An Approach to the Functional Description of an Information Retrieval System Based on a Generalized Model[1]

Carolyn J. Crouch
Richard E. Nance[2]

TR 84-655
December 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

---

[1] To appear in Information Technology: [illegible] and Development.

[2] Computer Science Department Virginia Polytechnic Institute and State University.

An Approach to the Functional Description of an Information Retrieval System Based on a Generalized Model

Carolyn J. Crouch
Cornell University

Richard E. Nance
Virginia Polytechnic Institute
and
State University

ABSTRACT

In this paper, the authors present a high level, functional approach to the description of a generalized information retrieval system. The description is based on the top-down decomposition of the system into modules and processes and on an appropriate data abstraction.

The purpose of this paper is to describe the behavior of the system in terms of the component processes and the interactions of these processes in terms of inputs, outputs, and the associated transformations. It allows one to view the system as a collection of abstract processes, each of which is concisely defined via a notation that describes what is accomplished but not how such a process is to be implemented. Semantically irrelevant details are removed, thereby producing a nonprocedural description which not only elucidates system behavior but serves as a basis for subsequent formal specification.

An Approach to the Functional Description of an Information Retrieval System Based on a Generalized Model

Carolyn J. Crouch
Cornell University

Richard E. Nance
Virginia Polytechnic Institute
and
State University

Introduction

A primary concern in the design and development of software is the generation, integration, and control of complex systems. It has recently been noted that over 90 per cent of the total cost associated with current systems development is due to the cost of software [1]. Yet the production of well-structured software, inherently difficult by nature of the programming task itself, is made increasingly more difficult by the characteristics of contemporary software design. Specifically, most software produced today is designed to function not as an independent unit but rather as a part of a larger and more sophisticated complex system.

A critical issue in the design of complex systems now surfaces; namely, how may the development of such systems be guided and controlled? Two major approaches to the problem can be identified. One approach deals primarily with the proper methods of program development. Stepwise refinement, modular coding, structured programming, and the use of abstract data types typify this approach. Another approach, which has received increasing attention over the past ten years, is that of software specification.

Formal specification techniques allow the specification writer to describe system behavior without reference to its implementation. On the basis of such a specification, it is then possible to determine that the implemented system actually satisfies or is consistent with its specification. Thus formal specification techniques permit verification of the system (i.e., establishment of its "correctness" with respect to its specification). A number of formal specification techniques have been discussed in the literature, including among others axiomatic, algebraic, denotational, and operational approaches. A discussion of these techniques is beyond the scope of this paper; the interested reader is directed to the literature [2,3,4,5,6,7].

Formal specification techniques aim at producing an abstract view of system behavior. Yet the intelligibility of the resultant specification is often questioned. Such specifications tend to be excessively detailed, entail considerable overhead, and may tend to obscure rather than clarify the procedures being described. They are often difficult to understand, couched in terms and formalisms unfamiliar to the person(s) charged with their implementation. Moreover, with these techniques, concern centers on the "mathematical tractability" of the description rather than on giving a clear overall understanding of the system and its behavior [8]. The complexity introduced as a consequence of formalism detracts from our understanding of the system.

Another major concern arises in dealing with the specification of large, complex systems. In large system specification, the domain of the specification methods must be extended from (typically) stacks, symbol tables, and database views to entire systems. Efforts made in this direction include formal specification languages [9] and automated specification systems and tools [10,11]. Underlying many of these automated support systems are database management systems based on the traditional formal models.

Consider now the issues involved in the specification of an information retrieval (IR) system--a large, complex system encompassing such diverse activities as interactive query processing, automatic indexing, and file manipulation. In contrast to the database environment, there are no formal models on which an intrinsic part of the specification can be based. Moreover, although each method has its proponents, no consensus of opinion exists as to which method of specification is best suited to deal with the description of large systems. In fact, it would appear that no <u>one</u> formal specification technique can describe such a system (i.e., the information retrieval system) in an effective manner. (Some examples suggest that a combination of techniques would be most effective [12,13].)

Liskov and Zilles state that even if one is unable to describe an entire system, "the ability to define some of the modules used in constructing a system in a precise, formal way would be a major advance in the construction of reliable software" [2]. In order to accomplish these goals in an information retrieval environment, research must be directed at

(1) producing a clear, comprehensive view of the system, in terms of its components, processes, and their interactions;

(2) identifying the module(s) of the system for which formal specification is most important; and

(3) determining which method or combination of methods is best suited to describe the behavior of each module.

Our focus in this paper is item (1) above. We use abstraction as a means of dealing with the system; our purpose is to describe the behavior of the system in terms of the abstract objects (sets) being manipulated and the processes associated with them as the system operates over time. Thus we present a high level description which is intended to elucidate and clarify the behavior of the IR system based on the decomposition of the system into modules and processes. The decomposition is in turn based on a detailed study of the functional characteristics of the system [14]. The resultant description is functional in the sense of [15]; i.e., it states exactly what is done by each process in terms of its input, its output, and the transformations performed. The description specifies what is to be done; the details of how to do it are left to the interpretation of the functions.

The functional description of the IR system may be considered a formalization of its requirements definition. It represents a link between the informal (natural language) requirements definition and the formal specification of the system. With reference to Wasserman's three views of a specification (the user, design, and verification views [11]), the functional description represents a unification of the user-design view (as opposed to that of the design-verification view). From the functional description, a formal specification of each system component may subsequently be produced, presuming an appropriate specification technique has been determined. The operational system which results may then be verified (at least to the extent it has been specified) by verification of its parts. Moreover, the functional description, although necessarily detailed and precise, presents a view of system behavior which is much more easily understood than that produced by formal verification techniques.

As a basis for our description, a model of a generalized information retrieval system is proposed. The model consists of six modules or subsystems; the logical processor, selector, descriptor file, locator, document file, and document analyzer. These subsystems function in an environment defined by the user and data blocks, which serve as points of input to the system. Proceeding from a discussion of the functional characteristics of the components, a functional representation of each subsystem and the relationships existing between them is developed. The description allows one to view the system as a collection of abstract processes or activities. Each process is concisely defined via a notation that describes what is accomplished but not how such a process is to be implemented. The semantically irrelevant properties of the description are removed.

The Generalized Model

The generalized model of an information retrieval (IR) system proposed in an earlier work by Crouch [14] is shown in Figure 1. The structural similarity to models proposed by other authors, notably Vickery [16], is acknowledged. This model is viewed as generalized since most operational retrieval systems are functionally compatible with it [17]. Salton's SMART system, with its emphasis on fully automatic content analysis for query and document processing, is the operational system which most closely resembles the model. In developing the representation of the IR system, we concentrate on the functions executed by or within each subsystem as well as the interactions which occur between the modules. A brief description of the model follows.

The user (generally assumed to be unfamiliar with mechanized IR systems or computers) submits a query to the system. The query is taken by the

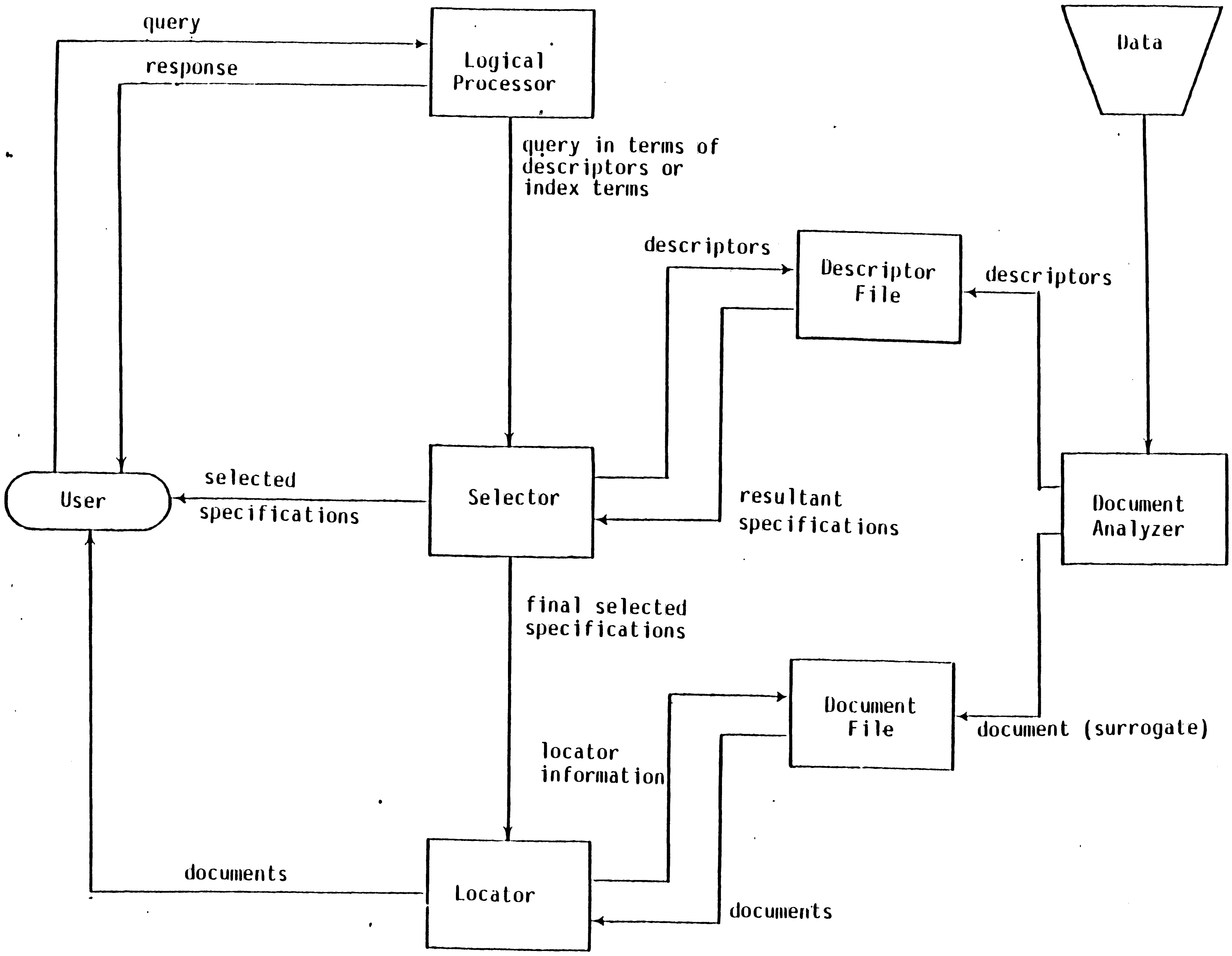


Figure 1. Model of a Generalized Information Retrieval System

logical processor, which operates on the query and outputs to the selector the query in terms of systems vocabulary (i.e., descriptors or index terms). The selector uses the descriptors to search the descriptor file (or index). The resultant specifications, i.e., pointers to those documents which have successfully satisfied the search according to some pre-established criteria, are returned to the selector. The selector in turn operates on these specifications to resolve the query and sends the final selected specifications to the locator, which uses this information to search the document file. The document surrogates themselves are returned via the locator to the user.

The second point of input to the system is represented by the component labeled data. All document-related data enters the system through the document analyzer. The document analyzer operates on the input to produce two outputs--a representation of the document in terms of descriptors, to be stored in the descriptor file along with a pointer to the document in the document file, and a representation of the document itself (i.e., a surrogate) to be stored in the document file.

Note the three feedback loops involving the user:

(1) from the user to the logical processor and back to the user,
(2) from the user to the logical processor and selector, then back to the user, and
(3) from the user to the logical processor, selector, and locator, then back to the user.

In the first case, the logical processor is asking the user to reformulate, clarify, or augment his query. In the second case, the selector is requesting user approval of the selected specifications, i.e. for the user to designate from amongst the set those that most accurately describe his/her needs. The third case represents the normal channel by means of which the document surrogates associated by the system with the user's query are returned to the user.

Note that dictionaries of various types (thesauri, stem and suffix dictionaries, phrase dictionaries, etc.) to the extent that they are used during content analysis must be accessible to both the logical processor and the document analyzer of the model. In the functional description of the IR system, we assume that such dictionaries are available for system use but that the methods by which these dictionaries are constructed are not of primary interest in terms of the overall system behavior.

Description of the Environment

The environment is described by the user and data blocks. Our assumptions about this environment are limited. We consider that the user is motivated by a need for information and interacts with the IR system in an attempt to satisfy this need. Thus he or she supplies a query in natural language form. The query is viewed initially as consisting of a set of individual characters or one-character strings. We designate the input query Y to be the set of all such strings initially utilized to describe the user's need for information:

$$Y::=a_1,a_2,\ldots,a_n=\{a|a\varepsilon A\}$$

where A is the finite symbol set recognizable by the system, i.e., the system alphabet.

Moreover, in conformance with conventional practice in information processing, we consider the query formulation to be based on Boolean logic. Although the advantages offered by the vector processing model over the Boolean model are well known [18,19], the most common query format in operational systems is undoubtedly Boolean.

The second part of the environment, the data block, represents the raw material input to the IR system. This input is assumed to consist of unprocessed textual material in a form convenient to the system. Certain conventions may be followed in compiling this material for input, but no manipulation by trained personnel prior to entry is assumed. Although the form of this data can undoubtedly affect system design, reducing the requirement for automatic content analysis for example, for our purposes this material is considered as a set of recorded symbols recognizable by the document analyzer. This set of recognizable recorded symbols is called a document, T, i.e.,

$$T::=a_1,a_2,\ldots,a_m=\{a|a\varepsilon A\}$$

where each document is composed of a finite number of symbols or characters a, which are members of the symbol set A. It should be noted that although the query and the document are expressed in terms of the same alphabet, there is not necessarily any other commonality; i.e., no terms or words are necessarily in common between the two.

We impose few requirements on the user and data blocks, consequently forcing the IR system to accept an increased responsibility at two points--the logical processsor and document analyzer. These modules necessarily become more complex as a result.

Approach to Functional Description

The symbols used in specifying the functional representation are defined in the three tables in the Appendix. Wherever possible we have attempted to follow the conventions employed in programming language definition or the usual mathematical notation. Since no single set of symbols and no standard terminology are universally accepted, we have taken the liberty of defining our own where required. The operators used in the functional representation are defined in Table A1. Basic definitions used in the development of the representation are given in Table A2. Additional notation is defined in Table A3 and introduced within each section as necessary. The approach taken is to "trace" a query as it is processed by the system, with equal attention paid to each process rather than emphasizing one aspect of system operation at the expense of others.

The Logical Processor (L)

We assume that the query is expressed in natural language form; if desirable, a minor degree of restriction could be imposed [20]. The primary task of the logical processor is to accept the query as input and to produce a normalized query, i.e., the query expressed in terms of systems vocabulary, as output to the selector. Production of this query can be subdivided into the following tasks:

(1) query formation--transformation of the individual, single-character strings or symbols α_i into tokens of the form $[\alpha]_i$ to create a token set Y_F,

(2) query reduction--removal of all grammatical contructs and tokens unrelated to the supposed "information content" of the query, thereby forming a reduced query, Y_D;

(3) query recognition--validation of the query, identifying the (reduced) token set Y_D as a legitimate query. This activity may entail performing a syntactic analysis of Y_D either independently or in dialogue with the user to enable modification according to system requirements. The result is a syntactically valid query, Y_R;

(4) query normalization--enhancement of Y_R by dictionary reference in the process of translating the query terms into [terms consistent with] systems vocabulary. The resulting query Y_N may now be processed by the selector;

(5) pre-search activities--utilization of the query formulation resulting from the three previous tasks to allow user feedback in further query modification.

We can represent the function of the logical processor by beginning with the input query Y, which is initially viewed as a set of characters or symbols comprised of members of the IR system alphabet, i.e.,

$$Y::= \alpha_1,\alpha_2,\ldots,{}_n=\{\alpha|\alpha\varepsilon A\}$$

The alphabet A is the finite symbol set recognizable by the system, $A::=\{\alpha_i, i=1,2,\ldots,\mu(A)\}$. This set can be partitioned into two disjoint subsets, A^T and A^N, where A^T is the subset of terminal symbols (e.g., blank) and A^N the subset of non-terminals. Set A^N can also be partitioned into two disjoint subsets, A^R and A^S. A^R is the set of all alphanumeric characters, whereas A^S is the subset of non-terminals used to indicate the special character symbols which are elements of A.

A. Query Formation (L_F)

The constitution of a valid token is system and/or collection dependent.

(For example, one could imagine circumstances in which the term "C02" is either a valid or an invalid token.) If the allowable query format consists of English text, the individual tokens are usually separated by defined terminators (elements of A^T or A^S). The query is given structure by a defined set of structural elements (for example, parentheses) which is itself a subset of the set of special characters, A^S. We assume that all query processing is performed in a left-to-right order.

The first function of the logical processor (L) is to form a token set. This is accomplished by concatenating consecutive elements in Y of type A^R until either a terminal character (an element of A^T) or a special character (an element of A^S) is encountered. The concatenated string becomes a token $[\alpha]_i$ of the transformed query set Y_F, and the terminators are deleted. The transformations associated with the query formation function L_F can be expressed as follows:

L_F: $\forall \alpha \varepsilon Y$

(1) $\alpha \rightarrow [\alpha]_j, \alpha \varepsilon A^S$

(2) $\perp\{\alpha_i\} \rightarrow [\alpha]_k, \alpha_i \varepsilon A^R$

(3) $\alpha \rightarrow \lambda, \alpha \varepsilon A^T$

The process of query formation, L_F, may now be expressed as:

$$L_F: \quad Y_F = \{[\alpha]_i \mid \alpha \rightarrow [\alpha]_j, \alpha \varepsilon A^S;\ \perp\{\alpha_i\} \rightarrow [\alpha]_k, \alpha_i \varepsilon A^R;\ \alpha \rightarrow \lambda, \alpha \varepsilon A^T; \alpha \varepsilon Y\}$$

That is, the query formation function L_F, acting upon its input Y, produces a set of tokens Y_F. Y_F consists of all tokens of the form $[\alpha]$, where the $[\alpha]_i$ are produced via the specified transformations and $\alpha \varepsilon Y$. For simplicity, let $y ::= [\alpha]_i$. Then $Y_F = \{[\alpha]_i\} = \{y\}$ and logical processor activity in query formation may be expressed as $L_F(Y) = Y_F$. Thus query formation is analogous to the lexical analysis phase of a compiler in that in each case, statements are scanned to produce tokens.

B. Query Reduction (L_D)

Consider A^S, the set of special characters. A^S can be partitioned into two disjoint subsets, A^U and A^V, where A^U represents the set of all characters which are used to denote the structure of the query (i.e., "structural elements" such as parentheses in the Boolean query) and A^V is the set of all non-structural elements (e.g., the usual symbols of punctuation used in English text).

The query reduction phase of logical processor activity "reduces" the input query Y_F by removing those tokens unrelated to the "information content" of the query. In a Boolean environment, reduction implies the removal of all non-structural special character tokens (elements of A^V). The resultant token set consists of alphanumeric tokens and structural elements (single-character tokens such as parentheses). It also includes the logical connectives "and" and "or", which are used to express the relation between terms in Y_F. These terms are by convention represented by the corresponding symbols "$\wedge$" and "$\vee$", respectively. For this reason, as well as to simplify subsequent processing, transformations associated with query reduction may be specified as follows:

L_D: $\forall y \varepsilon Y_F$

(1) $y \leftarrow \lambda, y \varepsilon A^V$

(2) $y \leftarrow [\wedge], y = [and]$

(3) $y \leftarrow [\vee], y = [or]$

Thus the query reduction function L_D operating on the reduced Boolean query Y_F may be defined as:

$$L_D: \quad Y_D = \{y \mid y \leftarrow \lambda, y \varepsilon A^V;\ y \leftarrow [\wedge], y = [and];\ y \leftarrow [\vee], y = [or];\ y \varepsilon Y_F\}$$

Thus $L_D(Y_F) = Y_D$, and Y_D now consists of query terms (tokens which are possible descriptors or index terms), structural elements, and logical operators.

C. Query Recognition (L_R)

During the query recognition phase, denoted by L_R, the logical processor acts either to accept the query as is (i.e., to validate the query) if it is recognizable by the system or to reject the query in the case of incomplete syntax. Thus

$$L_R: \quad Y_R = \begin{cases} Y_D & \text{if } Y_D \text{ is a valid query} \\ \emptyset & \text{if } Y_D \text{ is an invalid query} \end{cases}$$

In addition, the system could allow interaction with the user in an effort to produce a revised query if Y_R cannot be validated at this point. This process may be described as follows. If $L_R(Y_D)=\emptyset$, Y_D is returned to the user. The user can then submit a query Y', which may be either a totally new query or a modified version of Y_D. In either case,

$$L_R(L_D(L_F(Y'))) = \begin{cases} Y_R & \text{or} \\ \emptyset \end{cases}$$

and the process is continued until $L_R(Y_D)=Y_R$, $Y_R \neq \emptyset$. Thus query recognition is analogous to the syntactic analysis phase of the compiler in that as a result of its activity, its input is structurally or syntactically validated.

Of particular interest at this point is the process of query recognition for a Boolean query. Let U represent the set of query terms which are elements of Y_D. Then U may be defined as:

$$U::=\{y|y \notin A^U, y \notin o;\ y \varepsilon Y_D\}, \text{ where } U \subseteq Y_D \text{ and } o::=\{[\wedge],[v]\}.$$

Then B, the set of all (syntactically correct) Boolean queries associated with Y_D (i.e., whose terms are query terms of Y_D), may be defined as

$$B::=\{b\}, \text{ where } b::=u|[(]b[)]|bob.$$

If $Y_D \varepsilon B$, then $L_R(Y_D)=Y_R$, $Y_R \neq \emptyset$, and Y_R is recognizable. Otherwise, $L_R(Y_D)=\emptyset$ and a syntactic analysis of the reduced query Y_D fails. Assuming the output of query reduction is non-null, the next task of the logical processor is query normalization.

D. Query Normalization (L_N)

The query Y_R which is input to the query normalization phase of logical processor activity has now been validated syntactically. It consists of a token set composed of query terms, logical operators, and structural elements (i.e., parentheses). Yet for this query to be processed by the system, it must be in terms of the system vocabulary, i.e., each query term must be recognizable by the system. Thus the tasks of query normalization consist of (1) query term validation and (2) the transformation of each valid query term into its appropriate representation in terms of descriptors or index terms.

Consider U, the set of query terms which are elements of Y_R. If $L_R(Y_D)=Y_R$, $Y_R \neq \emptyset$, then $Y_R=Y_D$ and $U=\{y \mid y \notin A^U, y \neq o;\ y \varepsilon Y_R\}$, where $U \subseteq Y_R$. Then in the general case, the transformations associated with the query normalization process L_N may be defined quite simply as

L_N: $\forall u \varepsilon Y_R$

(1) $u \rightarrow d$, for $u=d, d \varepsilon \triangle$

(2) $u \rightarrow \lambda$, for $u \neq d, \forall d \varepsilon \triangle$

Query normalization is the activity during which thesauri and/or phrase dictionaries might be employed in the attempt to normalize the query, i.e., to align the query (expressed in terms of the user's vocabulary) with the system's representation of the data. How may the composition of $\triangle$, the set of all descriptors, be determined? Consider the frequency distribution of terms contained in the documents stored in the document file of the generalized model.

Many experiments have dealt with the task of recognizing or choosing appropriate descriptors for a document collection. It has been shown that the best terms are medium frequency terms with positive discrimination values; these terms may be used directly as descriptors (i.e., this criterion may be used to determine the composition of $\triangle$, the universal set of descriptors) [21]. The question arises as to whether to include in the descriptor set narrow, low frequency terms with near-zero discrimination values and broad, high-frequency terms with negative discrimination values. Omitting the former may result in a loss of precision, whereas omitting the latter may reduce recall. But transformations may be applied to these terms which improve their discriminative properties. Specifically, low frequency terms can be "expanded" by dictionary (thesaurus) reference and the set of high frequency terms can be "contracted" via the use of phrase dictionaries [18].

Consider the case of query expansion. For certain low frequency terms in certain environments, it may be desirable to augment or replace these terms with an appropriate thesaurus class. Logically, the query is expanded as follows.

Let U' represent the set of low frequency query terms which meet the specified criteria for expansion. Then for each $u \varepsilon U'$, $U' \subseteq U$, there exists a set $N(u)$ (thesaurus class) such that $N(u) \subseteq \triangle$. With reference to low frequency terms with near-zero discrimination values, the query normalization function L_N may be expressed as:

L_N: $\forall u \varepsilon U'$

(1) $u \mapsto \rho[\perp\{[(],\vee N(u),[)]\}]$

where the decomposition function ρ breaks the concatenated token of descriptors, logical operators, and structural elements into separate tokens. Each $u \varepsilon U'$ has now been replaced by the corresponding set of thesaurus class entries and the structure of the (Boolean) query has been maintained. In practice, the thesaurus class identifiers may replace each particular query term u, where $u \varepsilon U'$, in Y_R. Logically, however, Y_R has been expanded by the replacement of a single query term u, $u \varepsilon U'$, by the set of descriptors which are considered "synonymous" with it.*

Consider now the utilization of certain high frequency terms as descriptors. It may be decided to discard such terms on the basis of their lack of discriminative power. If not, such terms may be combined via dictionary reference into appropriate phrases of lower frequency with higher discrimination values.

Let U" represent the set of high frequency query terms which meet the criteria for normalization via phrase dictionary reference, where $U'' \subseteq U \subseteq Y_R$. Since Y_R is a syntactically valid Boolean query, all query terms consist of individual tokens, which are related via logical operators, and (possibly) structural elements (parentheses). These structural elements may be supplied by the user when the query is input to indicate that operations within parentheses take precedence over other operations when the query is processed (i.e., executed) by the selector. (Note that the order of

---

*The use of this method in dealing with low frequency terms in a Boolean environment is largely dependent upon the thesaurus. If a high degree of synonymy is exhibited between thesaurus class terms, the results of query expansion by this method should be as desired. Otherwise an unacceptable loss of precision can occur. A more viable approach might be to maintain the low frequency term in the indexing vocabulary and to expand only when an initial query involving that term fails to produce an acceptable result. More research is needed to establish a methodology for operating in this environment.

processing and the precedence of the logical operators is uniquely defined for each system.) To simplify the task when dealing with high frequency query terms, we assume that the grouping of any two query terms u_i and u_j expressed in the form "$u_i \wedge u_j$" indicates that these two terms are candidates for normalization via phrase dictionary reference, where one of the terms, u_i or u_j, is an element of U". (In practice these requirements could be modified; e.g., the number of terms related via the "$\wedge$" operator could be extended, etc. It is clear, however, that only first order conjunctive relationships should be considered.)

Consider the set of high frequency query terms U" associated with Y_R. Then there exists a set M, $M \subseteq \triangle$, all of whose members consist of tokens of the form $\iota\{m_i,m_j\}=[m_i m_j]$. Then M serves as a phrase dictionary whose entries are composed of individual tokens which by virtue of their co-occurrence have a specific meaning aside from that associated with each token individually. Hence each element of M itself serves as a unique descriptor d, an element of the set $\triangle$. This transformation may be defined as follows:

L_N: $\forall u_i \varepsilon U" \ni \iota\{u_i,u_j\}\varepsilon M$

(2) $\iota\{u_i,[\wedge],u_j\} \vdash d$

Subsequent transformations, accounting respectively for the medium frequency terms with positive discrimination values (elements of $\triangle$) and those terms meeting none of the aforementioned criteria, may then be expressed as

L_N: $\forall u \ni u=d, d\varepsilon\triangle$

(3) $u \leftarrow d$

and

L_N: $\forall u \ni u \neq d, \forall d \varepsilon \triangle$

(4) $u \leftarrow \lambda$

Thus in a generalized system in which document terms in all frequency ranges are recognized, the query normalization phase of logical processor activity may be expressed as follows:

L_N: $Y_N=\{y|u \rightarrow \rho[\iota\{[(],vN(u),[)]\}],u\varepsilon U'; \iota\{u_i,[\wedge],u_j\} \vdash d,$
$\forall u_i \varepsilon U" \ni \iota\{u_i,u_j\}\varepsilon M; u \leftarrow d, u=d, d\varepsilon\triangle; u \leftarrow \lambda, u \neq d, \forall d \varepsilon \triangle; y\varepsilon Y_R\}$

Thus $L_N(Y_R)=Y_N$ and the normalized query Y_N, expressed in terms of systems vocabulary (i.e., descriptors, logical connectives, and structural elements), is passed to the selector.

E. Pre-search Activities (L_P)

The function of the logical processor in presearch activities would involve output from the previous phases of reduction, recognition, and normalization. At this point the logical processor acts to return any information that might be of interest (i.e., Y_D, Y_R, and/or Y_N) to the user. For the Boolean query, this activity can be expressed as:

$$L_P: \ \text{User} \leftarrow \{y | y \varepsilon Y_N\} = Y_N.$$

One can visualize the function of the logical processor (L) to be defined in terms of the individual tasks as

$$L ::= \langle L_N(L_R(L_D(L_F(.)))),\ L_P(.)\rangle$$

where the angular brackets enclose the outputs produced by the module.

The Selector (S)

The selector (S), using the processed form of the query Y_N as input, retrieves from the descriptor file the set of all document identifiers associated with each descriptor $d \varepsilon Y_N$. The indicated logical operations are then performed in the order specified (with precedence of operations modified by parentheses). The result is a set of specifications, i.e., the set of all document identifiers associated with the initial query Y. In addition, selected specifications, e.g. a document file entry, the number of documents associated with each descriptor, etc., may be returned to the user. This may be termed post-search activity.

A. Document Selection (S_S)

In representing the function of the selector, we must consider the relationship between this module and the descriptor file. We represent the descriptor file as a passive entity acted upon by the selector and the document analyzer. Let T(d) denote the set of all documents associated with descriptor d, where $T(d) \subseteq \mathbf{D}$, the set of all documents. Beginning with the first descriptor $d \varepsilon Y_N$, all documents associated with this descriptor (T(d)) are identified and formed into a single token by application of the concatenation operator ($\perp$). The decomposition operator (ρ) breaks the token into components, where a component is either a document identifier or a parenthesis. This set of tokens replaces d in the query Y_N, and the sequence of operations is repeated beginning with the next descriptor d; i.e.,

S_S: $\forall y \varepsilon Y_N$

(1) $y \leftarrow \rho[\perp\{[(],T(d),[)]\}],y=d$

Finally the logical operators ($[\wedge]$,$[\vee]$) are replaced by the set union and intersection operators, ($[\cap]$,$[\cup]$) respectively, and the expression is evaluated to produce the set of document identifiers (loosely, the set of documents) $\overline{D}$ associated with (or relevant to) the original query Y. This operation of the selector may be expressed as

S_S: $\forall y \varepsilon Y_N$

(2) $y \leftarrow [\cap],y=[\wedge]$

(3) $y \leftarrow [\cup],y=[\vee]$

Thus S_S, the action of the selector in document selection, may be defined as

$$S_S: \; Y_S=\{y|y \leftarrow \rho[\perp\{[(],T(d),[)]\}],y=d;\; y \leftarrow [\cap],y=[\wedge];\; y \leftarrow [\cup],y=[\vee];\; y \varepsilon Y_N\}$$

Applying Ω, the operator which evaluates any valid set expression, to Y_S yields $\overline{D}$, the set of documents retrieved by the original query Y: $Y_S=\{D|D\varepsilon\overline{D}\}=\overline{D}$. Hence selector activity can be represented as $S_S(Y_N)=Y_S$, where $\Omega Y_S=\overline{D}$, and the original query Y has now been resolved.

B. Post-search (S_P)

The second function (sometimes called post-search activity) of the selector is to operate on the document set $\overline{D}$ in a manner so as to return some aspect of $\overline{D}$ to the user via the route labeled "selected specifications." The information returned is system dependent. In one case the number of elements in $\overline{D}$ ($\mu(\overline{D})$) might be sufficient while in another the number of documents associated with a particular descriptor ($T(d)$, where $d \varepsilon Y_N$) might be provided. We represent the activity of the selector in post-search as follows:

S_P: User$\Leftarrow\mu(\overline{D})$

Representing both functions of the selector requires the application of the document selection function followed by post-search activity, i.e.

$S::=<S_S(.), S_P(.)>$.

The Descriptor File

Representation of the descriptor file begins with the assumption that the main concern in our generalized retrieval system is single query processing (i.e., the query of the individual user), rather than the batch processing of multiple queries. As previously noted, the descriptor file is viewed as a passive entity, acted upon by the selector and the document analyzer. We characterize it by representing its organization rather than prescribing any active functions performed by it.

We consider the system vocabulary to be changing (increasing over time) and largely determined by the criteria invoked by the document analyzer. The essential task is to represent the process by which the set $T(d)$ is defined with reference to the organization of the descriptor file. Although virtually all operational systems utilize inverted files, we consider three principal file organizations, namely serial, inverted, and multilist.

1. The Serial File

A typical serial file entry is seen as follows:

D	d_i	d_j	...	d_k	...	d_m

Associated with each document D is a set R(D) of descriptors d_i. Recall that T(d) is the set of all documents associated with descriptor d. Let Q(D) represent the set of all descriptors associated with (contained in) document D. The serial file may then be characterized by:

(1) $T(d)::=\{d_D:R(D),\forall D\varepsilon\mathbf{D}\}$

(2) $Q(D)::=R(D)$

Thus T(d) is found by the following process. First d is compared to every element of the set R(D). If d is an element of R(D), D (the associated document identifier) is returned. The comparison is made for all D contained in the set **D.**

2. The Inverted File

A typical inverted file entry is reproduced below.

d	D_i	D_j	...	D_k	...	D_m

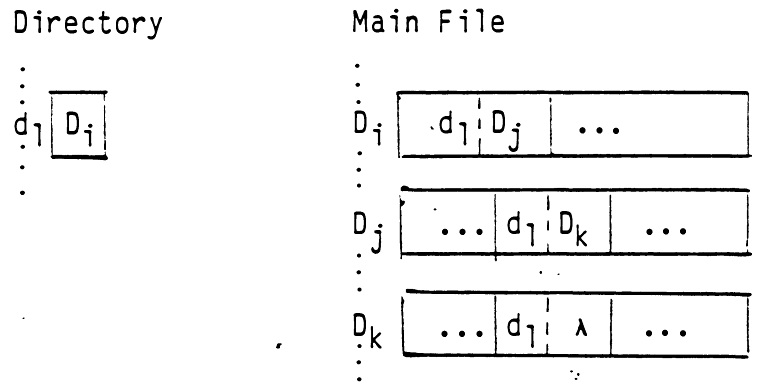
That is, associated with every descriptor d is a set R(d) of documents D_i. Inverted file organization can be represented by:

(1) $T(d)::=R(d)$

(2) $Q(D)::=\{D_d:R(d),\forall d\varepsilon\Delta\}$

3. The Multilist File

Multilist file organization is somewhat more complex than the others, since it involves the use of an additional file, frequently called the directory. The directory is ordered by key (i.e., descriptor). Each key has associated with it a pointer to a corresponding main file entry, which represents that document in the system with which the key (descriptor) is first associated. For each descriptor contained in the document, there is a pointer to the "next" document which contains that descriptor. This organization permits the use of a reduced directory, which gives access to a large main file in which the entries (documents) under each key or descriptor are not individually accessible but are linked. (For a discussion of multilist organization, see [22].) Multilist file organization is pictured below.



All main file entries are of the form (d_i, D_k) where $v(d_i)=D_k$. The diagram shows the directory entry associated with some specified descriptor d_1 and the corresponding main file entries. Thus multilist file organization can be characterized by:

(1) $T(d)::=\{d_{v(d)}:R(R(d))/v(d)=\lambda\} \cup R(d)$

(2) $Q(D)::=\{D_d:T(d),\forall d\varepsilon\triangle\}$

In each search of the descriptor file by the selector, the object of the search is the set $T(d)$.

The Locator (R)

Just as the selector searches the descriptor file in order to extract the documents or document identifiers associated with each descriptor in the normalized query, the locator (R) searches the document file to extract the record associated with each document in the set $(\bar{D})$ passed to it by the selector. This record may consist of the document title and related bibliographic information, an abstract, or an extract. In any case, the contents of the document file entry associated with the document are returned to the user under the heading of "located documents." We represent the function of the locator as simply:

R: $\text{User} \Leftarrow \{R(D) \mid D\varepsilon\bar{D}\}$

where R(D) is the entire data record (entry) associated with document D in the document file.

The Document File

The document file is composed of a set of entries of the form R(D), D ε **D**, which are the system's representation of the corresponding documents. The information contained within R(D) is determined by the criteria applied by the document analyzer. We assume that in every case a unique document identifier D is associated with the document record R(D). The document analyzer may leave the document (data) input virtually intact, operating only to construct the corresponding document representation for the descriptor file. Consequently, the document as originally input may be placed unaltered in the document file.

The document file, like the descriptor file, is considered a passive entity. Similarly, the representation involves defining the file organization which is assumed to be based simply on document identifier D (i.e., accession number) or perhaps on an ordering based on frequency of use. In either case, document file organization is represented simply as R(D), where Dε**D**.

The Document Analyzer (A)

The document analyzer constitutes the second port of entry for input external to the ISR system (the other being the logical processor). The function of the document analyzer is to process the incoming data in order to produce two outputs: (1) some indication of the content of the incoming document, to be stored in the descriptor file along with a pointer to the document in the document file, and (2) a representation of the document itself (i.e., the system's representation of the document), to be stored in the document file. Obtaining the description of document content is commonly called indexing.

The importance of the indexing task has been noted by numerous authors.

Automatic indexing techniques fall into four general categories: permutation indexing, citation indexing, statistical indexing, and semantic indexing procedures. While the application of the techniques in each category requires quite different assumptions and utilizes different aspects of the data, all operate on the data with the same objective: to construct a set of descriptors that "...somehow indicate the information content of the document... ." [23]

The second major task of the document analyzer is the construction and the storage of a document representation in the document file. This representation would include a document identifier, usually all the elements of a bibliographic reference (author, title, publisher, etc.), and might include citations, an abstract, an extract, or conceivably the complete document text.

The indexing task may be quite complex, whereas determining the document representation may be almost perfunctory. Considering the indexing function of the document analyzer, Vickery [16] recognizes three stages in the assignment of document descriptors;

(1) scan of the text to derive those words, phrases, and/or sentences which best represent information content;

(2) a decision as to which of the descriptors are worthy of being recorded in the descriptor file, in view of the purpose of the system, and

(3) the transformation of the selected descriptors into a standard "descriptor language," the resulting terms of which serve as the entry or entries in the descriptor file.

We describe these three processes by three functions, i.e., the document term formation function (A_F), the document term reduction function (A_D), and the descriptor determination function (A_p).

A. Document Term Formation (A_F)

Recall that the textual entry T associated with document D is defined as a set of characters or symbols, i.e.,

$T::=a_1,a_2,\ldots,a_m=\{a|a\varepsilon A\}$

Then the task of the document analyzer (A) in processing the document text is basically identical to the task of the logical processor in handling the query. Using the sets A^S, A^R, and A^T (as previously defined), we apply the document term formation function A_F to T as follows (where normal left-to-right processing is assumed):

A_F: $\forall \alpha \varepsilon T$

(1) $\alpha \leftarrow [\alpha]_j, \alpha \varepsilon A^S$

(2) $\perp\{\alpha_i\} \leftarrow [\alpha]_k, \alpha_i \varepsilon A^R$

(3) $\alpha \leftarrow \lambda, \alpha \varepsilon A^T$

Thus the function of the document analyzer in document term formation (A_F) may be expressed as

$$A_F: \quad T_F=\{[\alpha]_i | \alpha \leftarrow [\alpha]_j, \alpha \varepsilon A^S; \ \perp\{\alpha_i\} \leftarrow [\alpha]_k, \alpha_i \varepsilon A^R; \ \alpha \leftarrow \lambda, \alpha \varepsilon A^T; \ \alpha \varepsilon T\}$$

Then $A_F(T)=T_F$, and the document text T_F now consists of the token set $\{[\alpha]_i\}$. Let $t=[\alpha]_i$. Then $T_F=\{[\alpha]_i\}=\{t\}$.

B. Document Term Reduction (A_D)

Let $G::=\{[g]\}$, the set of all nonsubstantive or "stop list" terms, each of which is a token. This is the set of tokens which are non-meaningful (which do not add to the "information content" of the text being processed) in the context of this system. The token set A_F can now be reduced by removing stop list terms (elements of G) and various special character tokens (elements of A^S). Thus

A_D: $\forall t \varepsilon T_F$

(1) $t \leftarrow \lambda, t \varepsilon G$

(2) $t \leftarrow \lambda, t \varepsilon A^S$, and

$$A_D: \quad T_D=\{t | t \leftarrow \lambda, t \varepsilon G; \ t \leftarrow \lambda, t \varepsilon A^S; \ t \varepsilon T_F\}$$

Then $A_D(T_F)=T_D$ and all the tokens of the reduced token set T_D are possible descriptor terms. Note that in reducing the set of document tokens, one need not normally be concerned with special tokens (such as parentheses and logical connectives) which characterize the Boolean query. Nor is there an associated recognition phase, for the document text consists of unstructured natural language. However, the document text, like the query, must now be normalized (expressed in terms of the system vocabulary). Thus each document term (token) must be validated and each valid document term transformed into its appropriate representation in terms of descriptors.

C. Descriptor Determination (A_P)

The basic operation of the document analyzer is A_P, the descriptor determination function. This function operates on the set T_D to form Q(D), the set of all descriptors associated with document D. These descriptors are then inserted in the descriptor file via the appropriate file maintenance function.

The criteria used in the selection of descriptors (by A_P) is system dependent since a number of alternative indexing techniques could be used. The basic function, however, can be expressed as:

A_P: $\forall t \varepsilon T_D$

(1) $t \leftarrow d$, for $t=d$, $d \varepsilon \triangle$

(2) $t \leftarrow \lambda$, for $t \neq d$, $\forall d \varepsilon \triangle$

If a fixed descriptor (controlled) vocabulary is used, the determination of whether a particular document token t is a descriptor is easily made. If a controlled vocabulary is not used, other criteria may be applied to limit vocabulary (descriptor) growth based on the frequency distribution of document terms. Specifically, as discussed previously, low frequency terms with near-zero discrimination values may be normalized via dictionary (thesaurus) reference, and high frequency terms with negative discrimination values can be normalized via the use of phrase dictionaries. The medium frequency terms may be used directly as descriptors [18].

Let T' represent the set of low frequency document terms which meet the specified criteria for expansion, where $T' \subseteq T_D$. Then for each $t \varepsilon T'$, there exists a set N(t) (i.e., thesaurus class) such that $N(t) \subseteq \triangle$. Then

A_P: $\forall t \varepsilon T'$

(1) $t \leftarrow N(t)$

Logically the set T_D has been expanded by the replacement of the individual document terms t by the corresponding set of thesaurus class terms, N(t). In practice, each such t may be replaced by a single descriptor (the thesaurus class identifier) associated with N(t).

Similarly, consider the set of high frequency document terms T" associated with T_D which meet the criteria for reduction, where $T''\subseteq T_D$. Then there exists a set M, $M\subseteq\triangle$, all of whose members are word phrases of the form $[m_i,m_j]$, where each element of M is itself a unique descriptor, d. Descriptor determination in dealing with high frequency terms may be expressed as

A_P: $\forall t\varepsilon T'' \ni \perp\{t_i,t_j\}\varepsilon M$

(2) $\{t_i,t_j\}\vdash d, d\varepsilon\triangle$

Subsequent transformations may be defined as

A_P: $\forall t \ni t=d, d\varepsilon\triangle$

(3) $t\leftarrow d$

A_P: $\forall t \ni t\neq d, \forall d\varepsilon\triangle$

(4) $t\leftarrow\lambda$

Thus in a generalized system in which document terms of all frequency ranges are recognized, the descriptor determination function A_P of document analyzer activity may be expressed as:

$$A_P: \; T_P=\{t\,|\,t\leftarrow N(t), t\varepsilon T';\; \{t_i,t_j\}\vdash d, d\varepsilon\triangle, \forall t\varepsilon T'' \ni \perp\{t_i,t_j\}\varepsilon M;\; t\leftarrow d, t=d, d\varepsilon\triangle;\; t\leftarrow\lambda, t\neq d, \forall d\varepsilon\triangle;\; t\varepsilon T_D\}$$

Then $Q(D)=T_P$ and $A_P(T_D)=Q(D)$.

D. Descriptor File Maintenance (A_M)

Two additional functions remain to be accomplished by the document analyzer, and these relate to the file maintenance requirements (i.e., the addition of a new document D to the document file, with corresponding changes in the descriptor file.) For the descriptor file the tasks required differ according to the file organization employed. We denote the maintenance function required for the descriptor file by A_M and represent the activities as follows:

1. The Serial File

   A_M: $R(D)=\{d\,|\,d\varepsilon Q(D)\}$

2. The Inverted File

A_M: $R(d)=\{D \cup R(d), \forall d \varepsilon Q(D)\}$

3. The Multilist File

A_M: $R(D)=\{d | v(d)=\lambda, \forall d \varepsilon Q(D)\}$

$R(d)=D, \forall d_\ni d \varepsilon \{Q(D) \cap \Delta^C\}$

$$\left.\begin{array}{l} d_{v(d)}: R(R(d))/v(d)=\lambda \\ v(d)=D \end{array}\right\} \forall d_\ni d \varepsilon \{Q(D) \cap \Delta\}$$

In each case the file maintenance function begins with the set Q(D) produced by A_D. Note that the capability for increasing the descriptor set is shown explicitly for the multilist organization.

The serial and inverted file maintenance functions are simple. In the serial file the descriptor set is assigned to a document record, while the inverted file requires the addition of a document identifier to the set of document identifiers referenced by each descriptor d. For the multilist file, the first operation refers to the formation of the main file record, the second describes the formation of a new directory record, and the third describes the setting of the main file link. In all cases the universal set of descriptors Δ is considered to be dynamic, i.e., increasing over time. After each new document enters the system and the corresponding modifications have been made in the descriptor file, Δ changes as follows: $\Delta = \Delta \cup \{d | d \varepsilon Q(D)\}$ or $\Delta = \Delta \cup Q(D)$.

E. Document File Maintenance (A_N)

Along with determining the descriptor set Q(D) and using it appropriately in file maintenance functions, the document analyzer operates on the original text input T to construct and/or maintain the document file. This function (A_N) involves only the construction of the document record R(D) and the addition of the document D to the set of all documents **D**. The function of the document analyzer in document file maintenance may be defined very simply as

A_N: $R(D)= \alpha_1, \alpha_2, \ldots, \alpha_m = \{\alpha | \alpha \varepsilon A\} = T$

as defined originally and $\mathbf{D} = D \cup \mathbf{D}$.

In summary, the function of the document analyzer (A) can be represented as

$$A::=<A_M(A_P(A_D(A_F(.)))),\ A_N(.)>$$

where the angular brackets enclose the two outputs of document analyzer activity.

Summary

In this paper, the authors present a high level, functional approach to the description of a generalized information retrieval system. This description is based on a functional decomposition of the system into modules and processes and on an appropriate data abstraction.

Using a small set of operators or primitives, we provide a description of all the processes of a generalized IR system, from submission of the query to the receipt of the final selected documents by the user. By this means, a comprehensive overview of the system, as well as a thorough description of its behavior is produced in terms of the component processes and the interactions of these processes in terms of inputs, outputs, and the associated transformations. This nonprocedural description provides a necessary foundation for subsequent formal system specification.

The authors believe that the effort expended in organizing and developing a comprehensive, unified view of large, complex systems is well worthwhile in terms of the potential it offers for improved understanding of the system and its behavior. Moreover, such a description focuses attention on the system in its earliest stages of development, when many crucial decisions are made. Erroneous decisions made at this point can prove to be most costly in the final realization of the system.

Acknowledgements

The authors wish to express their appreciation to Gerard Salton, Dina Bitton, and Don Crouch for their suggestions and constructive criticism and to Fred Schneider and Abba Moitra for their most helpful comments on specification.

References

[1] Booch, G. Software Engineering with Ada. Benjamin/Cummings, New York, 1983.

[2] Liskov, B. and Zilles, S. Specification Techniques for Data Abstractions. IEEE Trans. Software Eng., Vol. SE-1, No. 1, March 1975, 7-19.

[3] Berg, H. K. et al. Formal Methods of Program Verification and Specification. Prentice-Hall, Inc., Englewood Cliffs, N. J. (1982).

[4] Bjorner, D. and Jones, C. Formal Specification & Software Development. Prentice-Hall International, Inc., Englewood Cliffs, N. J. (1982).

[5] Zave, P. The Operational Versus the Conventional Approach to Software Development. Comm. of the ACM, Vol. 27, No. 2, Feb. 1984, 104-118.

[6] Wegner, P. Capital Intensive Software Technology, Part 2: Programming in the Large. IEEE Software, Vol. 1, No. 3, July 1984, 24-31.

[7] Balzer, R., Cheatham, T., and Green, C. Software Technology in the 1990's: Using a New Paradigm. Computer, Vol. 16, No. 11, Nov. 1983, 39-45.

[8] Winograd, T. Beyond Programming Languages. Comm. of the ACM, Vol. 22, No. 7, July 1979, 391-401.

[9] Ambler, A. et al. Gypsy: a Language for Specification and Implementation of Verifiable Programs. Proc. of ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices, Vol. 12, No. 3, March 1977, 1-10.

[10] Biggerstaff, T. The Unified Design Specification System (UDS2), IEEE Proc. Specifications of Reliable Software, 1979, 104-118.

[11] Wasserman, A. Information System Design Methodology. Journal of the ASIS, Vol. 31, No. 1, Jan. 1980, 5-24.

[12] Wasserman, A. and Stinson, S. A Specification Method for Interactive Information Systems, IEEE Proc. Specifications of Reliable Software, 1979, 68-79.

[13] Leveson, N. G., Wasserman, A. I., and Berry, D. M. Basis: A Behavioral Approach to the Specification of Information Systems. Information Systems, Vol. 8, No. 1, 15-23.

[14] Crouch, C. J. Language Relations in a Generalized Information Storage and Retrieval System. Doctoral Thesis, Southern Methodist University, 1971.

[15] House, R. Comments on Program Specification and Testing. Comm. of the ACM, Vol. 23, No. 6, June 1980, 324-331.

[16] Vickery, B. C. On Retrieval System Theory. Butterworths, London (1965).

[17] Crouch, C. J. and Crouch, D. B. An Analysis of Document Retrieval Systems Using a Generalized Model. In: Information Systems, Vol. 3, Julius Tou, (Ed.), Plenum Publishing Co., New York, 1975, 219-237.

[18] Salton, G. and McGill, M. G. Introduction to Modern Information Retrieval. Mc-Graw Hill, New York (1983).

[19] Salton, G. The Use of Extended Boolean Logic in Information Retrieval. Technical Report TR 84-588, Department of Computer Science, Cornell University, Jan. 1984.

[20] Rubinoff, M. et al. Experimental Evaluation of Information Retrieval Through a Teletypewriter. Comm. of the ACM, Vol. 9, No. 9, Sept. 1968, 67-71.

[21] Salton, G., Yang, C. S., and Yu, C. T. A Theory of Term Importance in Automatic Text Analysis. Journal of the ASIS, Vol. 26, No. 1, Jan. 1975, 33-44.

[22] Salton, G. Automatic Information Organization and Retrieval. Mc-Graw Hill, New York, 1968.

[23] Bar-Hillel, Y. Language and Information. Addison Wesley, Reading, Mass., 1964.

Appendix

Operator	Description/Definition	Use
:	comparison	compares element on left side of operator to every element of the set on the right side of the operator
(,)	parentheses	alters the usual left-to-right execution of Boolean expression by giving higher priority to operations to be performed within innermost nested parentheses
$\cup$, $\cap$	union, intersection	set operators
Ω	set evaluation	operates on any valid set expression to resolve the expression
o	logical ($\wedge$,$\vee$)	o denotes either member of the set of logical operators (with the negation operator omitted for the sake of simplicity). The operators have an established priority, modified by the presence of parentheses.
oX	$oX::=[x_1 o x_2 \ldots o x_{\mu(X)}]$	token formation. The application of the operator o to the set X to form a token (where square brackets denote that the contents of the bracket is considered a single token and $\mu(X)$ denotes the number of elements in the set X).
$\rho[x]$	$\rho[x_1 x_2 \ldots x_n]::=\{[x]_1,[x]_2,\ldots,[x]_n\}$	token decomposition. The application of the decomposition operator ρ to the specified token [x] to form a set of tokens which together compose [x].
$\perp X$	$\perp X::=[x_1 x_2 \ldots x_{\mu(X)}]$	set concatenation. The application of the concatenation operator $\perp$ to the set X, concatenating consecutive elements of X to form a single token.

Table A1. Operators Used in the Functional Description

Notation	Description
D	a document (more specifically the identifier associated with a document, which represents the document within the system).
d	a descriptor (token)
D	set of all documents in the system
$\bar{D}$	set of all documents associated with (retrieved by) a specific query
$\triangle$	set of all descriptors in the system
R(x)	(1) contents of record r corresponding to record identifier x, or (2) a set of items associated with identifier x, or (3) a mapping which associates with an identifier x a set of items R(x)
Q(D)	set of all descriptors associated with document D
T(d)	set of all documents (document identifiers) associated with descriptor d
G::={[g]}	set of all nonsubstantive words or "stop list" terms, all of which are tokens ($G \cap \triangle = \emptyset$).
$A::=\{\alpha_i, i=1,2,\ldots,\mu(A)\}$	the set of unique symbols recognizable by the system, the system alphabet
$Y::=\alpha_1,\alpha_2,\ldots,\alpha_n=\{\alpha \mid \alpha \varepsilon A\}$	the input query, an ordered set whose elements are members of A
$T::=\alpha_1,\alpha_2,\ldots,\alpha_m=\{\alpha \mid \alpha \varepsilon A\}$	the document (surrogate), an ordered set whose elements are members of A
$x_t:f(z)$	t if $x \varepsilon f(z)$ Ø otherwise Note: t is the value returned
f(x)/y=z	$r \leftarrow x$ $r \leftarrow f(r)$ false y=z true return r
{f(x)/y=z}	$r \leftarrow x$ $r \leftarrow f(r)$ return r false y=z true

Table A2. Basic Definitions in the Functional Description

Notation	Description
::=	definition
=	equivalence
{a|condition(s)}	set of all a, for which condition/conditions holds/hold
{ }	any set
$\emptyset$	null set
ε	is an element of
$\Leftarrow$	return (to the user) of set on the right
$\leftarrow$	replacement of token (set) on the left by the token (set) on the right
o	a Boolean operator ($\wedge$,$\vee$)
F:	names a function F which is defined via the notation to the rignt
$\forall$	for all
$\ni$	such that
λ	token representing the null field ($\lambda \equiv [\lambda]$)
Y^C	the complement of set Y ($Y^C ::= \{y \mid y \notin Y\}$)
$\mu(A)$	the number of elements in the set A
α	any element of the system alphabet A
$[\alpha]$	any token
A^T	set of all terminal symbols
A^N	set of all non-terminals
A^S	set of all special characters
A^R	set of all alphanumeric characters
A^U	set of all structural elements
A^V	set of common punctuation symbols

Table A3. Notation Used in the Functional Description