information which the system must provide has been identified, the knowledge *about UNIX* which is necessary may be firmly identified. Yucca-* is in fact a design of a knowledge base about UNIX, to which a user interface must be attached. In this paper, we focus on design of that knowledge base, emphasizing in particular planning issues.

1.2. *System overview*

Unlike most domains of knowledge to which artificial intelligence is applied, the behavior of an operating system command language is formally specified and completely understood. In the terminology of Hart (Hart 1982), the source code of the UNIX operating system, together with an understanding of its semantics, forms a *deep* model of that system. Our main thesis, and the cornerstone of the design of Yucca-*, is that this formality and depth of available domain knowledge should be retained as a distinguishing feature of the knowledge representation scheme within an operating system consultation system, rather than discarded in favor of more general but less formal techniques. Nonetheless, while observing that knowledge about the behavior of UNIX lends itself well to formality, we readily acknowledge that understanding a user's query is not a completely formal process; the inherent imprecision of the man-machine interface must be addressed in any design. To achieve this end, Yucca-* is designed to be consistent with an underlying philosophy that it is possible and indeed desirable to separate *understanding* of a user's query from *solving* it. As a manifestation of this philosophy, the overall architecture of Yucca-* is sharply divided into two components, identified in Figure 1.

The understanding phase

The function of the *understander* is to interface to the user and generate a precise formulation of his questions into formal queries in a specially

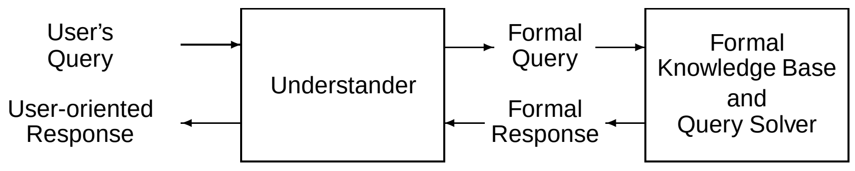

Figure 1. Overall architecture of Yucca-*.

designed formal query language termed *OSquel*, and to translate solutions of such queries into user-understandable responses. As noted previously, the exact nature of this module is not specified in the top-level design of Yucca-*, but is open to a variety of alternatives, including natural language. However, in any realization, this part of the system contains knowledge relevant to the *understanding* of queries, but not for solution. For example, posed a query asking how to print a file with pageheaders on the laser printer, the understander knows about printing, laser printers, and pageheaders. However, it does not know that the UNIX command `pr` is the one to use to put pageheaders on a file, that the `lpr` command, with the appropriate option, is the means to enqueue a file for printing on the laser printer, nor that the two should be interconnected with a UNIX pipe to accomplish the task. In short, it knows how to communicate with the user regarding conceptualizations of operating system concepts and express them as formal conceptualizations in OSquel, but it knows nothing of their implementation.

The solving phase

The function of the *formal knowledge base and query solver* is to find solutions to the formal queries posed by the understander. It does know about the semantics of `pr` and `lpr`, as well as how they can be interconnected via a pipe. On the other hand, it has no special knowledge about communicating with the user, such as user misconceptions and beliefs, or defaults implied by natural language. In short, it knows nothing of formulating conceptualization of command language, but everything about translating formal conceptualizations into implementations. Essentially, this module is a formal database of knowledge about the UNIX command language; everything about knowledge representation and inference in this component is completely formal. Thus, it is in this component that we exploit completely the formal aspects of domain knowledge about UNIX.

Coupling of the two components
Despite this separation of understanding and solving into two separate components, it is clear that the two modules must share enough information so that each is capable of understanding and expressing knowledge in the formal query language OSquel. To this end, the *formal static knowledge base* is accessible by both components. This knowledge base contains information about objects and their attributes in the extant domain, but not information about how commands alter their attributes. This information is sufficient to formulate and understand queries, but not to solve them.

Major thesis
It is a major thesis of this work that a division of the consultation system into two components, as described above, is not only possible but practical and advantageous. Specifically, we propose that detailed procedural domain knowledge can be effectively separated from that knowledge necessary to interface with and understand the user. In particular, we propose that a single "back-end" knowledge base about UNIX can be be used to support a variety of user interfaces of varying form and sophistication. We note that this philosophy is completely opposite to that employed in the UC natural-language consultation system (Wilensky et al. 1988), in which linguistic knowledge and domain knowledge about UNIX are contained in a common knowledge base.

Comparison to friendly interfaces to database systems
It is difficult to avoid noticing the similarity between the architecture of Yucca-* and that of natural-language front ends to database systems, such as PLANES (Waltz 1978), TEAM (Grosz 1983), and IRUS (Bates et al. 1986), and other "user-friendly" interfaces, such as QBE (Zloof 1977), and BAROQUE (Motro 1986). However, these systems all deal with the problem of attaching a front end to an existing database system, while Yucca-* involves the design and implementation of both components of the system from scratch. In addition, traditional database systems deal only with static knowledge, and so queries involving dynamics need not be supported. However, it is the conceptualization of *actions* which is the central component of the understander. We do note, however, that Salveter (Salveter 1984) has examined the problem of developing a natural-language front end to database systems which deal with updates as well as queries, and so must support concepts closely related to dynamic queries.

1.3. *Overview of plan realization issues*

Forms of assistance

It is important to distinguish our use of plans and planning from that of other help facilities. There are at least two distinct ways in which assistance may rendered by an operating system help facility. Yucca-* is a *consultation* system, meaning that it provides immediate real-time help to users regarding specific queries. Genereseth's *MACSYMA Advisor* (Genesereth 1979) is perhaps the prototypical example of such a utility. In this regard Yucca-* is similar to the *UC* UNIX consultation system (Wilensky et al. 1984), (Wilensky et al. 1988). Because such facilities typically provide help only when it is explicitly requested, they are sometimes termed *passive* assistants. Plan *realization* is the major planning-related activity of such systems, since the system takes a specification of a desired result and attempts to assemble a plan (*qua* command sequence) which realizes that result. This is in marked contrast to *critiquing systems*, which take a user's attempt at a problem and attempt to suggest corrections and/or improvements. This approach has been strongly promoted in the domain of providing expert medical advice (Miller 1984). However it is also central to several operating system help systems, including *Wizard* (Finin 1983) and *AQUA* (Quilici et al. 1986). In such systems, a user's attempt at solving a simple problem (such as deleting a group of files) is observed, and a "better" approach is suggested by the system. Plan *recognition* and *correction* is the key plan-related activity in this sort of system. Since such consultants often operate as observers of the user's actual attempts at problem solution, they are often termed *active* assistants. The current design of Yucca-* does not provide such help. We note that some operating system help facilities, such as *SC* (Hecking et al. 1988) and *USCSH* (Neelkandan et al. 1987), incorporate both active and passive assistants.

Types of queries

There are two major flavors of queries which users will pose, and which a consultation system should be prepared to service.

1. *Process queries*. The terminology *process query* is borrowed from (Scragg 1975), and refers to queries which deal with the dynamics of UNIX. The most important class of such queries is characterized by the required command being unknown, such as asking *how* to print a file with pageheaders on the laser printer, although their are other possibilities, such as asking *who* can read a certain class of files, in which the actor is unknown.
2. *Conceptual queries*. Occasionally, users will need clarification of the static structure of the underlying system, as characterized by the queries

asking for the definition of a UNIX pipe, or for information on how directories are organized in UNIX.

Planning and unknown action queries

Although Yucca-* has the capability of answering a wide variety of queries, its strength lies in its ability to answer detailed process queries in which the action is unknown. Consider, as a specific example, the query asking how to print a file with pageheaders and lines numbered on the laser printer. Even once this query is understood, its solution is quite nontrivial. Indeed, the natural solution, which is represented by the command `cat -n` *filename* `| pr | lpr -Plaser`, requires the interconnection of three basic commands, two of which contain explicitly identified options. While it would perhaps be possible to retain explicitly the solution for many such queries, such a knowledge base would at best be inflexible and nonrobust. Rather, Yucca-* is designed to assemble such an interconnection from primitive components, utilizing knowledge bases of basic command and interconnector descriptions. The assembly of such an interconnection is clearly a planning process. The query statement represents the goal, and the interconnection the plan. The overall approach of how such planning is performed, as well as how it is supported with appropriate knowledge bases, is the central topic of this article.

Scope of the plan domain

Even within the scope of passive consultation systems, there are different directions which may be taken. The emphasis in Yucca-* is on the solution of relatively complex domain-specific queries which involve the interconnection of several commands and options. Thus, our primary concern is how to view commands, command options, and command interconnectors as basic building blocks for complex command behavior, and how to realize an interconnection of such blocks affording a particular behavior. This involves *domain specific* planning. In the UC system, on the other hand, a major goal is to utilize the UNIX domain as a testbed for the evaluation of general frameworks for knowledge representation and planning. As such, UC emphasizes *general* planning to a much greater degree. As a result, Yucca-* is capable of answering much more technically complex queries. See (Chin 1988) for a detailed discussion of the rôle of planning in UC. The SC system (Hecking et al. 1988), like Yucca-*, employs domain-specific knowledge representation, but its emphasis is not on plan generation, but rather upon direct representation of command behavior. Yucca-* appears to be unique amongst UNIX consultation systems in its emphasis upon the solution of complex technical queries via plan generation.

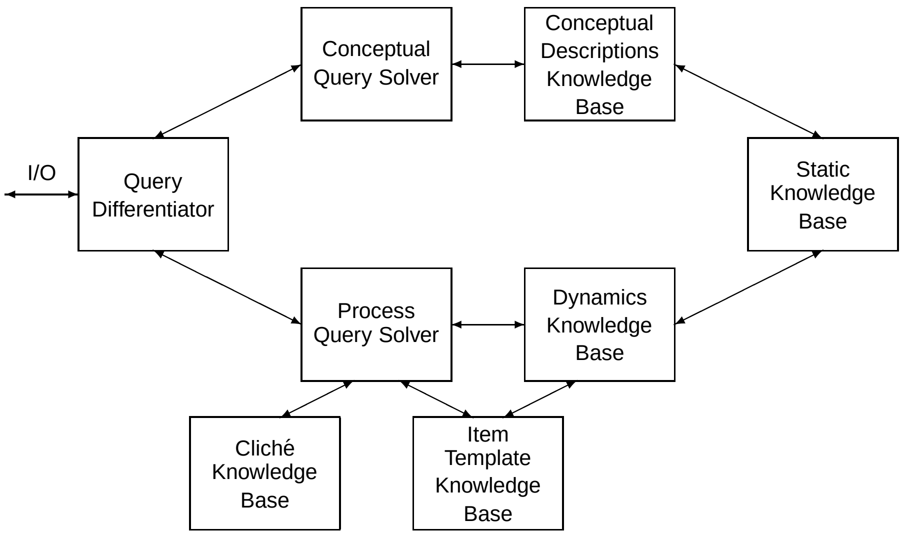

Figure 2. Overall architecture of back end.

2. Overview of the back end

In this section, we overview the architecture of the back end, and identify the places in this paper in which the individual components are elaborated. The overall architecture of the back end is depicted in Figure 2. Input queries are classified as either *conceptual* or *process*; the difference is easily determined by syntactic means. The sole rôle of the *query differentiator* is to decide of which type a query is, and to route it to the appropriate solver. This is a trivial task which is not further elaborated in this paper. Each query solver has access to particular knowledge bases tailored to its needs.

2.1. *Overview of query solution modules*

The conceptual query solver
The task of the conceptual query solver is to solve conceptual queries. It is not discussed further in this paper.

The process query solver
The process query solver actually consists of two distinct units. The *command synthesizer* is a complex goal identification and planning unit which is used to solve queries in which the action to be performed is the primary unknown. The ultimate goal of this paper is to describe the operation of this query solution unit, as it is "the" planner of the back end. This unit is described in Section 6. The *simulator* is used to solve all other dynamics queries; for reasons of space limitations, it is not discussed further in this paper.

2.2. *Overview of the knowledge bases*

Although this paper is ultimately about plan realization, such a topic cannot be considered in isolation. Rather, successful planning must be supported by appropriate knowledge representation and inference mechanisms. Therefore, we also provide a description of those aspects of knowledge representation which are relevant to plan generation.

The static knowledge base
The *static knowledge base* is central to the back end; all other knowledge bases depend critically upon concepts formulated in this unit. It consists of formal descriptions of the objects classes present in an operating system enivronment, such as file, directory, user, file protection, byte sequence, print queue, and the like. These object classes are organized in both generalization (IS-A) and aggregation (PART-OF) hierarchies. The static knowledge base contains definitions of well over one-hundred classes, and is described in Section 4.

The conceptual descriptions knowledge base
In Yucca-*, conceptual queries asking for a general description of the concept of *pipe* or of directory structure of UNIX are solved by providing the user with a short textual response explaining the concept. The *conceptual descriptions knowledge base* contains the text of those responses. Since this knowledge base is not used in the support of plan realization, it is not discussed further in this paper.

The dynamics knowledge base
This knowledge base contains formal descriptions of the syntax and semantics of several dozen UNIX commands, as well as definitions of supporting operations such as piping, command sequencing, and redirection. It is discussed in detail in Section 5. The complete representation of the `pr` command is contained in the Appendix of this paper.

```
(query:
  (dynamics: (((state: P) (action: F) (state: Q)) (actor: A)))
  (query-variables: <v-list>) ;; Variables to be bound in solution
  (external-variables: <v-list>) ;; Input/output parameters
  (local-variables: <v-list>) ;; Output only parameters
  (define P <static-formula-with-variables>) ;; Precondition
  (define Q <static-formula-with-variables>) ;; Postcondition
  (define F <static-formula-with-variables>) ;; Action
  (define A <static-formula-with-variables>) ;; Actor
)
```

Figure 3. Syntax of a process query.

The cliché and item template knowledge bases
These knowledge bases do not contain any "new" information about the structure or behavior of UNIX. Rather, they play a crucial supporting rôle in the *efficient* realization of plans. They are described in Section 6.

3. Query formulation

All formal queries are expressed in a formal query language termed *OSquel*. Although the details of query representation cannot be fully understood until the features of the static knowledge base have been elaborated, it is nonetheless useful to provide an overview of query formulation at this point, since the design of the static knowledge base, as well as the other knowledge bases and process query solvers, was strongly motivated by the needs inherent in query formulation as well as solution.

3.1. *Process queries*

Process queries are at the very heart of Yucca-*. The general syntactic format of such a query is shown in Figure 3.[5]

The underlying idea is that we express dynamics *axiomatically*, with formulas identifying the *precondition* necessary for the action to take place, the *postcondition* asserted as a result of the action, the *actor* performing the action, as well as the *action* itself. Each of these four components is represented as a formula in *SL*, which is a first-order language underlying the static knowledge base. The unknowns, or variables to be bound, are identified in the `query-variables:` entry. Queries are classified by identifying the location of the major unknown; there are four such classifications.

Unknown action queries
In unknown action queries, the primary unknown is the action `F`. There are three levels of such queries. In the most general case, the unknown action is a general command interconnection. An example would be the formalization of the request to explain how to print a file with pageheaders on the laser printer, which has as a solution `pr <filename> | lpr -Plaser`. Unknown action queries may also designate that the answer be a single command, such as the query asking which single command puts pageheaders on a file or they may designate that the answer be an option of an existing command, such as the query asking which option on the pr command allows specification of the pageheader value. To which of these three classifications a given formal query belongs is determined by the type of the unknown variable in the action slot of the query.

Unknown consequence queries
In unknown consequence queries, the major unknown is the result of performing an action, which is the postcondition `Q` of the formal query. Examples include formalizations of asking what happens if one logs out with jobs running in the background or asking what happens if one tries to delete a directory with files in it.

Unknown precondition queries
These queries ask what is necessary to achieve a given task, and so the primary unknown is the precondition `P` of the formal query. Examples include formalizations of asking what privileges are necessary to set the system date or asking what properties a directory needs to have so that one may delete it.

Unknown actor queries
Occasionally, queries arise in which the actor `A` is the primary unknown. An example is the query which asks who can read another user's files.

It is clear from data gathered on queries submitted to the prototype system UCC that unknown action queries are the dominant form of process query posed by users. Therefore, such queries are given primary emphasis in the design of the back end. This means that the selection of features for the back end is driven by the need to solve such queries as completely and efficiently as possible. The solution of unknown consequence, precondition, and actor queries is given only secondary consideration in Yucca-*, and is not discussed further in this report.

Secondary conditions
In classifying queries, we have alluded to the "primary unknown". However, the proper formalization of queries may require more than one

unknown. For example, in responding to the query asking how to print a file with pageheaders on the laser printer, while the command sequence `pr <filename> | lpr -Plaser` may be the "primary" response, the success of this solution is conditional upon the file identified by `<filename>` being readable by the user. Such a condition would be bound to a variable known as a *secondary precondition* of the query. Secondary postconditions are also necessary in some instances.

3.2. *Example of a process query*

Let us now take a look at the formalization of a specific unknown action query. The central notion is that of a *mutable object*; that is, an object whose characteristics may be changed by the execution of some action. In formalizing mutation, all objects are assigned a *version number*. Two instances of the same object, but with different version numbers, may differ from one another in value. As the specific example, let us formalize the query asking how to print a file with pageheaders and lines numbered on the laser printer. An informal tabulation of how this query is represented in OSquel is presented in Figure 4.

Several items are worthy of note. First of all, the version number of the input file `#I` does not change; this means that the solution is not permitted to change that file in any way. Second, `%laser-print-queue` (which identifies the laser printer queue) does change in version due to the action; this means that we expect its value to change. Third, the print queue entry `#PQE` is a mutation of the contents of `#I`, with *first* lines numbered and *then* pageheaders inserted. The decision that pagination should occur after numbering lines is one that is made by the front end in the process of query understanding, based upon what is a reasonable default in the absence of further information. Fourth, the action may be any command *sequence*, and is not restricted to a single command. Again, this decision is made by the front end, and is the default unless the user explicitly indicates that he is looking for a single command or option. Similarly, the actor defaults to the user asking the query; this selection is also made by default in the front end. Finally, both the preconditions and postconditions have secondary slots for "overflow"; we shall see how these are used when query solution is examined in detail in Section 6.

The actual formalization of this query is shown in Figure 5. Of course, we have not yet provided the necessary detail of information on the static knowledge base to fully explain the syntax and meaning. It is presented at this point to give the reader an idea of what an actual formal query looks like, as well as to provide a reference example which will be carried throughout this paper.

Preconditions:
- Version 0 of the *mutable object* file `#I` is owned by the current user.
- The contents of this file is a visible byte sequence.
- The mutable object `%laser-printer-queue` currently exists in version 0.
- The variable `?SPP` is to be bound to any *secondary precondition*.

Action:
- The variable `?CS` is to be bound to the command sequence answering the query.

Postconditions:
- The file `#I` does not change; it stays at version 0.
- `%laser-print-queue` is *mutated* to version 1.
- Version 1 of the mutable object `#PQE` identifies a new print queue entry which is owned by the current user and which is appended to version 0 of the queue.
- The contents of version 1 of `#PQE` is the same as that of version 0 of `#I`, save that it first has lines numbered, and then is paginated with standard pageheaders inserted.
- The variable `?SPQ` is to be bound to any *secondary postcondition*.

Actor:
- The actor is the current user.

Figure 4. Sketch of formalization of the query asking how to print a file with pageheaders and lines numbered on the laser printer.

4. The static knowledge base

4.1. *Overview and syntactic description*

The static knowledge base is the most ubiquitous entity in the entire Yucca-* system. Not only is every other component of the back end dependent upon it, but the definition of the query language OSquel is as well. Thus, this knowledge base is known even to the natural language front end.

The major purpose of the static knowledge base is to provide descriptions of those classes of objects which are either manipulated by commands or else function as actors in the execution of commands. There are currently definitions in the Yucca-* static knowledge base for over one-hundred *object classes*, broadly partitioned into groups, as indicated below.

1. *The master object classes*. These object classes contain descriptions of the top-level components of the entire UNIX system. They include definitions for `master-system`, `user-table`, `login-table`, `process-table`,

```
(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) ;; #I is a file.
              (identification: ((name: #I) (version: 0))) )
   (instance: (class: visible-byte-sequence)
              (identification:    ;; of visible characters.
                (retrieve: record-entry:
                              (field: contents)
                              (source: ((name: #I) (version: 0))))))
   (instance: (class: print-queue) ;; Laser print queue state 0.
              (identification: ((name: %laser-print-queue) (version: 0))))
   (secondary: ?SPP))) ;; Slot for additional constraints.
(define: Q
 (AND:
   (instance: (class: plain-file) ;; #I doesn't change.
              (identification: ((name: #I) (version: 0))))
   (instance: (class: print-queue) ;; Print queue does change.
              (identification: ((name: %laser-print-queue) (version: 1))))
   (instance: (class: print-queue-entry) ;; Print queue entry version 1
              (identification: ((name: #PQE) (version: 1))))
   (= (retrieve: entire-value: ;; #PQE is added to print queue.
                 (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: lines-numbered) ;; First number lines
                     (constraint: (value: all))
                     (argument:
                      (apply-filter: ;; Then paginate copy of #I
                      (name: paginated) ;; to get #PQE.
                      (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: ;; #PQE is owned by our user.
                (field: owner)
                (identification: ((name: #PQE) (version: 1))))
      (retrieve: entire-value:
                (source: %user)))
   (secondary: ?SPQ))) ;; Slot for additional consequences.
(define: F  ;; An arbitrary command interconnection.
   (instance: (class: command-sequence) (identification: ?CS)))
(define: A ;; The current user.
   (instance: (class: user) (identification: %user))) )
```

Figure 5. Formalization in OSquel of the query asking how to print a file with pageheaders and lines numbered on the laser printer.

and `directory-structure`, as well as for supporting concepts such as `directory-tree`.

2. *The file object classes.* These object classes provide the structural definitions for all types of files, including directories and devices. It also includes definitions for supporting concepts such as `file-node`, `i-node-number`, and various concepts related to file access, including paths and protection.

3. *The byte sequence objects*. These object classes provide definitions for special byte sequences, such as `visible-byte-sequence` and `formatter-byte-sequence`, as well as listings which represent displays of other collections, such as `collective-user-listing` and `collective-process-listing`.
4. *The user and connection object classes*. These object classes provides descriptions of system users, terminal connections by these users, and the special properties of command processors, but not processes themselves. Represented in particular are the details and parameters involved in a terminal connection, and the attributes seen and defined in a session with the system.
5. *The process object classes*. The central object class in this category is `process`, which models the notion of a UNIX process.
6. *The printer object classes*. This category contains those objects relevant to using system printers; in particular, notions related to print queues and their entries are included.
7. *The command object classes*. Although the semantics of commands are modelled in the dynamics knowledge base, there are object classes in the static knowledge base which identify commands by name. These are used principally in support of conceptual queries.
8. *The fundamental object classes*. The fundamental object classes deal with primitive notions such as time and date, bytes, characters, natural numbers and the like. They seldom are used on their own, but are ubiquitous in the definitions of more complex objects.
9. *The macro meta-object classes*. The macro meta-object classes are not really object classes; rather, they are definitions of macros which may be used to construct new classes. Their use will be discussed in 4.5 below.

The classification of objects into these groups, while conceptually useful, is quite informal. On the other hand, the object classes are formally classified in a generalization hierarchy, in much the spirit of generalization systems such as KL-ONE (Brachman and Schmolze 1985). This permits us to allow an object class to inherit all of the attributes of its parent. For example, the class `directory-file` inherits all of the attributes of the class `file`, since it is formally declared that a directory is a file.

With the exception of the macro meta-object classes, all object class definitions share a common syntax, which is given in Figure 6.

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.

The `hierarchical-descriptor` is a formula which classifies the object class in the hierarchy, while the following three entries all give information

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

Figure 6. Syntax of object class definition.

about the attributes of the object class. The `structure:` entry defines the "traditional" attributes of the class, and is elaborated in 4.3. The `filters:` and `generation-structure:` entries are used to recapture special types of attributes which occur in the modelling of command language behavior, and are described in 4.4 and 4.5, respectively.

4.2. *Object instances*

Object class declarations define the structure which actual objects must take. In Yucca-*, there are two distinct flavors of object instances.

Mutable objects
Operating system commands manipulate *instances* of object classes, which we term *mutable objects*. For example, In the query illustrated in Figure 5, `#I`, `#PQE`, and `%laser-print-queue`, are all names identifying mutable objects. By convention, names beginning with "`#`" denote object instances which are defined for local use, while names beginning with "`%`" denote canonical predesignated objects which are known in any use of the consultant.

Constant objects
It is often useful to have as a standard reference a "default" value for an object class, which is known by name to all parties using the static knowledge base. The object instance identified by `%paginated-standard-pageheaders` employed in the example query of Figure 5 is one such instance; it identifies an object instance of type `pagination` which defines visible byte sequence pagination in a canonical default way.

Although many of the details of object representation depend upon corresponding details of object class definition, and so are described in subsequent parts of this section, it is nonetheless important that we provide an overall idea of object representation at this point. The syntax of declaration of an object instance is shown in Figure 7.

Several examples may be found in the query depicted in Figure 5. The `<version-identifier>` may be either a natural number or the constant identifier `fixed:`. When it is the latter, a constant object instance, not subject to mutation, is declared. When it is the former, the version number identifies

```
(instance: (class: <object-class-name>)
           (identification: ((name: <object-instance-name>)
                             (version: <version-identifier>)))
           [<value-descriptor>] )
```

Figure 7. Syntax of object instance definition.

a particular *version* of the object instance, of which there may be several. Two object instances with the same name but different version numbers may have different values, but identity of both name and version number mandates identical values. Thus, in any statement or query, version `0` of `#I` *always* denotes the same object instance with exactly the same value, but version `1` of that same object instance may differ in value from version `0`.

The `<value-descriptor>` is used to declare explicitly the value of the particular version of the instance; examples will appear later. We now turn to elaboration upon the structure of object classes.

4.3. *Simple structure*

Basic notions

The `structure:` field of an object class definition is used to give a standard programming-language style data type to that object class. Currently, there are eight fundamental constructors which are used in this capacity; they are `enumerated:`, `record:`, `array:`, `union:`, `set-of:`, `sequence-of:`, `queue-of:`, and `stack-of:`. In addition, there is the `external:` declaration which is used to identify structures defined externally to Yucca-*, such as the integers.

The definitions of three classes associated with print queues are given in Figure 8.

Associated with the fundamental constructors are standard operators. The `retrieve:` operator is used to extract values from specific instances on a variety of types. There are also type-specific operators, such as `enqueue:` and `remove:` for type `queue-of:`. These operations may be used, along with the equality predicate, to specify rather complex constraints on individual states. See Figure 5 for examples.

Foreign attributes

One of the more important problems in designing an overall consultation system is user modelling. Even though we expect the user to have a reasonable conceptualization of how an operating system is organized and how it operates, the user may have misconceptions as well. To the extent that such misconceptions occur because the user is migrating from another operating

```
(object-class: print-queue
               (hierarchy: toplevel-static)
               (structure: (queue-of: (base: print-queue-entry)))
               )

(object-class: print-queue-entry
               (hierarchy: toplevel-static)
               (structure: (record:
                             (attributes:
                              ((name: entry-id)
                               (value: entry-id-value))
                              ((name: owner)
                               (value: user-id))
                              ((name: contents)
                               (value: print-queue-entry-type))
                              )))
               )

(object-class: print-queue-entry-type
               (hierarchy: toplevel-static)
               (structure: (union:
                             (components:
                              visible-byte-sequence
                              formatter-byte-sequence)))
               )
```

Figure 8. Some examples of object class definitions.

```
(object-class: directory
               ....
               (structure: (record: (attributes:
                                     ....
                                     ((name: password)
                                      (foreign:
                                       ((applicability: TOPS-20)
                                        (values: password-type)))
                                     .... )))    )
```

Figure 9. Example illustrating foreign attributes.

system which has some similarity to UNIX in general structure but which differs in detail, we may include attributes which are not applicable within UNIX, but which do have meaning in the context of other operating systems. Such attributes are termed *foreign*. An example illustrating the notion of associating a password with a directory is given in Figure 9.

In this figure, note that the attribute `password:` is designated as foreign, and that its applicability to another system (in this case TOPS-20) is asserted as well. A query enquiring as to a method to set a directory password, say, would not be rejected as not understood; rather, the consultant would be able to respond with a statement that associating a password with a directory is not a situation found in UNIX, but rather only in TOPS-20. It would also be possible to attach an English-language explanation of how UNIX deals with directory access without protection, although we have not incorporated this in the present design.

The method is not without limitations, however. It seems to work rather well for operating systems such as TOPS-20, VAX/VMS,[6] and MS-DOS. These systems all conceptualize the working domain of the command language in reasonably similar ways, although they differ substantially in detail. On the other hand, operating systems such as VM/CMS conceptualize the entire notion of a user's working environment very differently than do UNIX and its relatives. At one point, we attempted to incorporate certain aspects of VM/CMS into the Yucca model, but found it to be almost impossible. Clearly, to address the modelling of users migrating from systems such as VM/CMS, much more sophisticated techniques are necessary. Currently, Yucca-* makes no accommodation for such needs within the formal model.

4.4. *Filtered attributes*

It is often necessary to specify constraints on the values of objects. In particular, mutations effected by commands will change the values of objects in particular ways which are most easily conceptualized by a change of constraint. Referring again to our running example query asking how to print a file with pageheaders and lines numbered on the laser printer, the output is to be a constrained version of the input, the constraints being identified by the addition of pageheaders and line numbers.

While it is in principle possible to represent such constraints by defining logical conditions which are to hold on printable byte sequences, such formalizations would be enormously complex, and the actual definitions themselves would be of little direct use. In modelling the notion of pagination, for example, we are not really interested in a microscopic definition of the conditions under which a byte sequence is paginated. Rather, it is enough to know that it is indeed paginated, and omit the irrelevant details.

To model this sort of constraint, we employ the notion of a *filtered attribute* in Yucca-*. Figure 10 illustrates the concepts relevant to representing pagination in this manner.

Syntactically, filtered attributes appear to be no different than do ordinary attributes in a record field. However, their use is fundamentally applicative rather than assertive. For example, we cannot directly speak of the contents of a file instance `#F` as paginated; rather, we must say that a particular version of `#F` is the result of applying a pagination filter *applied* to another byte sequence (which might be the contents of an earlier version of that same file).

Because of this applicative nature of filters, the order in which they are applied is significant. Referring again to Figure 5, it is asserted in the post-condition that the `lines-numbered` filter is applied to the result of applying the `paginated` filter to the contents of version `0` of `#I` to obtain the contents of

```
(object-class: visible-byte-sequence
               (hierarchy: (ISA: ASCII-sequence
                                 (condition: declared)))
               (filters:   ((name: numbered-lines)
                            (values: line-numbering))
                           ....
                           ((name: paginated)
                            (values: pagination))
                           .... ) )

(object-class: pagination
               (hierarchy: toplevel-static)
               (structure:
                 (record: (attributes:
                   ((name: lines-per-page)  (values: natural-number))
                   ((name: page-separator)  (values: separator-type))
                   ((name: header-size)     (values: natural-number))
                   ((name: trailer-size)    (values: natural-number))
                   ((name: start-page)      (values: natural-number))
                   ((name: stop-page)       (values: natural-number))
                   ((name: pageheader)      (values: printable-sequence)) ))))
```

Figure 10. Example illustrating filtered attributes.

version 1 of #PQE. It is clear that such ordering is necessary, as first numbering lines and then inserting pageheaders is quite different than the other way around.

Note from Figure 10 that the allowed value of a filter may be another object class. The allowed values for the paginated filter are taken from legal instances of objects of type pagination, which has a record structure. In the example query, the actual instance used is the default instance %paginated-standard-pageheaders, which will be discussed in more detail in 4.6.

4.5. *The display-object problem*

One of the most complex conceptual problems encountered in modelling the command language environment is the *display-object problem*. This problem arises because of the need to distinguish between an instance of an object and a display of that instance. For example, when posed a query asking how to find out who is logged onto the system, the proper response, technically speaking, is not a collection of users, but rather a file whose contents is a listing of the identifications of those users. Similarly, the proper response to the query asking how to find out the names of the files in a directory is not a list of files, but rather a file whose contents is a listing of those files. We humans make this translation so automatically that we are usually not even aware of it. Yet, this translation cannot be made automatically by a consultation system. The reason is at least twofold. First, the result needs to be displayed to the user, and we cannot display users; rather, we can only display byte sequences. In the formal model, it makes no sense to set the contents of standard output to a set of users. Second, such displays typically have special attributes not shared by the objects displayed. For example, it makes no sense to speak of placing files in columns, yet it makes perfect sense to display the names of files in several columns.

One way to address this problem, which was that taken in the earlier design of Yucca, is to define directly two classes of objects, one for direct representation and one for display. However, this masks the direct and systematic relationship between objects and their displays, and forces a great deal of manual translation of attributes to displays. To address this problem more systematically, in Yucca-* we employ the notion of a *macro meta-object*. In most general terms, a macro meta-object is a macro which takes as arguments one or more object classes, and generates a new object class from it. The definition of the macro meta-object which generates a display of the attributes of a set of objects, together with an instantiation yielding an object class definition for the display of a set of users, is shown in Figure 11.

Every macro meta-object must have at least one source parameter. The `source-parameters:` entry identifies the object class or classes which must be supplied as parameters for the expansion. In a display macro, it identifies that which we are to display. In the example of Figure 11, the source parameter is a set of objects of type `?IS`. The `external-parameters:` defines the actual parameter types which must be bound in order to generate the instantiation. In effect, they serve as parameters for the source parameters. In the example, this parameter is `?IS`, which represents the base type of the set to be displayed. The macro `display-set-attributes` places no constraints on it, but the instantiation `collective-user-listing` binds it to be of type `user` (since we are generating a display of users). The `generators:` field of the macro identifies how the input parameter will be used to generate the attribute structure in the expanded macro. In the example, another macro, which generates a display for attributes of a record, is invoked.

In addition to the attributes generated in this fashion, there are two sources of `special-attributes:`. First of all, the macro itself provides attributes which apply to any instantiation of it. In the example of Figure 11, there are four such attributes, identifying criteria for sorting the objects as well as certain display formats. Additionally, the instantiation definition may define additional attributes in its `generation-structure:` component. In this example, there is just one, `user-listing-display-format`, which in effect decides whether it is a "`w`-command-like" listing or "`f`-command-like" listing which is desired.

Although this syntax seems quite imposing, the actual *expanded* instantiation is much more understandable. It is shown (in part) in Figure 12.

```
(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))
                      ((name: display-format)
                       (values: generic-display-format))
                      ((name: column-format)
                       (values: generic-column-format)) ))

(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 11. Example of a macro meta-object and an instantiation.

```
(object-class: collective-user-listing
 (hierarchy: (ISA: (macro-expand: display-set-attributes
                                  (external-parameter-bindings:
                                   (individual-source: user)))
                   (condition: declared)))
 (structure: ((name: id)               ;; Attributes from type user.
              (values: simple-select));; Only a few are shown here.
             ((name: account)
              (values: simple-select))
             ...
             ((name: sort-criterion);; Attributes from macro.
              (values: generic-sort-criteria))
             ((name: sort-direction)
              (values: direction))
             ((name: display-format)
              (values: generic-display-format))
             ((name: column-format)
              (values: generic-column-format))
             ((name: display-format);; Attribute from instantiation definition.
             (values: user-display-listing-format)) ) )
```

Figure 12. Example of a full expansion of an instantiation.

%master-system	—	denotes the (unique) master system.
%login-table	—	denotes the login table of **%master-system**.
%user	—	denotes the user id of the user presenting the query.
%terminal-connection	—	denotes the terminal connection of **%user**.
%standard-input	—	denotes the standard input file of **%terminal-connection**.
%standard-output	—	denotes the standard output file of **%terminal-connection**.
%command-processor	—	denotes the command processor of **%terminal-connection**.

Figure 13. Some major global variables.

All of the attributes of the object class `user` are included, but the `values:` field of each attribute has been changed. For example, in the definition of object class `user:`, the allowable values for the field `id` are in the type `user-id`. But in the display object, the allowable values are in `simple-select`, which is just a two-valued type indicating whether or not the value of the attribute is to be displayed. By selecting appropriate values for each field in this expansion, we obtain a detailed and complete description of the format of a display of users.

4.6. *Global and default instances*

Global instances

There are several global object instances which retain their meaning permanently. Several of the more important such objects are tabulated in Figure 13.

There are also several more "local" global variables which apply to a particular installation, but not globally to UNIX systems. The variable `%laser-print-queue` used in the query of Figure 5 is one such example.

Default instances

In addition to global instances, there are several *default instances* which are used to express queries more succinctly. A particular example is `%paginated-standard-pageheaders`, which is used to express the pagination condition in the query of Figure 5. Its definition is shown in Figure 14. Refer to the definition of `pagination` in Figure 10. Basically, it represents the default mode of pagination which is understood, via the formal query language, to both the front end and the back end. This instance is not absolutely necessary; rather, it could be constructed from scratch each time that it is needed. But it is far more efficient to define it once, and use it repeatedly. Of course, this default may be modified if so needed. For example, if the formal query called for 60 lines per page, the syntactic modification shown

```
(instance:
 (class: pagination)
 (identification:
  ((name: %default-standard-pageheaders)
   (version: fixed:)
   (record: (attributes:
             ((name: lines-per-page)  (value: 66))
             ((name: page-separator)  (value: blank-lines))
             ((name: header-size)     (value: 5))
             ((name: trailer-size)    (value: 5))
             ((name: start-page)      (value: 1))
             ((name: stop-page)       (value: nil))
             ((name: pageheader)      (value: %standard-pr-pageheader))))))
```

Figure 14. Definition of a default instance.

```
        ...
   (constraint: (modified: %paginated-standard-pageheaders
                           ((field: lines-per-page) (value: 60))))
        ...
```

Figure 15. Syntactic modification to locally alter a default instance.

in Figure 15 within the query would accomplish that task. This approach is most appropriate if a large instance has only a few differences from a default instance.

Note that embedded within the definition is another default instance, `%standard-pr-pageheader`.

5. The dynamic knowledge base

5.1. *Principles of command semantics representation*

If the static knowledge base is considered to be the foundation of the back end, the dynamic knowledge base is the heart. It is in this knowledge base that the semantics of the UNIX command language is embodied. The basic philosophy is very simple. The essence of command language behavior is represented by a relatively small collection of primitive descriptions, together with an interconnection calculus for combining these primitives into more complex ones.

Primitive components

The syntax of a primitive component (also called a *simple dynamic-object class* or *simple module*) is not unlike that of a formal query (although it will generally be much less complex), save that primitive components do not contain free variables. Figure 16 provides a more formal description.

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

Figure 16. Syntax of dynamic object class definition.

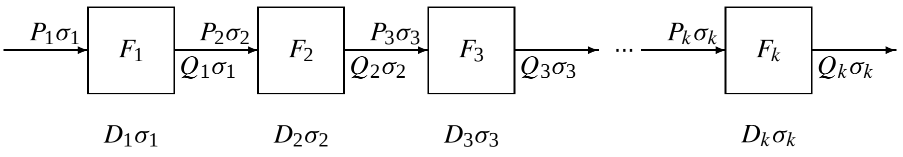

Figure 17. Serial interconnection of dynamic modules.

Each of the four components is shown to be optional. This is just an indication that they may default to certain values when not specified; this will be elaborated upon in 5.2. For the time being, assume that all components are present.

Interconnection calculus

At present, only serial interconnection is supported. The basic idea is as follows. Let $D_1 = \langle P_1, F_1, Q_1, A_1\rangle$ and $D_2 = \langle P_2, F_2, Q_2, A_2\rangle$ be simple components, with P_i denoting the precondition, F_i the action, Q_i the postcondition, and A_i the actor of D_i. Let $D_i\sigma$ denote the module resulting from the application of the *unifier* σ to each component of the dynamic description.[7] Then we may interconnect D_1 to D_2 sequentially with unifier σ_1 applied to D_1 and σ_2 applied to D_2 if and only if $Q_1\sigma_1 \models P_2\sigma_2$. In other words, we require that the postcondition of the first module be strong enough to imply the precondition of the second. A simple component interconnection may thus be viewed as a sequential composition, as illustrated in Figure 17.

Adequacy of the interconnection calculus

The utility of this interconnection scheme arises from the fact that not only are simple commands represented by such modules, but so too are their options. Thus, to realize the UNIX command "`pr -h myheader -l60 foo`", we use the interconnection depicted in Figure 18 below.

The module `pr-command-main` contains the main definition of the semantics of the optionless `pr` command, while the modules `pr-change-`

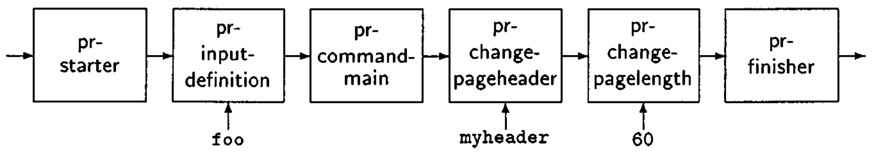

Figure 18. Realization of the command "`pr -t myheader -l60 foo`".

`pageheaders` and `pr-change-pagelength` contain the definitions for the "`-t`" and "`-l`" options respectively. The other three modules are for command support, and will be discussed in 5.2. The important point here is that the semantics of a command may be realized with the use of three types of simple modules.

1. A main module describing the behavior of the basic command. This module is included in any interconnection realizing the associated command.
2. A collection of modules, one for each option, describing the behavior of that option. A given module from this class is included in the interconnection only if the associated option is selected.
3. A fixed set of "support" modules which are always included in any realization of the given command.

In addition to modules defining the behavior of individual commands, there are also modules defining the action of the *command interconnectors* pipe "`|`" and sequence "`;`", *command grouping* via the brackets "`(..)`", and *redirection* of input and output. Thus, not only can individual commands be realized within this framework, but interconnections of more than one command can be also.

Limitations of the interconnection calculus

At this point, we are modelling only the serial interconnection of commands via sequencing and piping, and redirection of input and output. More sophisticated shell programming constructs, such as conditionals and loops, are not supported. This decision has been made largely in the interests of practicality. To support arbitrary programming language style constructs would move Yucca-* out of the domain of a simple command language consultation system and into the domain of a sophisticated programming aid, such as the *Programmer's Apprentice* (Rich 1981; Waters 1985a; Waters 1985b). Such an extension would add enormously to the complexity of the project.

```
(define-command-manager <name>
  (define-interconnection-categories: <ic-description>)
  (define-interconnection-syntax: <is-description>)
  (define-object-classes: <oc-description>))
```

Figure 19. Overall syntax of a command manager.

5.2. *Command managers*

The problem with a flat organization

It is quite possible to regard the dynamics database as a flat collection of simple modules, in much the same way that one would regard a logic database as a flat collection of clauses. Indeed, this was precisely the approach taken in the earlier Yucca system. However, we found that this leads to at least two difficulties. First, it becomes necessary to incorporate into the definition of each command support and option module specific logical "interconnection" preconditions and postconditions which permit the use of those support modules only with the associated command, and only in the proper sequence. It must be remembered that this interconnection scheme is only *logical*; we cannot realize each simple module separately. For example, since it is physically impossible to realize the `pr` command with an option from the `cat` command, the formal representation must prohibit this as well. The incorporation of these interconnection constraints contributes markedly to the complexity of the precondition and postcondition representations. Second, it fails to make use of critical information which can aid in the efficient search for a proper plan of action; namely, that it is possible, on purely syntactic grounds, to eliminate almost all possible interconnections simply because they associated options or support from one command with those of another.

Organization by command

In Yucca-*, we have addressed the problem indicated in the previous paragraph by organizing the simple modules associated with each command into a single unit known as a *command manager*. In addition, there are separate command managers for the redirection, command grouping, and sequential interconnection operators. Each command manager contains three distinct components, as identified in the syntactic description of Figure 19.

There are three main types of information embodied in a command manager.

1. The *object classes* entry contains the semantic description of each simple module associated with that command. This includes the *main module*, the *option modules*, and the *auxiliary modules*.
2. The *interconnection syntax* embodies two main entities:

```
(define-command-manager: pr-command-manager
  (define-interconnection-categories:
   (stream-input:  <conditions for interconnection to accept stream of input>
                   <criteria to identify input port>)
   (file-input:    <conditions for interconnection to use a file as input >
                   <criteria to identify input port>)
   (stream-output: <output as stream is mandatory>
                   <criteria to identify output port>) )
 (define-interconnection-syntax:
   (sequence:
     ;; Modules used must be interconnected in this order.
     ;; Formal interconnection constraints may thus be omitted.
     <syntax for pr-starter>                ; mandatory
     <syntax for pr-input-definition>       ; mandatory + optional argument
     <syntax for pr-command-main>           ; mandatory
     <syntax for pr-change-pageheaders>     ; optional + mandatory argument
     <syntax for pr-drop-header-trailer>    ; optional
     <syntax for pr-change-pagelength>      ; optional + mandatory argument
     <syntax for pr-change-startpage>       ; optional + mandatory argument
     <syntax for pr-change-page-separator> ; optional
     <syntax for pr-finisher>               ; mandatory ) )
 (define-object-classes:
       ;; Order is not important here.
       <axiomatic semantics for pr-command-main>
       <axiomatic semantics for pr-starter>
       <axiomatic semantics for pr-finisher>
       <axiomatic semantics for pr-input-definition>
       <axiomatic semantics for pr-drop-header-trailer>
       <axiomatic semantics for pr-change-pageheader>
       <axiomatic semantics for pr-change-pagelength>
       <axiomatic semantics for pr-change-page-separator>
       <axiomatic semantics for pr-change-startpage> )
)
```

Figure 20. Structure of the command manager for the `pr` command.

  – Direct rules for interconnecting the simple modules, without need to resort to the checking of formal logical interconnection rules.
  – The information necessary to associate module interconnections with command-line syntax, including user-supplied parameters.

3. The *interconnection categories* define the information necessary to interconnect one entire command to another. In effect, they define the "port" characteristics of the command. They are discussed further in 5.3.

We now illustrate the concept by examining in detail the command manager for the `pr` command. The overall composition of this manager is depicted in Figure 20 below. Due to its length, the complete *formal* definition of the command manager is contained in the Appendix.

There are several points which should be emphasized relative to this formalism.

1. The semantic description of each individual simple module does *not* contain any formal constraints on the interconnection. Indeed, examination of the formal definitions in the Appendix reveals that many of the dynamic object classes have no preconditions and/or postconditions. Since the interconnection syntax defines precisely the way in which these modules may be interconnected, such conditions are unnecessary.
2. The modules may contain common variable names which do not represent the same thing. This is not possible in a flat organization, since accidental variable name collision might then lead to erroneous conclusions.
3. The action component is not given for any of the modules. This information is obtained entirely from the interconnection syntax.
4. The actor is listed only in the `pr-starter` and `pr-command-main` modules. The actor is assumed to be the same for all components of a local interconnection within a command manager, and hence need not be specified repeatedly.

Now let us examine the interconnection depicted in Figure 18 in more detail.

1. The rôle of the `pr-starter` module is twofold. First, it sets up the initial conditions for execution of a `pr` command. In particular, it establishes the definition of the default pagination conditions of the command via the pagination object `#P`. Second, it identifies the actor associated with the action. This latter value defaults to `%terminal-connection`, as identified in the interconnection syntax specification, but may be changed by altering the input parameter `$actor`. As indicated in the interconnection syntax, inclusion of this module is mandatory in any realization of `pr`.
2. The rôle of the `pr-input-definition` module is to identify the source of input to the command. As indicated in the interconnection syntax, this value defaults to `%standard-input` if not supplied via the parameter `$user-supplied-input-file`. This module is also mandatory.
3. The rôle of the `pr-command-main` module is to represent the core of the command. In this case, it asserts in its preconditions the existence of the mutable objects associated with its input and output, and asserts that both must be readable. In its postconditions it asserts that the contents of the input is copied to the output. Note that no pagination or the like has been applied at this point, although the actual conditions are carried through unchanged.
4. The rôle of the `pr-change-pageheader` module is to change the pageheader to the user supplied value, delivered via the variable `$user-supplied-header`. It is specified via an `updated:` operation on the current instance of the pagination object `#P` to yield a new version.

stream-input	—	The input to the command is from default standard input.
file-input	—	The input to the command is a regular file.
file-sequence input	—	The input to the command is a sequence of files.
no-input	—	The command takes no input.
other-input	—	The input convention is none of the above.
stream-output	—	The output from the command is to default standard output.
file-output	—	The output from the command is to a regular file.
queue-input	—	The command requires a queue as input.
queue-output	—	The command generates a queue value as output.
enqueue-output	—	The command enqueues its output.
other-output	—	The output from the command is none of the above.

Figure 21. Interconnection categories.

It is an optional module, and may be included or excluded in any interconnection.

5. The rôle of the `pr-change-pagelength` serves a similar function in changing the pagelength.
6. The module `pr-finisher` must terminate any realization of `pr`. It actually applies the filter which was assembled by `pr-starter` and modified by the option filters to obtain the final output object.

It is important to observe that conditions which are not affected by a given filter progress through the interconnection. For example, since the option filters do not affect any of the objects except for `#P`, these other objects are not altered by the option filters.

5.3. *Interconnection categories*

Each command manager contains the information necessary to describe the assembly of its associated command. However, many query solutions will require the interconnection of several commands. To facilitate the characterization of *command* interconnection, each command module characterizes the results that it produces in terms of *interconnection categories*. Presently, there are eleven interconnection categories in the Yucca-* formalism. Six categorize input and five output. They are tabulated in Figure 21.

The idea is that only commands which match up may be interconnected. For example, to interconnect via a pipe, the first command must be of type stream output and the second of type stream input. This feature is utilized extensively in the realization of solutions to unknown action queries, as described in the next section.

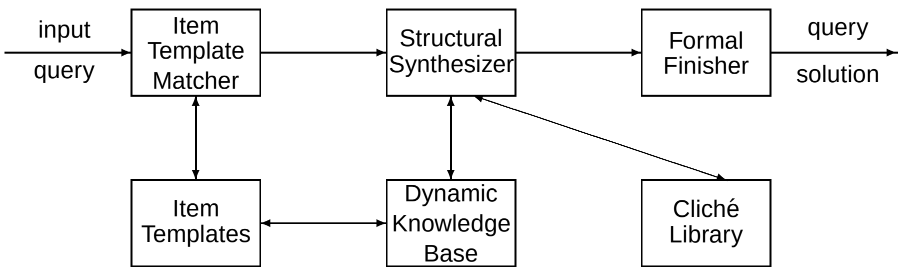

Figure 22. Architecture of the command synthesizer.

6. Solution of unknown action queries

6.1. *The overall architecture*

As noted earlier, the process of solving an unknown action query may be regarded as one of plan generation. We are given an initial state (the precondition identified in the formal query), a final state (the postcondition), and a set of possible actions (the dynamics knowledge base), and we must find a plan (i.e., an interconnection of simple modules) which realizes the goal. In recent years, there has been a great deal of progress in the general area of planning, as may be found in the recent anthologies (Brown 1987) and (Georgeff and Lansky 1986). However, we believe that it is still the case that special techniques which take advantage of the particular application at hand will yield far more efficient solutions than will the application of general tools. To this end, we have developed a highly specialized architecture, the *command synthesizer*, customized to the goal of solving unknown action queries in the UNIX domain. Shown in Figure 22 is its architecture.

There are three main steps in the solution process.

1. *Selection.* In this step, the main commands and options which are candidates for participation in the final solution are selected, without particular regard for the way in which they will be interconnected. This function is executed by the *item-template matcher*, with the support of the *item-template database*, and is elaborated in 6.2.
2. *Interconnection.* In this step, a plan for interconnection of the components selected in step 1 is assembled. This function is executed by the

structural synthesizer, with the support of the *cliché library*, and is elaborated in 6.3.
3. *Refinement and binding*. In this step, the interconnections assembled in step 2 are used to identify the actual variable bindings necessary to solve the query. In addition, the solution of necessary secondary subgoals is addressed at this step. All of this is executed by the *formal finisher*, and is elaborated in 6.4.

6.2. *Item templates and the item template matcher*

Consider once again the example query asking how to print a file with page-headers and lines numbered on the laser printer, and its formalization of Figure 5. We know, as UNIX experts, that

```
cat -n filename | pr | lpr -Plaser
```

is a solution to this query, assuming that our laser printer is designated by the system name "`laser`". In effect, the above solution is a "plan" which the consultant must assemble. In the first step in assembling this plan, we take advantage of a significant feature of the command language domain. Namely, by examining the postconditions of the plan to be constructed, we can determine immediately a "core" set of participating commands. For the example query, the selections are as follows.

1. Because a new print queue entry is being generated, any plan must include the UNIX command which enqueues entries into print queues; namely, the `lpr` command.
2. Because the particular queue involved is the laser printer queue, the option on `lpr` used to specify a particular queue (the "`-P`" option) must be involved also.
3. Because the new print queue entry must have its lines numbered, the command which accomplishes this task (the "`-n`" option of the `cat` command) must be involved in the solution.
4. Because the new print queue entry must be paginated, the command which accomplishes pagination (the `pr` command) must be a participant in the solution.

Of course, there are other participants in the final solution, such as the two pipes. Other queries might require redirection or command grouping as well. However, the "core" commands of the solution have been identified.

The item templates

To implement effectively a process which selects commands and options as outlined above, a method of efficiently identifying those simple modules

```
((name <attribute-name>)
  (alter-value:            <list-of-dynamic-object-classes>)
 [(observe-value:          <list-of-dynamic-object-classes>)]
 [(alter-default-value:    <list-of-dynamic-object-classes>)]
 [(observe-default-value:  <list-of-dynamic-object-classes>)])
```

Figure 23. Syntax of an item template.

```
(define-item-template:
  (class: pagination)
  (structure:
   ((name: lines-per-page) (alter-value: (pr-change-pagelength)))
   ((name: page-separator) (alter-value: (pr-change-page-separator)))
   ((name: header-size)    (alter-value: (pr-drop-header-trailer)))
   ((name: trailer-size)   (alter-value: (pr-drop-header-trailer)))
   ((name: start-page)     (alter-value: (pr-change-startpage)))
   ((name: stop-page)      (alter-value: nil ))
   ((name: pageheader)     (alter-value: (pr-change-pageheader)))))
```

Figure 24. Example of an item template.

which effect a particular mutation is needed. This function is provided by the *item templates*. Basically, the item templates form an index from attributes of static object classes to those simple dynamic object classes which alter and/or observe their values. The overall syntax of an item template is shown in Figure 23.

For a given attribute, the only entry which is mandatory is that used for altering its value. Figure 24 provides the definition of the item template for the `pagination` object class.

Notice that the association between attributes and simple modules is neither injective nor surjective. For example, the `pr-drop-header-trailer` module, an option of the `pr` command, is associated with both the `header-size` and `trailer-size` attributes of the pagination object. This is because this module is the (only) way to alter the sizes of the margins in a pagination; they must be modified simultaneously. On the other hand, there is no way to define the page at which printing stops, so no command is associated with the `stop-page` attribute.

The item template matcher

The *item template matcher* is basically a peruser of formal queries. Its primary function is to examine the formal query and identify the mutations which are mandated to take place, and then to select the appropriate simple dynamic modules which are necessary to effect such a mutation. It does so in two passes.

Simple Module	Binding Within Query
`lpr-command-main`	`(instance: .. #I .. print-queue)`
`lpr-option-select-printer`	`%laser-print-queue`
`cat-command-number-all-lines`	`(apply-filter: .. numbered-lines) ..`
`pr-command-main`	`(apply-filter: .. paginated) ..`

Figure 25. Synopsis of output of the item template matcher for example query.

1. In the first pass, it sets up a table which identifies explicitly all declared objects occurring in either the precondition or postcondition of the query. It also flags them as fixed or potentially mutated.
2. In the second pass, it examines the prescribed mutations of those objects which are potentially altered. This examination will identify simple attributes of those objects whose values are changed, or filters to be applied to those objects. For each of these simple attributes and filters, the appropriate commands are selected, with the aid of the item templates.

The collection of simple dynamic modules so selected, together with bindings to the specific clauses in the formal query to which they apply, is the output of the item template matcher. It is important to emphasize that no interconnection or ordering of these modules is performed at this step. In the terminology of Sacerdoti (Sacerdoti 1977), this first stage of planning is totally nonlinear. For the example query, the set of modules returned, together with their query bindings, is shown in Figure 25.

For simplicity, we have shown only a fragment of the formal query associated with the command, while the actual algorithm returns the *entire* query with pointers into it from the simple modules selected.

Universality of this approach

It is central to the success of this approach that all unknown action queries be representable by mutations on objects. From our empirical investigations, this does seem to be the case, even when at first thought a query might appear to be of a different form. For example, the query asking simply how to delete a file might appear to be one of deleting an object rather than altering an attribute. However, upon further consideration, it becomes clear that deleting a file is equivalent to altering the directory in which it resides. Thus, in this query, the mutated object is not the file to be deleted, but rather its parent directory. Viewed in this fashion, our approach works well. The item template for the `entries` attribute of the object class `directory` points to the simple dynamic module `rm-command-main` (as well as several simple modules, such as `mv-command-main` and `cp-command-main`, which is exactly what we want).

Other entries in item templates
In Figure 23, observe that there are four possible entries for an individual item template, rather than just one, with the latter three optional. These optional forms are carryovers from the earlier Yucca system, but still may be used in alternative query formulations. For example, consider the query asking how to obtain a directory listing. The "preferred" formalization in Yucca-* would call for the delivery of a byte sequence on the user's standard output which is a display of the directory contents. However, in the earlier Yucca system, this could have been formalized as a delivery of the directory contents, with a display filter applied. Since there is no direct display object identified in this case, no `alter-value:` field is applicable. However, the `observe-value:` field of the attribute `entries` of a directory points to the simple module `ls-command-main`, which is the key participant in any solution. We are retaining these alternate fields at this point because Yucca-* is a highly experimental system, and the possibility that query formalization and solution along this alternate path may be a better ultimate choice remains.

6.3. *Clichés and the structural synthesizer*

The item template matcher delivers the identities of several simple dynamic modules, together with associations to clauses in the formal query, to the structural synthesizer. The task of the structural synthesizer is to interconnect these simple dynamic modules in a fashion suitable for ultimate query solution. The actual process consists of two steps, *individual command assembly* and *cliché selection and interconnection assembly*. We now elaborate upon these steps.

Individual command assembly
The item template matcher delivers to this unit a collection of individual simple dynamic modules, together with query clause attachments. The first step is to take these isolated modules and assemble complete interconnections for individual commands. As all of the information necessary to assemble a complete command is contained in the associated command manager, this is usually a completely straightforward process. However, there are two cases in which complications may occur.

1. It may be the case that multiple instances of the same command are necessary. There are two ways in which this may occur. First, different command options may be associated with the alteration of distinct objects. This occurs very rarely, but it is possible. A rather contrived example is the formalization of the query asking how to concatenate two files with the lines of the first numbered and the control characters made visible in the second. A solution using two distinct instances of `cat` is

(`cat -n` *file1*`;` `cat -v` *file2*`)` `>` *result*.

No solution using only one instance is possible. Second, it may be necessary to interlace filters. This also occurs rarely, but is possible. For example, consider the formalization of the query asking how to process a file so as to paginate it, number lines, use formfeeds to separate pages, and make those formfeeds visible. A solution is

`cat -n` *filename* | `pr -f` | `cat -v >` *result*

which requires two distinct instances of the `cat` command. Because the pagination must be done after the insertion of line numbers but before the marking of the visible characters, no solution with only one instance of `cat` is possible.

2. It may be the case that incompatible options have been identified. For example, the `pr` command options "`-h`" (use user-supplied page-header) and "`-t`" (essentially, do not paginate) are *logically* incompatible, although the command allows both to be specified and simply ignores the pageheader. This incompatibility is known to the `pr` command manager, and a request to include both in an interconnection would result in an error. In such an occurrence, the maximal compatible commands will be synthesized (the representations of both "`pr -h`" and "`pr -t`" in this case), but these results will be flagged as "incomplete".

In any case, that which is returned is a collection of assembled commands. For the example query, the three assemblies shown in Figure 26 are obtained.

Cliché selection and interconnection assembly

Once the individual commands which may participate in the solution are selected, the next step is to interconnect them properly. There are four low-level interconnection operations.

1. *Piping*. This is the standard UNIX output-to-input connection.
2. *Sequencing*. This is simply sequential execution of commands, as in

   `cat -n` *file*`; pr`.

3. *Grouping*. This forces commands to be assembled in blocks, and is represented syntactically in UNIX with parentheses. An example, given earlier in this section, is

   (`cat -n` *file1*`;` `cat -v` *file2*`)` `>` *result*.

4. *Redirection*. The current model includes input redirection "<", as well as the three forms of output redirection, ">", "»", and ">`!`".

It must be remembered that the applicable interconnection operators are usually not identified by the item template matcher. Therefore, before the

```
(assembled-command:
 (manager: cat-command-manager)
 (free: ?actor ?infile )
 (sequence:
  ((cat-starter         (parameter: $actor ?actor))
   (cat-input-definition (parameter: $user-supplied-input-file ?infile))
   (cat-option-number-all-lines)
   (cat-command-main)
   (cat-finisher)))
 )

(assembled-command:
 (manager: pr-command-manager)
 (free: ?actor ?infile)
 (sequence:
  ((pr-starter         (parameter: $actor ?actor))
   (pr-input-definition (parameter: $user-supplied-input-file ?infile))
   (pr-command-main)
   (pr-finisher)))
 )

(assembled-command:
 (manager: lpr-command-manager)
 (free: ?actor ?infile ?pname)
 (sequence:
  ((lpr-starter              (parameter: $actor ?actor))
   (lpr-input-definition      (parameter: $user-supplied-input-file ?infile))
   (lpr-option-select-printer (parameter: $user-supplied-printer-name ?pname))
   (lpr-command-main)
   (lpr-finisher)))
 )
```

Figure 26. Examples of assembled commands.

identified commands can actually be interconnected, it is necessary to select the appropriate connectives from the above list. While this could be done by directly examining the formal query and the way in which the commands are bound to its clauses, this would prove to be rather inefficient. Rather, we take advantage of the fact that command interconnections almost always occur in ways based upon a small number of simple patterns, known as *clichés*. Our use of clichés was inspired by a similar technique utilized in the Programmer's Apprentice system (Rich 1981), (Waters 1985a), (Waters 1985b). In that system, clichés are used to model common control patterns used by programmers. Our clichés are much simpler, representing only sequential interconnection. A few of the more common clichés employed in Yucca-* are given in the table of Figure 27.

Clichés have *slots*, which must be filled with objects of the appropriate *interconnection category*. As concrete examples, the two clichés to be used in the solution of our example query are given in Figure 28 below.

The actual process of cliché selection and command interconnection proceeds by the following algorithm.

Apply a filter and redirect the output to a file.
Use a file for filter input.
Redirect both the input and the output of a filter.
Filter a file and print the result.
Compose two filters.
Roff or TEX a source file.

Figure 27. Some common clichés.

```
(define-cliche:
  (name: file->out-filter<->print-queue)
  (result-type: (AND: stream-input queue-input enqueue-output))
  (nondefault-components:
   (name: initial-filter
          (categories: (AND: file-input stream-output)))
   (name: print-module
          (categories: (AND: stream-input queue-input enqueue-output))))
  (interconnection-constraint:
   (pipe-couple: (arguments:(output-line: initial-filter)
                            (input-line: print-module)))) )

(define-cliche:
  (name: file->out-filter<->in-out-filter)
  (result-type: (AND: file-input stream-output))
  (nondefault-components:
   (name: first-filter
          (categories: (AND: file-input stream-output)))
   (name: second-filter
          (categories: (AND: stream-input stream-output))))
  (interconnection-constraint:
   (pipe-couple: (arguments:(output-line: first-filter)
                            (input-line: second-filter)))) )
```

Figure 28. Definitions of some common clichés.

1. *Determine temporal order*

The formal query, together with the bindings of assembled commands to clauses, is examined to determine a *temporal partial order* on the fully composed commands. $C_1 < C_2$ means that execution of command C_1 is to take place prior to the execution of C_2. This order is determined by several factors, including nesting in `apply:` and `append:`[8] operators. The ordering is partial because in many cases it is not possible to determine easily at this point the appropriate order; indeed, there are cases in which order is not relevant. In the example query, the assembled `cat` command is determined to precede temporally the assembled `pr` command, because of the order of application of the filters. However the assembled `lpr` command is not temporally comparable to either at this point. This is because the print queue entry and result of the `cat-pr` filter are *asserted* to be equal, and there is no implied temporal ordering on the arguments of "=".

2. *Determine equivalence*
The commands are associated in a *quasi-equivalence*, relative to specific functions within the query. Two commands will be determined to be quasi-equivalent if they were selected as "alternates" for the same purpose by the item template matcher. For example, under certain conditions, the UNIX commands `cat`, `pr`, `head`, `tail`, and `cp` can all have identical effects. Thus, it is possible that more than one may be selected as a candidate for a particular application. In our example, there are no quasi-equivalent commands.

3. *Determine interconnection category*
The interconnection category of the entire formal query is determined by examination. In the case of the running example of printing a file on the laser printer with pageheaders and lines numberered, it is

```
(AND: file-input queue-input enqueue-output)
```

because we are taking an input file, processing it, and then placing it into a print queue. This is easily determined because `#I` and `%laser-print-queue` are the only objects declared in the precondition, and `%laser-print-queue` is the only object to be mutated.

4. *Send on simple results*
If a command has been selected which is not comparable temporally to any other command, and it has an interconnection category binding which matches the interconnection category of the entire query, then that command is passed along, together with the formal query, to the next step. In the example query, this does not occur.

5. *Select and fill clichés*
All clichés whose interconnection categories match those of the query are identified. The slots of these clichés are filled, recursively if necessary. In the example query, the only cliché identified at this step is `file->out-filter->print-queue`, which is defined in Figure 28. Its `print-module` slot is filled immediately with the assembled `lpr` command. However, its `initial-filter` slot cannot be filled immediately because there are two commands, the assembled `cat` and `pr` commands, still to be utilized. Rather, a recursive attempt to assemble these two commands in such a way as to fill the `initial-filter` slot is initiated. This attempt

```
(assembled-interconnection:
 (cliches: filter-input-file-and-print)
 (free: ?actor ?infile ?pname)
 (sequence:
  (cat-starter
   (parameter: $actor ?actor))
  (cat-input-definition
   (parameter: $user-supplied-input-file ?infile))
  (cat-option-number-all-lines)
  (cat-command-main)
  (cat-finisher)
  (pipe-interconnection-main)
  (pr-starter
   (parameter: $actor ?actor))
  (pr-input-definition
   (parameter: $user-supplied-input-file :default))
  (pr-command-main)
  (pr-finisher)
  (pipe-interconnector-main)
  (lpr-starter
   (parameter: $actor ?actor))
  (lpr-input-definition
   (parameter: $user-supplied-input-file default:))
  (lpr-command-main)
  (lpr-option-select-printer
   (parameter: $user-supplied-printer-name ?pname))
  (lpr-finisher) )
 )
```

Figure 29. The assembled command for query solution.

easily finds that the cliché `file->out-filter->in-out-filter` is exactly what is needed. The two commands are used to fill the appropriate slots of this filter, and the result is used to instantiate the `initial-filter` of the `file->out-filter->print-queue` cliché. Each such full instantiation is sent on to the formal finisher.

6. *Expand clichés*
The instantiated clichés are expanded. In the case of our example, the result is given in Figure 29.

6.4. *The formal finisher*

The interconnected commands determined by the structural synthesizer represent potential solutions to the query. However, the actual solution, in the form of bindings to query variables, must still be determined. It is the rôle of the formal finisher to achieve this task. The algorithm which it obeys is sketched as follows.

1. *Expand the interconnections*
The interconnections, as supplied by the structural synthesizer, are in the form of a list of simple modules to be executed in sequence. In order to ascertain

```
(expanded-command:
 (dynamics: ( (((state: P) (action: F) (state: Q)) (actor: A)) ) )
 (variables: #I #PQE #Q #C)
 (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: #Q) (version: 0))))
    (expand: (readable ((name: #I) (version: 0))))) )
 (define: Q
   (AND:
    (instance: (class: plain-file)
               (identification: ((name: #I) (version: 0))))
    (instance: (class: print-queue)
               (identification: ((name: #Q) (version: 1))))
    (instance: (class: print-queue-entry)
               (identification: ((name: #PQE) (version: 1))))
    (= (retrieve: entire-value:
                  (source: ((name: #Q) (version: 1))))
       (enqueue: (retrieve: entire-value:
                            (source: ((name: #Q) (version: 0))))
                 (retrieve: entire-value:
                            (source: ((name: #PQE) (version: 1))))))
    (= (apply-filter: (name: lines-numbered)
                      (constraint: (value: all))
                      (argument:
                       (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))) )) )
 (define: F
   (instance: (class: command-sequence) (identification: #C))
   (value-descriptor: (syntax: "cat -n | pr | lpr -Plaser")))
 (define: A
   (instance: (class: user) (identification: %user))) )
```

Figure 30. Full expansion of example command interconnection.

bindings to query variables, this sequence must be expanded and then identified with a single dynamic object, which we call the *solution dynamics*. The result will look much like a formal query, but there will be no secondary preconditions or postconditions, the command interconnection will be bound rather than a unknown, and external variables will be replaced by command variables. The result of this process is shown in Figure 30.

2. *Bind the variables*

Once the potential solutions have been expanded into actual dynamic formulas, it is possible to determine variable connections. This is done in two steps.

1. *Unify*. Variables in the solution dynamics are replaced with the corresponding ones of the formal query. In the example, the variable `#Q` has the value `%laser-print-queue` substituted for it. The command manager for `lpr` contains the information necessary to make the association between printer names and print queues.
2. *Bind the primary unknown*. The unknown command is bound to the command found in the command interconnection dynamics. In the example, the variable `?CS` would is bound to the solution command sequence.
3. *Identify secondary conditions*. Any mismatches in the preconditions and/or postconditions must be reconciled. This is done by relegating the mismatched components to secondary conditions. In the example query, the solution dynamics contains the precondition

   ```
   (expand: (readable: ((name: #I) (version: 0))))
   ```

   which does not match any formula in the precondition of the query. Therefore, it is relegated to be a secondary precondition, and the query variable `?SPP` is bound to this entire formula. Since there are no such formulas in the postcondition, the variable `?SPQ` is bound to the identically true formula.

Multiple solutions

All interconnections are processed in this fashion. The one which is actually returned is the one with the weakest secondary preconditions and postconditions.

Contradictory conditions

In relegating a formula to a secondary precondition or postcondition, it is necessary to ensure that it does not contradict any condition asserted by the formula. As long as the precondition (*resp.* postcondition) of the solution dynamics simply make additions to those of the formal query, no problem can arise. However, if the formal query asserted additional conditions not matched by the solution dynamics, care must be taken. For example, suppose that our example query were further expanded to ask how to print an unreadable file with pageheaders and lines numbered on the laser printer. The condition that `#I` be unreadable would now be asserted as a precondition of the query, in direct contradiction to the precondition of the solution dynamics. In such a circumstance, Yucca-* regards the contradictory precondition as a new subgoal, and formulates, within the formal finisher, two new unknown action queries, one to make the file readable, and another to restore it to its unreadable state. These two new queries are then solved, and interconnected to the original query to yield a final solution.

7. Conclusions and Project Status

7.1. *Conclusions and Future Directions*

The power of formal semantics
To model details of a system correctly, one needs to employ a *deep* model of that system. We have presented a detailed design for a UNIX consultation system in which the underlying model of UNIX employs a deep model based upon completely formal semantics. The major advantage of such a model is that it takes full advantage of the inherent structure of the underlying domain of command language behavior. We believe that, compared to similar systems which embed the modelling and planning mechanism into a general framework of knowledge representation and inference, Yucca-* will operate a greater level of detail with a more modest commitment of resources.

Formal specification and documentation
Although Yucca-* is an external utility which is to run on top of an existing system, important lessons relative to integral utilities have been learned also. Specifically, 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 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. This suggests that a parallel investigation directed at the formal specfication, implementation, and documentation of UNIX commands is an important direction, which we are currently initiating.

7.2. *Current Status*

Knowledge bases
The implementation of all Yucca-* back end knowledge bases is well along. We have completed definitions for about 150 static objects, as well as about fifteen UNIX commands and five clichés. In the near future, we expect to have constructed models for a total of about 25 commands and ten to fifteen clichés. Several dozen formal queries have been hand-generated in complete detail, and we are confident of the scope and power of OSquel.

Solvers
A protoptype command synthesizer is currently under development. Although some components have been been implemented in a skeletal fashion, the entire unit is not yet functional. Priority at this point is on the command synthesizer; other query solvers will be built only after the command synthesizer is operational.

8. Recent Developments on the Yucca-* Project

Since the original version of this report was completed in November of 1989, there has been further work on implementation of the concepts described in the previous sections. The bulk of the completed work is reported in two M.S. theses (Rachlin 1993) and (Sila 1993).

In (Rachlin 1993), John Rachlin presents an implementation of the planning mechanism for Yucca-*, written in Common Lisp. His implementation demonstrates rather conclusively that the original design, based upon item templates and command managers, is an effective one.

In (Sila 1993), Noreen Sila presents an implementation of a graphical menu-driven front end for Yucca-*. The implementation uses C++-based tools, including the Forms Library and the Silicon Graphics GL Library. A key feature of such front-ends is their simplicity. Recapturing both static and dynamic queries, the key feature of her design is its simplicity. While menu-driven user interfaces are not the most sophisticated, they are quite useable for a wide range of queries. She demonstrates that, using the proper tools, it is possible to construct a rather effective front-end with relatively little effort.

Notes

1 UNIX is a trademark of X/Open, Inc.
2 TOPS-20 is a trademark of Digital Equipment Corporation.
3 VM/CMS is a trademark of IBM Corporation.
4 MS-DOS is a trademark of Microsoft Corporation.
5 Actually, there is a slightly more general format, which is occasionally necessary, involving a sequence of actions. Such queries are not elaborated in this report.
6 VAX/VMS is a trademark of Digital Equipment Corporation.
7 We assume a basic familiarity with the rôle of unifiers in logical formulation. See, e.g. (Genesereth and Nilsson 1987) for details.
8 The `append:` operator is used to append the contents of two byte sequences together.

Appendix. Formal definition of `pr` command

```
(define-command-manager: pr-command-manager:

  (define-interconnection-syntax:
    (sequence:
     ((name: pr-starter)
      (status: mandatory)
      (parameter: $actor (default: %terminal-connection))
      (command-line-syntax: ""))
     ((name: pr-input-definition)
      (status: mandatory)
      (parameter: $user-supplied-input-file (default: %standard-input))
      (command-line-syntax: filename (position: trail)))
     ((name: pr-command-main)
      (status: mandatory)
      (command-line-syntax: "pr"))
     ((name: pr-drop-header-trailer)
      (status: optional)
      (condition: (disjoint: pr-change-pageheaders))
      (command-line-syntax: "-t"))
     ((name: pr-change-pageheader)
      (status: optional)
      (condition: (disjoint: pr-drop-header-trailer))
      (parameter: $user-supplied-header)
      (command-line-syntax: (string-append: "-h " $user-supplied-header)))
     ((name: pr-change-pagelength)
      (status: optional)
      (parameter: $user-supplied-pagelength)
      (command-line-syntax: (string-append: "-l" $user-supplied-pagelength)))
     ((name: pr-change-startpage)
      (status: optional)
      (parameter: $user-supplied-startpage)
      (command-line-syntax: (string-append: "+" $user-supplied-startpage)))
     ((name: pr-change-page-separator)
      (status: optional)
      (command-line-syntax: "-f"))
     ((name: pr-finisher)
      (status: mandatory)
      (command-line-syntax: "")) )
    )

  (define-object-classes:
    (dynamic-object-class: pr-command-main
      (hierarchy: basic-command-filter)
      (preconditions:
       (AND:
        (instance: (class: plain-file)
                   (identification: ((name: #I) (version: 0))))
        (instance: (class: visible-byte-sequence)
                   (definition:
                     (retrieve: record-entry:
                                (field: contents)
                                (source: ((name: #I) (version: 0))))
                     (identification: (name: #B) (version: 0))))
        (expand: (readable ((name: #I) (version: 0))))
        (expand: (readable ((name: #U) (version: 0)))) ))
      (postconditions:
       (AND:
        (= (entire-value: ((name: #I) (version: 0)))
           (entire-value: ((name: #O) (version: 1))))))
      (actor:
       (instance: (class: terminal-connection)
                  (identification: ((name: #U) (version: 0)))))
      )
```

```
(dynamic-object-class: pr-starter
  (hierarchy: starter-filter)
      (preconditions: (= ((name: #P) (version: 0))
                         (object-instance:
                          (class: pagination)
                          (%paginated-standard-pageheaders))))
      (actor:
       (= ((name: #U) (version: 0))
          %actor))
  )

(dynamic-object-class: pr-finisher
  (hierarchy: finisher-filter)
  (postconditions:
   (= ((name: #O) (version: #current))
      ((name: %standard-output) (version: #current)))
   (= (entire-value: ((name: #O)) (version: #current)))
   (apply-filter: (filter: ((name: #P) (version: #current)))
                  (target: ((name: #O) (version: 0)))))
  )

(dynamic-object-class: pr-input-definition
  (hierarchy: option-filter)
  (postcondition:
  (= ((name: #I) (version: #0))
     ((name: $user-supplied-input-file) (version: 0))))
  )

(dynamic-object-class: pr-drop-header-trailer
  (hierarchy: option-filter)
  (postconditions:
   (updated: ((name: #P) (version: $current))
             (action:
              (newvalue: record-entry: (name: header-size)
                         (value: 0))
              (newvalue: record-entry: (name: trailer-size)
                         (value: 0)))))
  )

(dynamic-object-class: pr-change-pageheader
  (hierarchy: option-filter)
  (postconditions:
  (updated: ((name: #P) (version: $current))
            (action:
             (newvalue: record-entry:
                        (name: pageheader)
                        (value: $user-supplied-header)))))
  )

(dynamic-object-class: pr-change-pagelength
  (hierarchy: option-filter)
  (postconditions:
   (updated: ((name: #P) (version: $current))
             (action:
              (newvalue: record-entry:
                         (name: lines-per-page)
                         (value: $user-supplied-pagelength)))))
  )

(dynamic-object-class: pr-change-startpage
  (hierarchy: option-filter)
  (postconditions:
   (updated: ((name: #P) (version: $current))
             (action:
              (newvalue: record-entry:
                         (name: start-page)
                         (value: $user-supplied-startpage)))))
  )
```

```
 (dynamic-object-class: pr-change-page-separator
   (hierarchy: option-filter)
   (postconditions: (updated: ((name: #P) (version: $current))
                              (action:
                               (newvalue: record-entry:
                                          (name: page-separator)
                                          (value: formfeed)))))
  )
 )

(define-interconnection-categories:
  (stream-input:
   (condition: (is-null: $user-supplied-input-file))
   (identification: (input-line: #I)))
  (file-input:
   (condition: (not: (is-null $user-supplied-input-file)))
   (identification: (input-line #I)))
  (stream-output:
   (condition: t)
   (identification: (output-line: #O)))
  ) )
```

References

Bates, M., Moser, M. G. & Stallard, D. (1986). The IRUS Transportable Natural Language Database Interface. In Kerschberg, L. (ed.) *Expert Database System, Proceedings of the First International Workshop*, 617–630. Kiawah Island, SC: Benjamin/Cummings.

Brachman, R. J. & Schmolze, J. G. (1985). An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* **9**: 171–216.

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

Chin, D. N. (1988). *Intelligent Agents as a Basis for Natural Language Interfaces*. Technical Report UCB/USD 88/396, Computer Science Division of EECS, University of California at Berkeley.

Douglass, R. J. & Hegner, S. J. (1982). An Expert Consultant for the UNIX Operating System: Bridging the Gap between the User and Command Language Semantics. In *Proceedings of the Fourth CSCSI/SCEIO Conference*, 119–127. Saskatoon: Canadian Society for the Computational Study of Intelligence.

Fenschel, R. S. & Estrin, G. (1982). Self-describing Systems Using Integral Help. *IEEE Transactions on Systems, Man, and Cybernetics* **12**(2): 162–167.

Finin, T. W. (1983). Providing Help and Advice in Task Oriented Systems. In *Proceedings of the Eighth IJCAI*, 176–178. Karlsruhe: Morgan-Kaufmann.

Genesereth, M. R. (1979). The Role of Plans in Automated Consultation. In *Proceedings of the Sixth IJCAI*, 311–319. Tokyo: Morgan-Kaufmann.

Genesereth, M. R. & Nilsson, N. J. (1987). *Logical Foundations of Aritificial Intelligence*. Morgan Kaufmann.

Georgeff, M. P. & Lansky, A. L., (eds.) (1986). *Reasoning about Actions and Plans, Proceedings of the 1986 Workshop*. Morgan Kaufmann.

Grosz, B. J. (1983). TEAM: A Transportable Natural Language Interface System. In *Proceedings of the 1983 Conference on Applied Natural Language Processing*, 39–45. Santa Monica: Association for Computational Linguistics.

Hart, P. E. (1982). Directions for AI in the Eighties. *SIGART Newsletter* **79**: 11–16.

Hayes, P. J. (1982a). Cooperative Command Interaction Through the Cousin System. In *Proceedings of the International Conference on Man/Machine Systems*. University of Manchester Institute of Science and Technology.

Hayes, P. J. (1982b). Uniform Help Facilities for a Cooperative User Interface. In *Proceedings of the 1982 NCC*. Houston.

Hayes, P. J. & Szekely, P. A. (1983). *Graceful Interaction through the Cousin Command Interface*. Technical Report CMU-CS-83-102, Carnegie Mellon University, Pittsburgh, PA.

Hecking, M., Kemke, C., Nessen, E., Dengler, D., Gutmann, M. & Hector, G. (1988). *The SINIX Consultant – A Progress Report*. Technical Report Memo Nr. 28, Universität des Saarlandes, FB 10 Informatik IV.

Hegner, S. J. (1988). Representation of Command Language Behavior for an Operating System Consultation Facility. In *Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications*, 50–55. San Diego: IEEE Computer Society.

Hegner, S. J. & Douglass, R. J. (1984). Knowledge Base Design for an Operating System Expert Consultant. In *Proceedings of the Fifth CSCSI/SCEIO Conference*, 159–161. London: Canadian Society for the Computational Study of Intelligence.

Mark, W., Wilczynski, D., Lingard, R. & Lipkis, T. (1980). *Research Plans in the Area of Cooperative Interactive Systems*. Technical report, USC/Information Sciences Institute.

Miller, P. L. (1984). *A Critiquing Approach to Expert Computer Advice: ATTENDING*. Pitman: London.

Motro, A. (1986). BAROQUE: A Browser for Relational Databases. *ACM Transactions on Office Information Systems* **4**: 164–181.

Neelkandan, H., Matthews, M. M. & Biswas, G. (1987). An Intelligent Assistance System in the Unix Domain. In *Proceedings of Third Annual Expert Systems in Government Conference*. Computer Science Press.

Quilici, A., Dyer, M. & Flowers, M. (1988). Recognizing and Responding to Plan-oriented Misconceptions. *Computational Linguistics* **14**(3): 38–51.

Rachlin, J. N. (1993). *Intelligent Planning in the UNIX Help Domain*. Master's thesis, University of Vermont.

Rich, C. (1981). A Formal Representation for Plans in the Programmer's Apprentice. In *Proceedings of the Seventh IJCAI*, 1044–1052. Vancouver: Morgan-Kaufmann.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Elsevier North-Holland.

Salveter, S. (1984). *Natural Language Database Update*. Technical Report TR# 84/001, Computer Science Department, Boston University.

Scragg, G. W. (1975). Answering Process Questions. In *Proceedings of the Fourth IJCAI*, 435–442. Tbilisi: Morgan-Kaufmann.

Sila, N. P. (1993). *The Design of A Graphical Interface for an Intelligent UNIX Consultation System*. Master's thesis, University of Vermont.

Waltz, D. L. (1978). An English Language Question Answering System for a Large Relational Database. *Communicatios of the ACM* **21**: 526–539.

Waters, R. C. (1985a). KBEmacs: Where's the AI? *AI Magazine* **7**(1): 47–61.

Waters, R. C. (1985b). The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering* **11**: 1296–1320.

Wilensky, R., Arens, Y. & Chin, D. (1984). Talking to UNIX in English: An Overview of UC. *Communications of the ACM* **27**: 574–593.

Wilensky, R., Chin, D. N., Luria, M., Martin, J., Mayfield, J. & Wu, D. (1988). The Berkeley UNIX Consultant Project. *Computational Linguistics* **14**(4): 35–84.

Zloof, M. M. (1977). Query-by-Example: A Data Base Language. *IBM Systems Journal* **16**: 324–343.