An Overview of the Architecture for MicroArras 1.0

TR88-042

September 1988

Scott Southard
John B. Smith
Stephen F. Weiss
Gordon J. Ferguson

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A TextLab Report
This research was also supported by the National Endowment for the Humanities, Grant #RT-20483-84 and by Northern Telecom, Inc.
UNC is an Equal Opportunity/Affirmative Action Institution.

Introduction

This document describes the architecture of MicroArras, a full–text retrieval system being developed at the University of North Carolina at Chapel Hill. The purpose of this report is to describe the retrieval and analysis components of MicroArras in enough detail to allow the reader to understand the relationships among the different modules of the system and the design decisions made in developing the system. Consequently, the description is technical, discussing the system from the programmer's rather than the user's point of view. For a description of the system from the user's perspective, see [Smith, *et. al.*, 1986]. Because this paper concentrates on the architecture of the system, rather than its implementation, the reader does not have to know particulars, such as the programming language (C) or the development environments (UNIX and MS–DOS). However, the reader should have some knowledge of structured software design principles.

The first section of the paper presents an overview of the system. The overview describes the general structure of the system, which is composed of three basic parts: the User Interface, the Engine (that part of the system that does the text retrieval and manipulation), and a language called Flange that is used for two–way communication between the Interface and the Engine. Because the User Interface is not part of the retrieval system, no further details will be presented on the User Interface beyond the brief description in the first section.

The second section describes the communication language, Flange, in more detail. This section describes the architecture of Flange in terms of both the format of Flange messages and the protocols of message passing. This section also presents the routines responsible for sending and receiving Flange messages.

The third section describes the architecture of the Engine. The modules which make up the Engine are presented in terms of a hierarchy of levels, and the interaction between the modules is described.

The Appendices provide additional technical details for Flange, message passing protocols, as well as a description of MicroArras system states.

I. System Overview

Introduction

MicroArras is composed of two sub–programs, the User Interface and the Engine, which communicate with each other through a communication link. The User Interface is responsible for querying the user for requests and sending these requests through the communication link to the Engine; it is also responsible for receiving responses from the Engine and displaying results to the user. The Engine accepts queries from the User Interface and supplies appropriate responses. The User Interface performs all interaction with the user; the Engine performs all text retrieval and manipulation. These interactions are shown in Figure 1.

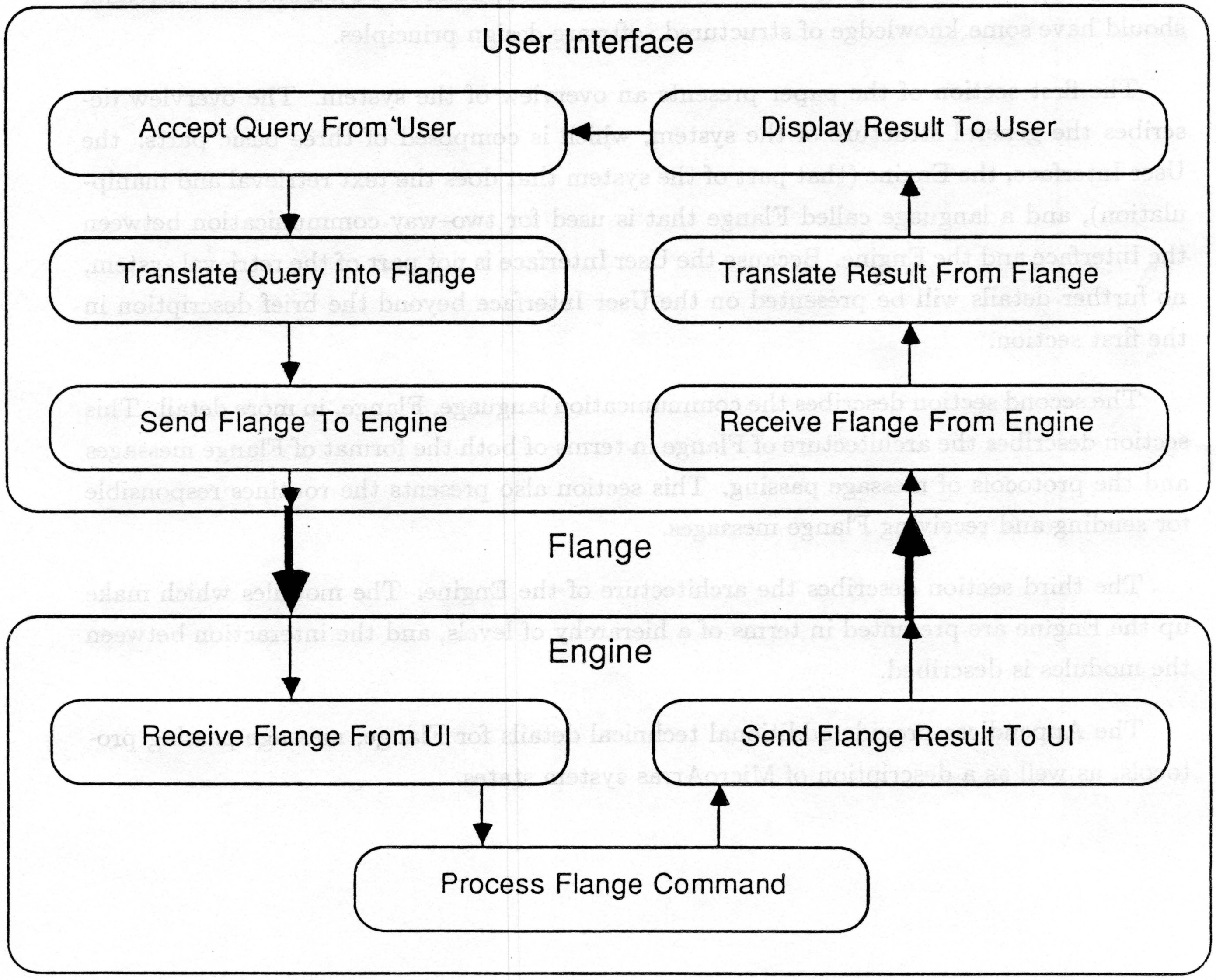

Figure 1.

Flange Overview

Communication between the Engine and Interface is accomplished by sending messages in Flange, a text processing language developed specifically for this purpose in MicroArras. The information contained in Flange messages includes commands given to the Engine, data returned to the Interface, and various control messages. All information passed between the Interface and the Engine in both directions is in the form of Flange messages.

Flange messages fall into four basic categories: *command* messages, *return* messages, *special return* messages, and *control* messages. Commands sent by the User Interface to the Engine are simply requests for the Engine to do some kind of work. Return messages are sent by the Engine to the User Interface when the Engine has processed a command; these messages may contain data or error information, depending on the situation. Special return messages contain return information from a command other than that sent from the Interface (e.g., a command read from a file). Control messages are sent by the User Interface to specify how the Engine is to proceed in situations where the Engine has a choice of types of information to return. The most common control messages are those used to abort or continue the Engine's processing of a command.

User Interface Overview

The method by which the User Interface queries the user or displays information is irrelevant so far as the Engine is concerned. Thus the User Interface may take any number of forms, so long as it can interact with the Engine; the only requirement is that it communicate in Flange. In fact, programs other than a user interface may interact with the Engine. One such application we have developed is an intelligent assistant function, based on expert system technology, that helps users' search the textual database by reformulating queries until the desired number are found and then arranging them in order of probable interest [Gauch & Smith, 1987].

Although the Interface could conceivably be implemented as a part of the same program as the Engine, it is more interesting to consider it as a separate entity. Flange commands are composed of ascii characters and may be transmitted easily through virtually any communications medium. Consequently, the User Interface may even be run on a different machine than the Engine, with Flange messages sent through a phone line or a hardwired connection.

Current work on a menu based interface is nearing completion, while future versions may include graphics–based direct manipulation designs. The User Interface may be simple, exploiting only those functions directly provided by Flange commands, or it may be more complex, either by analyzing and modifying the data returned by the Engine before displaying it, or by supporting complex user queries which require several Flange commands to process. In fact, programs other than interface programs may communicate with the Engine in Flange. In a separate project, expert system technology is being used to build intelligent functions. In this design, the expert system conducts the dialogue rather than the User Interface.

Engine Overview

The MicroArras Engine is the workhorse of the system. It has three primary purposes. First, it very quickly retrieves text from a database of documents. The documents in the database are stored as inverted files [Smith, 1987], which allows fast access to their content regardless of their size. The second purpose of the Engine is to provide functions that create and manipulate high level abstractions of textual "objects". A few examples of text objects are *tokens*, which represent single occurrences of words in a text, and *spans*, which represent sequences of consecutive words in a text. An example of a token is the 293rd word in the text *Atlas Shrugged*. An example of a span is the first two paragraphs of the second chapter of *The Fountainhead*. The third purpose of the Engine is to provide facilities for managing Flange conversation and Flange message interpretation. The modules contained in each of these three levels are described below:

The highest level of the Engine supports Flange management. This level consists of modules that perform verification of Flange commands, execution of Flange commands, communication with the User Interface via Flange messages, and Symbol Table management.

The intermediate level of the Engine supports manipulation of text objects. Various text objects are supported, and routines are provided for creating, deleting, and modifying each of the various types.

The lowest level of the Engine supports access to the inverted text file and the bibliographic file, which stores bibliographic information concerning the texts in the inverted file. All accesses to either file are made through the two modules at this level.

Besides providing these functions, the Engine also performs maintenance operations such as internal error handling and memory management. However, knowledge of these modules is not necessary for understanding the system, and they will not be discussed any further.

II. Flange Architecture

Introduction

This section describes the architecture of Flange. The presentation will concentrate on the format of Flange messages and the message passing protocols, rather than the syntax of particular messages. For information on the syntax of messages refer to the *Flange Specification Document*. This section will also present a few relevant details of the realization of the system.

Format

Flange messages have a strict format. Every message is composed of a *type*, a possible *subtype*, zero or more *data syllables*, and an *end-of-message syllable*. The presence of a subtype depends on the message type. The presence and order of data syllables is determined by the type and subtype. Each data syllable is composed of a syllable type and (possibly null) content.

The message subtypes and syllable types are numerous and will not be given here. The message types will be discussed, however, to aid the discussion of the message passing protocols given in the next section. The Appendix contains a complete list of Flange message types, subtypes, and syllable types.

Flange messages sent by the User Interface are sent complete, in a single packet. Messages sent by the Engine may require multiple packets for a complete message. Control messages are provided to allow the User Interface to tell the Engine whether to continue or abort multiple–packet messages between packets.

Following is a list of the Flange message types. Normal Flange conversations between the User Interface and the Engine consists of messages of the first five types. The last three types are used rarely but are included here for completeness.

COMMAND	A command sent from the User Interface to the Engine. The subtype specifies the specific command to be processed, and the data syllables contain the content of the command
RETURN	A return message sent from the Engine to the User Interface. The subtype specifies the type of return message, and the data syllables contain the content of the return message.

CONTINUE	A control message sent from the User Interface to the Engine. This message tells the Engine to send the next packet of the multiple–packet return message from the previous command. No subtype or data syllables are given.
ABORT	A control message sent from the User Interface to the Engine. This message tells the Engine to send no more packets of the multiple–packet return message from the previous command. No subtype or data syllables are given.
RESET	A control message sent from the Engine to the User Interface if the User Interface aborted a RETURN message before all packets of the message were sent. No subtype or data syllables are given.
SHOW_FLANGE	A control message sent from the User Interface to the Engine after receiving a RETURN_FILE message. This message type is used to force the Engine to send a RETURN_COMMAND message. No subtype or data syllables are given.
DISASTER	A control message sent from the Engine to the User Interface if the Engine suffers an internal error, such as an out of memory condition. No subtype or data syllables are given.
RETURN_COMMAND	A special return message sent from the Engine to the User Interface when the user has requested to see the Flange command that was used to define some particular object. The subtype specifies the type of command being returned, and the data syllables contain the content of the command being returned.

RETURN_FILE	A special return message sent from the Engine to the User Interface in response to a command to read and process Flange commands from a file. The subtype and data syllables are the same as those in an ordinary return message, but the RETURN_FILE type specifies that the command yielding this return message was read from a file rather than sent from the Interface.

Protocols

The most common message types are the COMMAND and RETURN types. Normal conversation between the User Interface and the Engine consists of the Interface and Engine alternately sending COMMAND and RETURN messages. As mentioned previously, messages sent by the User Interface are sent in a single packet, while messages sent by the Engine may require multiple packets for a complete message. If a return message is longer than a single packet, the Engine will append a WAIT_SYLLABLE as the last data syllable to tell the Interface that more data is yet to come. In this case the Interface should not send another command, but should instead send one of the control messages CONTINUE or ABORT. A sample dialogue illustrating this interaction is given below:

1. The User Interface sends a COMMAND requesting that a window be defined. A window is an object representing portions of text from one or more documents in the database (for example, the first three chapters of some text).

2. The Engine sends a RETURN message saying it has defined the window.

3. The User Interface sends a COMMAND requesting that the text comprising the window be displayed.

4. The Engine sends a RETURN message containing the text in the window; since the window may be large a WAIT_SYLLABLE may be appended to the end of the first packet of text. Before returning any more text, the Engine waits for a control message from the User Interface.

5. The User Interface sends a CONTINUE message.

6. The Engine sends the next packet of text. This packet completes the response, so no WAIT_SYLLABLE is appended to the message.

7. The User Interface sends another COMMAND.

The message passing protocol is strict. On system startup the Engine sends a message to the User Interface, letting the Interface know that the Engine is ready for input. The User Interface and Engine are then free to send messages to each other, with the following restrictions:

- Consecutive messages must be passed in different directions; i.e., neither the Interface nor the Engine may send a message without first receiving a response to its previous message.
- If the Engine receives a COMMAND message, it must respond by sending a RETURN message. There are two exceptions to this restriction: if a READ_WS command was received by the Engine it may return a READ_FILE message, and if a DISP_WS command was received the Engine may return a RETURN_COMMAND message.
- If the Engine receives a CONTINUE message, it must send the packet of syllables logically following the most recent message it sent. Since the type and subtype were given in an earlier message, they are not sent in subsequent packets of the same logical message.
- If the Engine receives an ABORT message, it sends a RESET message.
- If the User Interface receives a RETURN message from the Engine, and the last syllable is not a WAIT_SYLLABLE, then the User Interface must send a COMMAND.
- If the User Interface receives a RETURN message which contains a WAIT_SYLLABLE as the last syllable, then the User Interface must send either a CONTINUE or an ABORT message.
- If the User Interface receives a RESET message, it sends a COMMAND.

Three message types, RETURN_FILE, RETURN_COMMAND, and DISASTER are exceptions to the above rules and require special protocols. Examination of these protocols is not necessary for a basic understanding of the system, but the protocols have been included in the Appendix for completeness.

Realization

The Engine uses two routines to send and receive Flange messages. Put_string() is used to send a message to the User Interface, and get_string() is used to receive a message from the User Interface. All messages passed between the Engine and the User Interface are sent via these two routines.

The message passing protocols described above must be implemented in both the User Interface and the Engine. Both the Interface and the Engine must keep track of the state of the communications in order to follow the protocols. The Engine maintains the state in the communications module, which is described in section III, below; the communications module calls put_string() and get_string() as necessary to pass messages. Put_string() is called by the the communications module to send return messages, special return messages, and control messages to the User Interface, and get_string() is used to read control messages sent by the Interface. The only other place put_string() and get_string() are called is in the top level loop, which reads and processes command messages.

III. Engine Architecture

Introduction

As explained above, the MicroArras Engine is responsible for several functions. It will be useful in this section to view the Engine as composing three levels, as described in the Engine Overview section. The three levels which will be examined are the Command Processor (the highest level), the Arrish Engine (the intermediate level), and the Text Access level (the lowest level). Note that the Arrish Engine is a subset of the Engine as a whole. This document will always refer to the Arrish Engine as such, or by the abbreviation "AE", so there should be no ambiguity with the single word "Engine". The responsibilities of the Command Processor (CP) are verifying and executing Flange commands, providing Flange communications facilities, and storing objects. The responsibilities of the Arrish Engine (AE) are to create, modify, and destroy objects. The responsibility of the Text Access (TA) level is to retrieve text and bibliographic information.

The modules that accomplish these functions are outlined below, and are described in detail in the following sections:

Level	Function	Module
CP	Flange Verification	Verify
	Flange Execution	Execute
	Flange Communication	Communication
	Object Storage	Symbol Table
AE	Object Manipulation	Arithmetic Variables
		Category Expressions
		Contexts
		Frequencies
		Hierarchies
		Segment Marks
		SIE Specifications
		Spans
		Tokens
		Token Lists
		Type Lists
		Windows
TA	Text Access	Text Access
	Bibliographic Search	Search

The relationships among modules is shown in Figure 2. A thin solid arrow between two modules indicates that the module at the tail of the arrow may call the module at the head. A thick arrow indicates access in some form other than a function call (e.g., via Flange message passing or file i/o). The single thin dashed arrow represents the limited ability of the Object Manipulation routines to call the Communications Module. In this case only the routine comm_write_word(), which sends a single Flange data syllable, may be called from the Object Manipulation routines.

The following important points should be noted:

- Only the Verify module may call the Execute module.
- Only the Verify and Execute modules can call the Symbol Table Manager.
- The Command Processor modules know the format and syntax of Flange messages, but the Arrish Engine modules do not.
- The Arrish Engine modules know about the internals of the objects they manipulate. The Command Processor modules know nothing about object internals and must call the appropriate module to access information about an object.

MicroArras Call Graph

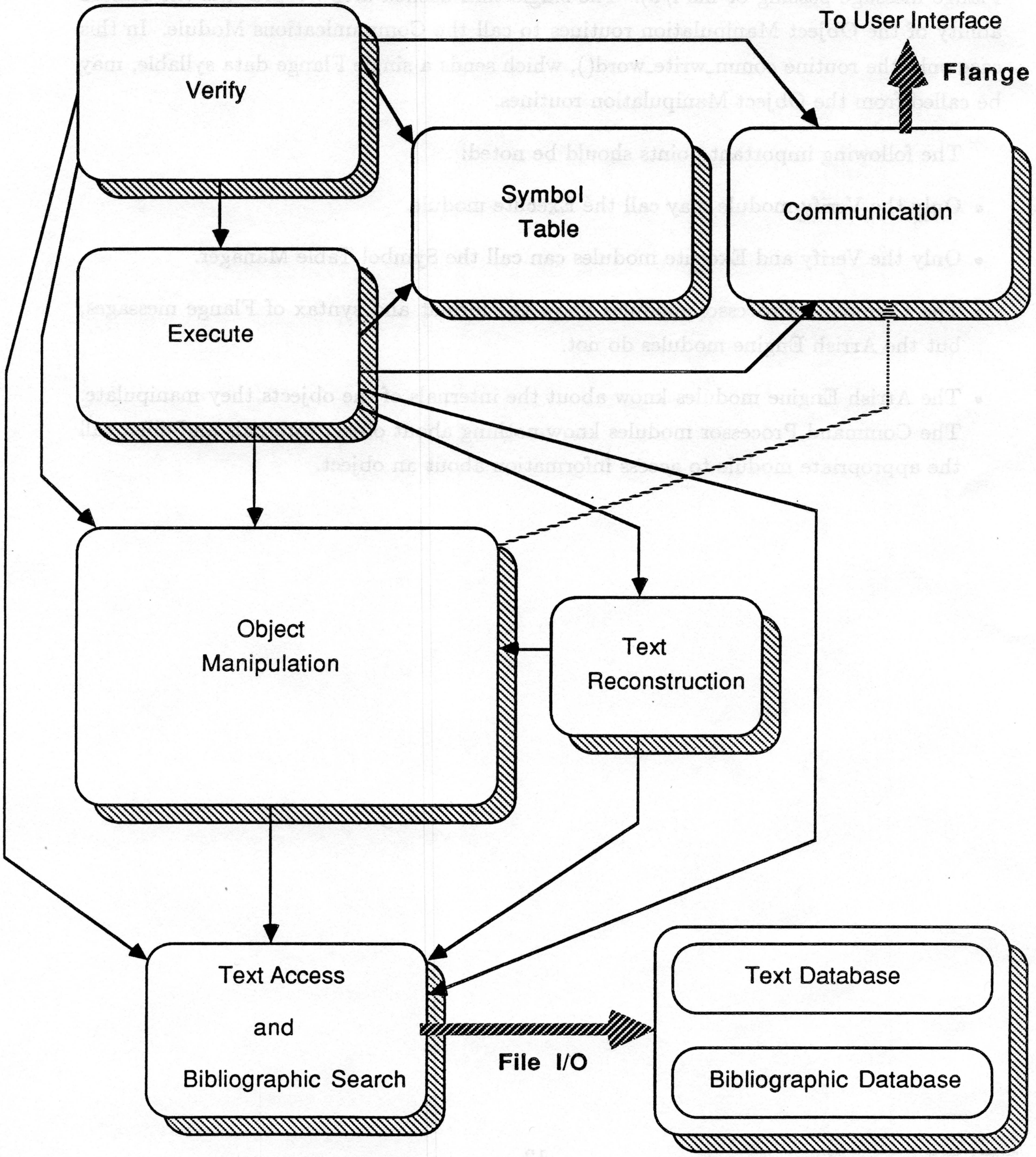

Figure 2.

Command Processor

Verification

When a Flange command is sent from the User Interface, the command is first checked for correctness. There are four steps to this process. First, the overall structure of the message is checked. This step verifies that the message has the type COMMAND, a legal subtype, some positive number of legal data syllables, and an end–of–message syllable. It also verifies that each syllable, as well as the whole command, is of acceptable length.

The second stage of verification is data syllable verification. Each data syllable is checked to verify that the information contained in the syllable is of the type given by the syllable type. For instance, if the syllable type is WINDOW_NAME, the string following the syllable type must be the name of a window object known by the Engine.

The third stage of verification is the data syllable sequence check. This stage verifies that data syllables are present and in the proper order in the message, through constraints determined from the message subtype.

The last stage verifies any special conditions particular to the command.

If a command fails any of the verification steps, a RETURN message with subtype ERRMSG is sent to the User Interface, and no further processing of the command is done. The data syllables of the error message specify the type of error that occurred and the offending syllable of the command, if appropriate. A complete listing of Flange error messages is included in the realization of the system in the file "flange.h".

If the message passes all four verification steps successfully, it is then executed.

Note: a command may still result in an error even though it successfully negotiated the verification stage. This is because some subtle errors cannot be checked for until deep within the execution stage.

Execution

There are two basic types of Flange commands: those that define objects and those that display information. Definition routines define or redefine objects and place the definitions in the symbol table. These routines commonly call Arrish Engine routines to create the object to be stored. Display routines access information about the object to be displayed from the symbol table, or in the case of bibliographic search commands, from the bibliographic file. They then send a Flange message containing the information to be displayed to the User Interface, which displays it to the user.

Thus, the execution routines normally call the Symbol Table Manager to access objects and Arrish Engine modules to manipulate those objects. For example, if a window is being defined, it is first created by the Arrish Engine window module, then stored in the symbol table by the Symbol Table Manager. If the text in the window is later displayed (i.e., the Flange command DISP_TEXT is sent from the User Interface), then the object is retrieved from the symbol table via the Symbol Table Manager and displayed via the Text Reconstruction module.

Symbol Table Manager

The Symbol Table Manager keeps track of information for all text objects in the system, including the name of the object, the type of the object, the Flange command that created the object, and the object itself. Several functions are provided by the Symbol Table Manager: creation of a symbol table entry, deletion of an entry, retrieval of an object, retrieval of the command that created an object, and verification that an object with a given name and type exists.

Communication Module

The Communications module provides the mechanism for transmitting Flange messages to the User Interface. It is a high level interface to the low level put_string() routine, providing buffering for large packets of information. All responses to the User Interface and any control messages from the User Interface are sent through this module. Facilities provided include routines to start a Flange message, add a syllable to a Flange message, and close (and thus send) a Flange message.

The module is also responsible for maintaining the state of the system; it implements the Flange protocol described earlier. The state insures that the Engine and the User Interface remain in sync with each other and that they obey the message passing protocol. Most of the state maintenance required is done implicitly within this module, but routines are provided to allow other modules to retrieve and modify the state of the system. This allows routines that require special handling to force the communications module to take special action on messages sent by those routines. For example, the READ_WS command reads Flange commands from a file and executes them just as if they were sent from the User Interface. Before executing the commands the routine which processes the READ_WS command explicitly sets the state of the system. The commands are then executed, and each sends a RETURN message which is intercepted by the communications module and converted to a RETURN_FILE message. The routine then resets the state to its normal processing state and sends a RETURN message. This scheme allows the User Interface to know which return messages are from the commands read from the file and which is from the READ_WS command itself.

The Appendix gives a complete finite state graph of all states in the system. Following are the most important system states and their meanings:

State	Meaning
init_st	System is initializing
ready_st	Engine is awaiting a COMMAND
processing_st	Engine is processing a command
ack_st	Engine is awaiting a CONTINUE or ABORT
reset_st	Engine received an ABORT, but isn't yet ready for input
shutdown_st	Engine is preparing to shut down

Arrish Engine

Object Manipulation Modules

The Arrish Engine is concerned with the manipulation of text objects. Text objects are concrete implementations of high level abstractions. The text objects supported by the Arrish Engine are:

Token	a linear position of a word in a given text (for example, the fifth word in *Anna Karenina*)
Token List	an ordered list of tokens
Type List	an alphabetically ordered set of words
Span	a sequence of consecutive words in a text
Window	an ordered list of spans
Segment Mark	a mark indicating the beginning of a conceptual segment (for example, paragraph, line, or sentence)
Mark Set	an ordered list of segment marks
SIE Specification	a specification of the segment marks in effect during the display of a window of text
Context	a description of an area surrounding a token (for example, three words on each side of the token)
Category	an unordered set of token lists, type lists, and other categories, or a boolean combination of other categories
Hierarchy	an hierarchically ordered list of segment marks
Arithmetic Variable	a scalar, vector, or matrix of numbers
Frequency	an arithmetic variable which is a vector of numbers representing a frequency distribution (for example, the number of sentences in each paragraph of a text)

These object types can be divided into three intersecting groups. The first group consists of those objects which are directly related to the text database. This group consists of tokens and segment marks, since each text is made up solely of tokens (the text's content) and marks (the text's structure). The next group is made up of those objects that are implemented in the Engine but are not recognized by Flange. These are tokens, spans, mark sets, and frequencies. Objects in this group are used to implement the higher level objects that are recognized by Flange. These high level objects comprise the third group, and consist of token lists, type lists, windows, segment marks, SIE specifications, contexts, categories, hierarchies, and arithmetic variables.

The object manipulation modules provide routines to create, modify, and delete objects of these types. Most of these modules use routines in other AE modules for support. For instance, some window routines may call the span module, and many modules call the token module.

The following sections provide details of the text objects, as well as the relationships among the objects. These descriptions use the terms *static* and *dynamic*. A static object is bound to a certain position or set of positions in a document when the object is defined. A dynamic object is one which must be applied to a static object before it is bound to a specific location. For example, a token is static since it is bound to a specific location in a specific text (for example, the twentieth word in *Even Cowgirls Get The Blues*, while a type (word) is dynamic, since it may reference any occurrence of that word in any text (for example, the word "thumb").

Tokens

A token is a linear position in a text. Examples of tokens are the 3,498th word of *The Unbearable Lightness of Being* or the first word of *The King's Indian*. Tokens do not reference any other type of object; they consist only of a text and a position within that text. Tokens are static.

Token Lists

A token list is an ordered list of tokens. Multiple occurrences of the same token in a list are not allowed. Tokens are ordered primarily by text, and secondarily by position within text. Text order is derived from the position of the text within the database of texts. For example, a token list may consist of the second and ninth words of *The Society of Mind* and the first word of *The Master and Margarita*, in that order. Token lists do not reference any other type of object except tokens. Token lists are static.

Type Lists

A type list is an alphabetically ordered list of types, or words. Multiple occurrences of the same word are not allowed. The words may or may not occur in any given text. A type list may consist of the words "ego" and "selfish", for example. Type lists are not dependent on any other object. Type lists are dynamic, and become bound when used within a window.

Spans

A span is a set of sequential words in a text, and thus can be represented by a pair of tokens, representing the first and last words in the span. The first token must be less than or equal to the second token. For example, a span may consist of the second paragraph of *The Mind's I.* Spans reference tokens. Spans are static.

Windows

A window is an ordered sequence of spans. Spans are ordered by the tokens beginning the spans. Spans in a window may not overlap; that is, if a token is in one span of a window it will not be in another. For example, a window may consist of a span consisting of the first paragraph of *The Moon is a Harsh Mistress* and the third paragraph of the same text. Windows are dependent on spans (and thus on tokens, since spans depend on tokens). Windows are static.

Segment Marks

Segment marks are used to logically partition a text. Marks are an inherent part of the texts and thus cannot be defined. Before a document is put into the database, formatting information is inserted into the text. These format marks define the beginning of logical sections of the text, such as words, lines, pages, paragraphs, and chapters. The Engine recognizes the segment marks that have been inserted into the text as objects and provides mechanisms for computing locations of marks (for example, where the second paragraph of *The Vampire Lestat* begins), translating between marks (for example, determining what page the third chapter begins on), and performing various other functions. Segment marks are dynamic, since an index and a text must be applied to a mark before it can specify a position. For example, "paragraph" doesn't specify an absolute position, but "paragraph 2 of *The Sirens of Titan*" does.

Mark Sets

A mark set is an ordered list of segment marks. Order in a list is imposed by the user and is unrestricted, although some orders make more sense than others. A mark may be repeated in a set, although this would probably not be useful. The facilities described above for manipulating segment marks are implemented in the Arrish Engine in the mark set module. Segment marks are implicit within a document and cannot be defined, while mark sets may be defined to consist of arbitrary lists of marks. For example, a mark set may consist of chapter, paragraph, sentence, and word marks. Because they are dependent on segment marks, mark sets are dynamic.

SIE Specifications

SIE (Segment-In-Effect) specifications describe the formatting information to be returned with text being sent to the User Interface by the Engine. Three attributes are covered by an SIE specification. The first specifies the segment marks to be used to describe the starting position of the text being returned (for example, returned text could be referenced as either "chapter 2, paragraph 1, sentence 7, word 3", or "page 35, line 12, word 13"). The second attribute specifies the segment titles to be returned, allowing, for example, titles of chapters to be displayed if they are present in the document. The third attribute controls whether or not the segment marks describing the starting position are hierarchical with respect to one another; that is, whether relative or absolute indexing should be used in describing the starting position. For example, "chapter 3, paragraph 1" is hierarchical, while "chapter 3, page 79" may not be. In the first case, what is meant is that the starting position of the text begins in the first paragraph of the third chapter, while in the second case the starting position of the text is in the third chapter, and is on page seventy-nine. SIE specifications depend only on segment marks. SIE specifications are not used to reference text positions, and are thus neither static nor dynamic.

Contexts

A context is used to delineate an area around a token (the context's *pivot*). They are like types, however, in that they are not bound to a single text or position. A context is applied to a token to yield a span. For example, a context may specify five words on each side of the pivot; when this context is applied to a token it would yield a span of 11 tokens. Contexts reference segment marks. Contexts are dynamic, and are bound when applied to a token.

Categories

A category (or "category expression") is either a boolean combination of other categories or a group of type lists, token lists, and other categories. A boolean category is called a "configuration expression", and may be, for example, "*Cat1* and (*Cat2* or *Cat3*)", where *Cat1*, *Cat2*, and *Cat3* are previously defined categories. The boolean operators apply within a given context, so a token is in the example category above if it is within the context of *Cat1* and is within the context of either *Cat2* or *Cat3*. The second type of category is called a "recursive category". A recursive category is a collection of other categories. For example, a recursive category may consist of *Types1* and *TokenList1*, where *Types1* and *TokenList1* are previously defined type lists and token lists, respectively. Categories are dependent on type lists, token lists, other categories, and contexts. Categories are dynamic; a category is evaluated in a window to yield a token list.

Hierarchies

A hierarchy is an ordered list of segment marks. Order is imposed by the user and unrestricted; however, hierarchy objects are used to support logical hierarchies (e.g., "chapter, paragraph, sentence, word"), so non–hierarchical ordering of marks is not really useful. Hierarchies, like mark lists, must be paired with indexes and evaluated within a text to yield an absolute position, and are thus dynamic.

Arithmetic Variables

An arithmetic variable is a zero–, one–, or two–dimensional array (i.e., a scalar, vector, or matrix) of floating point numbers. Variables do not reference text positions and are neither static nor dynamic.

Frequencies

A frequency is an arithmetic variable representing a frequency distribution. For example, it may represent the frequency with which a certain word occurs in each chapter of a given text. Frequencies have no structure of their own; they are simply arithmetic variables that are vectors. Since frequencies are arithmetic variables they can be considered to be neither static nor dynamic. However, since a frequency represents a distribution of tokens (which are static) across a window (which is also static), it may be considered to be static entity.

The relationships among the objects are shown in Figure 3. The highest level objects are at the bottom, while the more basic low level objects are at the top. The arrows can be read "are built upon", so for instance windows are built upon spans, and spans and token lists are built upon tokens.

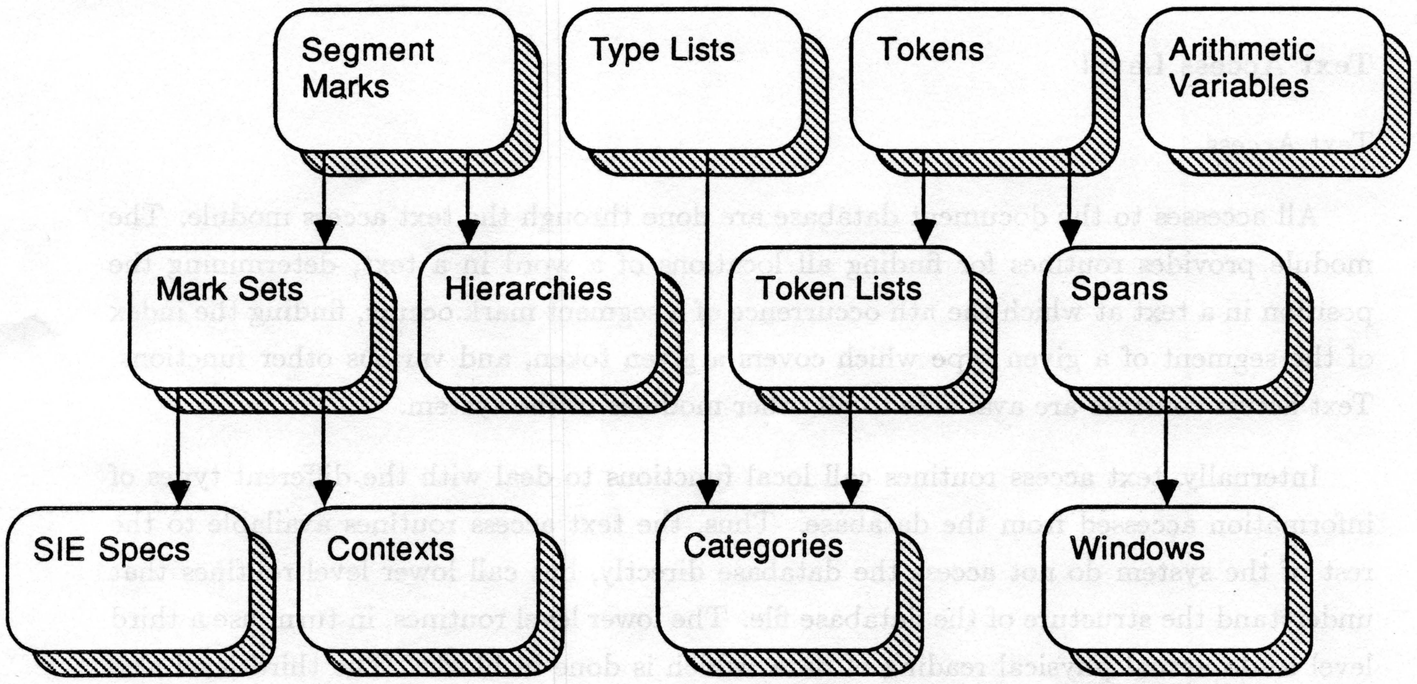
Segment Marks
Type Lists
Tokens
Arithmetic Variables
Mark Sets
Hierarchies
Token Lists
Spans
SIE Specs
Contexts
Categories
Windows

Figure 3.

Text Access Level

Text Access

All accesses to the document database are done through the text access module. The module provides routines for finding all locations of a word in a text, determining the position in a text at which the *n*th occurrence of a segment mark occurs, finding the index of the segment of a given type which covers a given token, and various other functions. Text access routines are available to all other modules of the system.

Internally, text access routines call local functions to deal with the different types of information accessed from the database. Thus, the text access routines available to the rest of the system do not access the database directly, but call lower level routines that understand the structure of the database file. The lower level routines, in turn, use a third level to do actual physical reading of data, which is done in blocks. The third level also maintains a list of buffers containing recently read blocks.

Bibliographic Search

All accesses to the bibliographic file are done through the bibliographic search module. This module provides routines for finding bibliographic citations for texts, for locating texts whose citations match a given boolean search request (for example, all texts whose author is Ayn Rand or were written between 1935 and 1940), and for computing the intersection and union of text lists.

IV. Acknowledgments

Portions of this research were supported by the National Endowment for the Humanities (Grant # RT-20483-84) and by Northern Telecom, Inc. The project was led by faculty researchers John B. Smith and Stephen F. Weiss. Research Associate Gordon J. Ferguson was chief architect of the system. Graduate students who contributed to the coding effort included Scott Southard, Bobby Stam, Candee Ellis, John Gauch, Susan Gauch, Richard Potter, and James Ericksen. Scott Southard was the principal author of this report.

References

Smith, J. B., Weiss, S. F., & Ferguson, G. J. (1986), *MicroArras: An Overview*, Chapel Hill, NC: UNC Department of Computer Science Technical Report # 86-017.

Gauch, S. & Smith, J. B. (1987), *Intelligent Search of Full-Text Databases*, Chapel Hill, NC: UNC Department of Computer Science Technical Report # 87-035.

Appendices

Flange Message Types

COMMAND	A command being sent from the User Interface to the Engine. The subtype specifies the specific command to be processed, and the data syllables contain the content of the command
RETURN	A return message sent from the Engine to the User Interface. The subtype specifies the type of return message, and the data syllables contain the content of the return message.
RETURN_COMMAND	A special return message sent from the Engine to the User Interface in response to a DISP_WS or READ_WS command. The subtype specifies the type of command being returned, and the data syllables contain the content of the command being returned.
RETURN_FILE	A special return message sent from the Engine to the User Interface in response to a READ_WS command. The subtype and data syllables are the same as those in an ordinary return message, but the RETURN_FILE type specifies that the command yielding this return message was read from a file rather than sent from the Interface.
RESET	A control message sent from the Engine to the User Interface if the User Interface aborted a RETURN message. No subtype or data syllables are given.
DISASTER	A control message sent from the Engine to the User Interface if the Engine suffers an internal error, such as an out of memory condition. No subtype or data syllables are given.
CONTINUE	A control message sent from the User Interface to the Engine. This message tells the Engine to continue sending the return message from the previous command. No subtype or data syllables are given.
ABORT	A control message sent from the User Interface to the Engine. This message tells the Engine to stop sending the return message from the previous command. No subtype or data syllables are given.
SHOW_FLANGE	A control message sent from the User Interface to the Engine after receiving a RETURN_FILE message. This message type is used to force the Engine to send a RETURN_COMMAND message. No subtype or data syllables are given.

Flange Command Message Subtypes

WINDOW_DEF	Define a window from a list of endpoints or pivot–context pairs.
WINDOW_SET	Set the default window.
WINDOW_UNION	Define a window as the union of other windows.
WINDOW_INTERSECT	Define a window as the intersection of other windows.
WINDOW_CLIP	Define a window as part of another window.
CONTEXT_DEF	Define a context.
CONTEXT_SET	Set the default context.
SIE_DEF	Define a segment–in–effect specification.
SIE_SET	Set the default segment–in–effect specification.
DEF_HIER	Define a hierarchy.
LOCATE_HIER	Translate between hierarchies.
HIER_MARKS	Display format marks in hierarchy.
DEF_LIN_CAT	Define a token list (linear category).
LIN_CAT_ADD	Add elements to a token list.
LIN_CAT_DELETE	Delete elements from a token list.
LIN_CAT_CLIP	Define a token list as the first n tokens of another list.
LIN_CAT_SIZE	Display the number of tokens in a token list.
DEF_TLIST	Define a type list by enumerating types in the list.
TLIST_ADD	Add types to a type list.
TLIST_DELETE	Delete types from a type list.
PATTERN	Define a type list of all words matching a pattern.
RANGE	Define a type list of all words in an alphabetic range.
WORD_BY_FREQ	Define a type list of all words occurring with a given frequency.

CO_OCCURENCE	Define a type list of all words occurring close to a given token list.
TLIST_CLIP	Define a type list of the first n tokens of another list.
PATTERN_CLIP	Define a type list of all words of another list matching a pattern.
RANGE_CLIP	Define a type list of all words of another list in an alphabetic range.
TLIST_SIZE	Display the number of types in a list.
DEF_REC_CAT	Define a recursive category from other categories, type lists, and token lists.
REC_CAT_ADD	Add elements to a recursive category.
REC_CAT_DELETE	Delete elements from a recursive category.
CHANGE_WINDOW	Modify the window attached to a recursive category.
CONFIG	Define a configuration expression.
EVAL_CAT	Define a token list by evaluating a recursive category.
DEF_VAR	Define an arithmetic variable.
ARITH_VAR	Perform an arithmetic operation on variables.
FREQUENCY	Define an arithmetic variable by performing a frequency distribution of format marks or category words across a window.
SIZES	Define an arithmetic variable by computing the distribution of sizes of a frequency distribution.
READ_WS	Read Flange commands from a file.
WRITE_WS	Write Flange commands into a file.
DELETE_WS	Delete objects in the workspace.
DISP_WS	Display names of objects in workspace, or Flange commands use to create named objects.
DISP_TEXT	Display text in a window.
DISP_CONC	Display a concordance of words in a category or list.

DISP_DICT	Display the words in a type list, with their frequency of occurrence.
DISP_VAR	Display an arithmetic variable.
CANON	Display a list of the canonical format marks.
DISP_MARKS	Display the format marks used in a specific text.
FIND_TEXT	Display names of texts in database that match boolean search constraints.
FIND_CITATION	Display the citation for the given text.
OK_TOKEN	Verify that syllable is valid.
TOKEN_TYPE	Display the syllable type of the given syllable.
STOP	Stop the MicroArras session.
NOP	No operation.

Flange Return Message Subtypes

CANON_LIST	A list of canonical format marks.
CITATION	A bibliographic citation.
ERRMSG	An error message.
HIER_POS	A hierarchical description of a position in a text.
LEXICON	A list of words, possibly with their frequencies of occurrence.
MARK_LIST	A list of segment marks used in a text.
NAMES	A list of defined objects.
NUMBERS	A scalar, vector, or matrix of numbers.
SIZE	A number representing the length of a token list or type list.
SYLLABLE_TYPES	A list of syllable types.
TEXT	A sequence of text words and segment marks.
TEXT_LIST	A list of text titles.
WARNMSG	A warning message.

Flange Data Syllable Types

FILE_NAME	The name of a file.
TEXT_NAME	The name of a text in the database.
WINDOW_NAME	The name of a window.
CONTEXT_NAME	The name of a context.
CAT_NAME	The name of a category.
TLIST_NAME	The name of a type list.
VAR_NAME	The name of an arithmetic variable.
SIE_NAME	The name of an SIE specification.
HIER_NAME	The name of a hierarchy.
NAME	The name of an object.
NEW_NAME	An undefined name.
FIG_OP	A boolean operator used in a configuration expression.
FIELD_OP	A field of a bibliographic citation.
ARITH_OP	An arithmetic operation.
BOOL_OP	A boolean operator used in a bibliographic search.
WRITE_OP	An operator specifying whether to overwrite or create a file.
CLASS_OP	A class of object (window, context, etc.).
SEARCH_OP	An operator specifying precision of bibliographic search.
STRING	Any string of characters.
COUNT	A non-negative integer.
FMARK	A segment (format) mark.
TEXT_WORD	A word in a text.
LINENO	A line number of a configuration or arithmetic expression.
PARTITION	A positive integer representing a number of partitions.

INTEGER	An unrestricted integer.
FLOAT	A floating point number.
NEW_LINE	A syllable used as a delimiter in a Flange message.
WAIT_CODE	A code used for Flange protocol management.
UNKNOWN	An object of unknown class.
ANY	An object of any object class.
END	The end of a command or return message.

Flange Message Passing Protocols

The following table presents the possible actions that may be taken by the Engine when it receives a message of the given type from the User Interface:

User Interface	Engine	When Used
COMMAND	RETURN	Normal return message
	RETURN_FILE	Return message from command read from a file
	RETURN_COMMAND	Content of message of command read from a file
	DISASTER	Engine internal error
CONTINUE	(Next Packet)	More of last message needs to be sent
	RETURN	Finished sending commands read from file or symbol table
	RETURN_COMMAND	If Engine is sending commands read from file
	DISASTER	Engine internal error
ABORT	RESET	User aborted return message between packets
	DISASTER	Engine internal error
SHOW_FLANGE	RETURN_COMMAND	User requested content of message read from file or symbol table
	DISASTER	Engine internal error

The following table presents the possible actions that may be taken by the User Interface when it receives a message of the given type from the Engine:

Engine	User Interface	When Used
RETURN	COMMAND	Send another command
RETURN_COMMAND	CONTINUE	Send next command read from file or symbol table, or return message
	ABORT	Stop sending commands read from file or symbol table; just send return message
RETURN_FILE	CONTINUE	Send next command read from file
	ABORT	Stop sending commands read from file
	SHOW_FLANGE	Show the Flange command just read from file
RESET	COMMAND	Send another command
DISASTER	any	Prepare for Engine to die

Figure 4 shows all possible states of the system and the relations among them. The rectangular nodes represent the states the Engine is in when ready to accept a Flange message from the User Interface. The elliptical nodes represent states passed through during processing of a message. Transitions are made from a rectangular node to an elliptical node when the Engine receives a Flange message from the User Interface, and from an elliptical node to a rectangular node when the Engine sends a message to the Interface. For example, if the system is in the processing_st state and receives a command other than READ_WS, the system will be in the wait_st state. Eventually the Engine will send a Flange message. If the message is short enough to fit into a buffer the system follows the arrow back to the processing_st state. If not, the system goes to the ack_st state and is then ready to accept only a CONTINUE or ABORT control message.

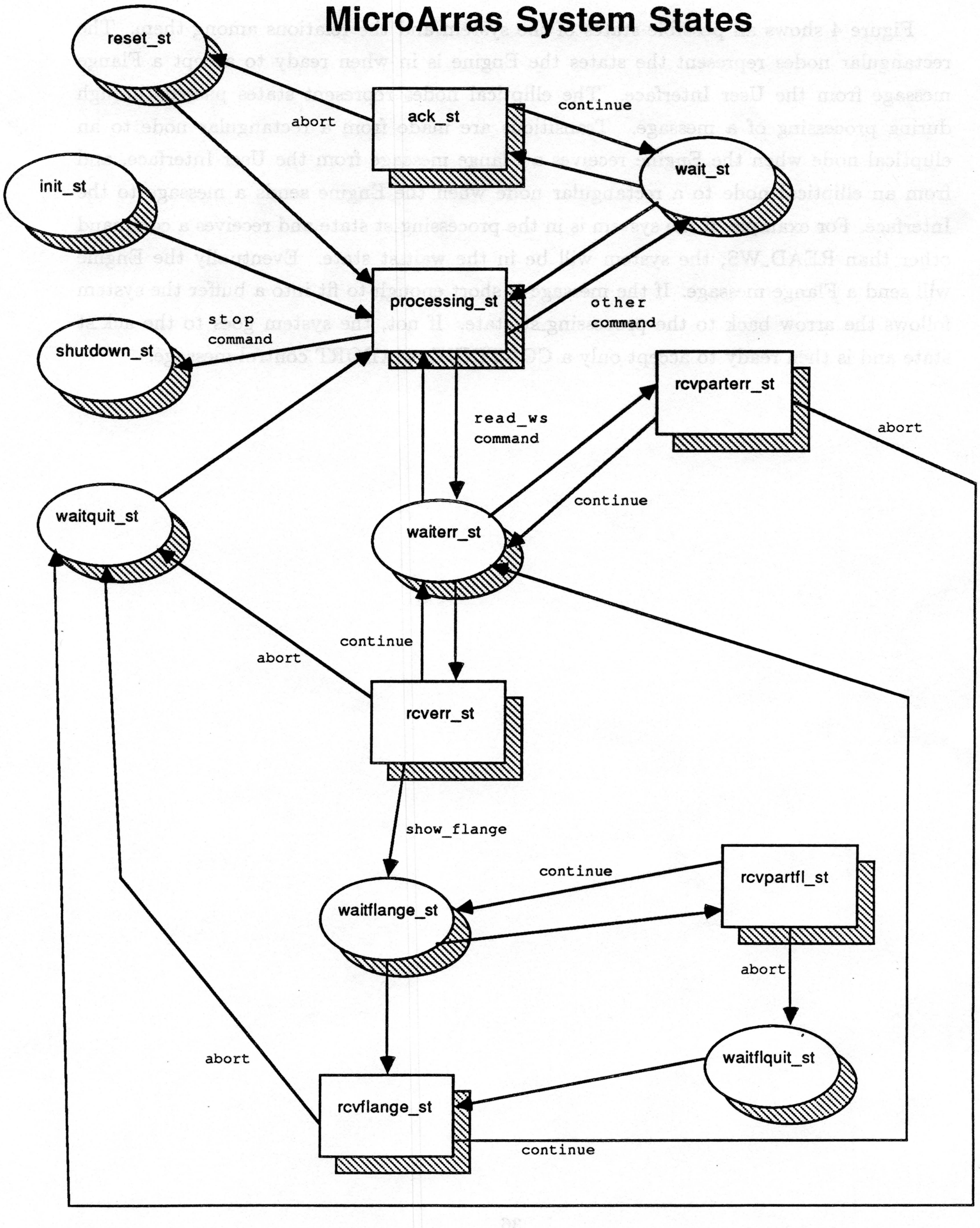
MicroArras System States
reset_st
ack_st
abort
continue
wait_st
init_st
processing_st
other command
stop command
shutdown_st
rcvparterr_st
abort
read_ws command
continue
waitquit_st
waiterr_st
continue
abort
rcverr_st
show_flange
continue
rcvpartfl_st
waitflange_st
abort
waitflquit_st
abort
rcvflange_st
continue

Figure 4.