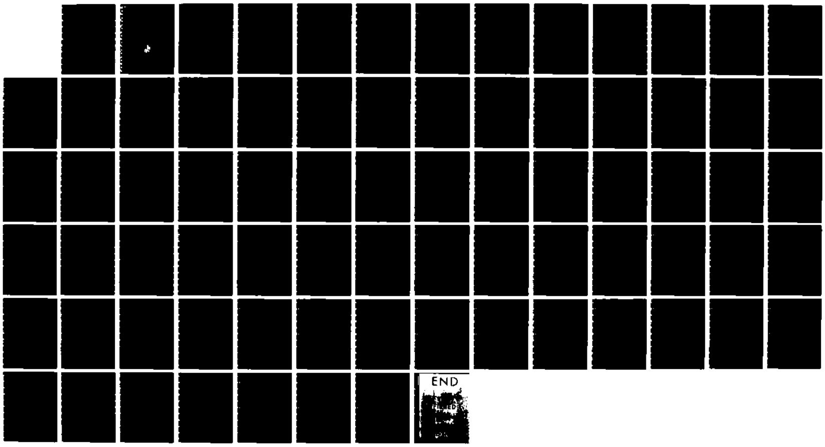
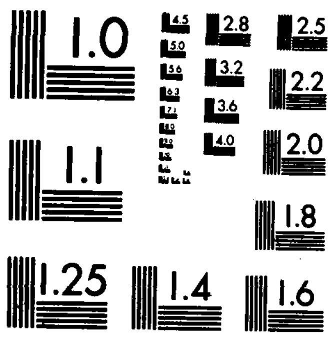
AD-A136 943   ISSUES IN INTERACTION LANGUAGE SPECIFICATION AND REPRESENTATION(U) VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG COMPUTER S.. D H JOHNSON ET AL. NOV 83   1/1

UNCLASSIFIED   CSIE-83-15 N00014-81-K-0143   F/G 5/8   NL




1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

1

ISSUES IN INTERACTION LANGUAGE SPECIFICATION AND REPRESENTATION

Deborah H. Johnson
H. Rex Hartson

Virginia Polytechnic Institute and State University
Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

CSIE-83-15 November 1983

ISSUES IN INTERACTION LANGUAGE SPECIFICATION AND REPRESENTATION

Deborah H. Johnson

H. Rex Hartson

TECHNICAL REPORT

Prepared for
Engineering Psychology Group, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CSIE-83-15	2. GOVT ACCESSION NO. AD-A136943	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Issues in Interaction Language Specification and Representation		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Deborah H. Johnson H. Rex Hartson		8. CONTRACT OR GRANT NUMBER(s) N00014-81-K-0143

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, Virginia 24061
		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
		61153N42; RR04209;
		RR0420901;
		NR SRO-101

11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 442 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE November 1983
		13. NUMBER OF PAGES 64
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Human Factors, Languages, Interaction Languages, Design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Interaction between a human and a computer necessarily involves the use of a language in which the two can communicate. For application systems which are created under the Dialogue Management System (DMS), this language is usually an interaction language. Issues in the implementation of interaction languages are discussed, including language design, language specification and representation schemes, and language recognition. Components of an interaction language are classified into categories which are analyzed in terms of their specification needs. A model for interaction language specification is presented which depicts

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Continued)

several inter-related submodels as a communication path between a dialogue author and an end-user.

Because the dialogue author who is creating the user interface for application systems is not expected to be a language specialist, an automated tool to facilitate interaction language design, specification, representation, and parsing is being incorporated into the Author's Interactive Dialogue Environment (AIDE). An interactive example-based interface for syntax specification, Language-By-Example (LBE), guides the dialogue author at design-time in specifying an interaction language for an application system. An example of the use of LBE for defining command strings is presented.

ACKNOWLEDGMENT

This research was supported by the Office of Naval Research under contract number N00014-81-K-0413 and work unit number SRO-101. The effort was supported by the Engineering Psychology Group, Office of Naval Research under the technical direction of Dr. John J. O'Hare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

The authors wish to acknowledge Dr. Roger Ehrich, who first realized the importance of language in human-computer interfaces under DMS and AIDE, and continually encouraged its consideration.

ABSTRACT

Interaction between a human and a computer necessarily involves the use of a language in which the two can communicate. For application systems which are created under the Dialogue Management System (DMS), this language is usually an interaction language. Issues in the implementation of interaction languages are discussed, including language design, language specification and representation schemes, and language recognition. Components of an interaction language are classified into categories which are analyzed in terms of their specification needs. A model for interaction language specification is presented which depicts several inter-related submodels as a communication path between a dialogue author and an end-user.

Because the dialogue author who is creating the user interface for application systems is not expected to be a language specialist, an automated tool to facilitate interaction language design, specification, representation, and parsing is being incorporated into the Author's Interactive Dialogue Environment (AIDE). An interactive example-based interface for syntax specification, Language-By-Example (LBE), guides the dialogue author at design-time in specifying an interaction language for an application system. An example of the use of LBE for defining command strings is presented.

CR Categories and Subject Descriptors: D.2.1 [Software Engineering Requirements/Specifications], D.2.2 [Tools and Techniques], D.3.1 [Programming Languages Formal Definitions and Theory], D.3.2 [Programming Languages Classification], H.1 [Information Systems Models and Principles]

General Terms: Human Factors, Languages, Interaction Languages, Design

TABLE OF CONTENTS

Issues in Interaction Language Specification and Representation

Deborah H. Johnson
H. Rex Hartson

"Combine the technology of the future with a total summer camp experience in the mountains of southwest Virginia. Residential computer camp for 10-16 year olds, with instruction by fully qualified staff..."

1. INTRODUCTION

This advertisement from a recent issue of the Virginia Tech *Collegiate Times* serves as a broad statement on the widespread proliferation of computers in twentieth century life. No longer an esoteric magic box usable by only a select few, the computer is a fact of life in today's world. Everyone, from grandmothers using on-line information storage and retrieval systems at the public library to ten year olds attending summer computer camp, is being introduced to the wonder of this electronic marvel. Unfortunately, "wonder" can have more than one meaning, especially when associated with the use of computers. The sense of effectiveness and efficiency one can experience when using a computer may all too quickly be replaced by a feeling of uncertainty and frustration. This frequently occurs because of the lack of emphasis on development of an effective, natural human-computer interface. Because of the rapid expansion of computers into all areas of life, the focus has been largely on simply "getting something working," while little or no attention has been paid to making this machine easy for humans to use. Its power and productiveness are often masked by a user interface that is difficult and confusing for a human. Thus, the need for an effective human-computer interface is apparent.

2. OVERVIEW OF DMS AND AIDE

2.1. BACKGROUND AND PURPOSE OF DMS

At Virginia Tech, the Office of Naval Research is sponsoring a three year program devoted to the research and development of effective human-computer interfaces. The work is being done jointly by the Departments of Computer Science and Industrial Engineering. One task of the research effort is focusing specifically on the management of human-computer dialogues. A major goal of this task is the development of an automated human-computer system to aid in the development of other human-computer application systems. This Dialogue Management System (DMS) [HARTH83] is an extensive research effort currently involving two faculty members, five graduate research assistants, and one full-time programmer.

2.2. NEW CONCEPTS AND ROLES IN DMS

2.2.1. Dialogue Independence

Because the emphasis of our work is on human-computer dialogues, our research group believes that the dialogue which occurs between the computer and the human user is as important as the computational software of an application system. If the human-computer interface is not easily usable by a human, the robustness, correctness, and efficiency of the associated computational component is of little consequence.

In response to this need, the concept of dialogue independence has developed as the underlying premise of DMS. *Dialogue independence* entails the separation of the dialogue component from the computational component of an application system. The dialogue component and computational component must communicate through a common interface so that they can be integrated together for execution of the completed application system. This interface between the dialogue component and computational component conducts an "internal dialogue" and the interface between the dialogue components and the user conducts an "external dialogue." (See Figure 1.) *External dialogue* is the traditional human-computer interface for the interaction between the user and the system. It is highly variable, limited only by the imagination of the person who creates the content of the dialogue component. *Internal dialogue*, on the other hand, has no direct connection to the user of the system, but serves as a link between the dia-

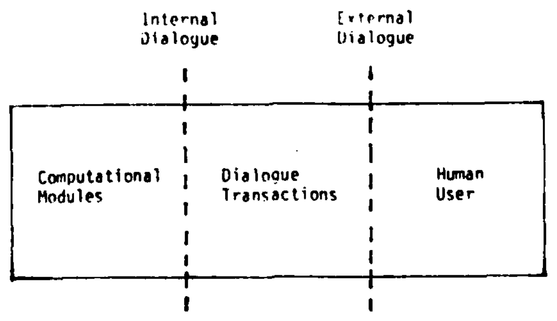


Figure 1. Internal and External Dialogue

logue component and the computational component of a system. It must therefore be formally specified and is much less variable in its form.

2.2.2. Dialogue Author

In order to emphasize the separation of dialogue and computational components of software systems, separate roles are responsible for each of the two components. An application programmer writes computational components, but writes no dialogue. A new role, that of a ***dialogue author***, has sole responsibility for developing the dialogue which comprises the human-computer interface of a system. The dialogue author is a person who is not necessarily a skilled programmer, but rather is oriented towards the human factors of human-computer interface development. The dialogue author creates the dialogue so that its content reflects the principles of good human-computer interaction.

The major reason for this separation of roles between the dialogue author and an application programmer is that an application programmer is generally not skilled in writing good human-computer dialogues. An application programmer is typically much more intent upon development of algorithms and other computational considerations. An application programmer may even find it distracting, in the middle of a lengthy, logically complex piece of software, to have to write code interacting with the user to deal with such things as input data checking or message format consistency. The main objective of a dialogue author is to write dialogue that incorporates good human-computer interface guidelines, without having to know programming languages and techniques.

Thus, the separation of dialogue from the computational software component, and the parallel roles of the dialogue author and the application programmer, form the underlying philosophy of DMS. This separation produces more effective human-computer interfaces, and it allows these interfaces to be quickly and easily modified as user needs change.

2.3. COMPONENTS OF DMS

DMS consists of four major components: automated tools for the author, automated tools for the programmer, an execution environment, and a holistic methodology to integrate all the other components together. The tools facilitate and encourage the creation of software systems with quality human-computer interfaces. The application programmer uses the automated tools of the programming environment to create computational components for an application system. At the same time, the dialogue author uses the automated tools of the Author's Interactive Dialogue Environment (AIDE) [JOHND82] to create the dialogue components for that same application system. DMS also provides the multiprocess execution environment for itself, as well as for the application systems it is used to create. These application systems are embedded in the DMS execution environment for their own execution. Finally, DMS provides the comprehensive methodology for system development. Use of the automated tools by both dialogue author and application programmer, inter-role communication issues between dialogue author and application programmer, and system documentation for use by both are emphasized in the DMS methodology. All of these aspects are critical if the dialogue author and the application programmer are to interact successfully to produce a fully integrated completed application system. This methodology is fully discussed in [YUNTT84].

2.4. DIALOGUE TRANSACTION MODEL

Every exchange of information between a computer and its user follows a specific sequence. Under DMS, the exchange is called a dialogue transaction, and a human-computer interface is composed of many transactions. A *dialogue transaction model* has been developed to serve as the basis for the understanding and design of human-computer interfaces using AIDE. An interaction within a transaction is comprised of three parts: system *display*, followed by human *language input*, followed by system *confirmation*. Any of these parts may be implicit in a given interaction.

The display can have many syntactic forms, each of which serves essentially the same semantic function. Menus, keypads, touch-panel displays, graphical icons, ordinary textual or voice requests for input, or even a simple "Ready" message are all examples of displays. The language or input part of an interaction embodies the interaction language, or the language of communication between the end-user and the application system. Whatever the syntactic form of the user language input, it can be precisely defined as a part of an interaction language. This part is quite complex, and is the major topic of this paper. Finally, the confirmation part is typically either a textual or a graphical response to a user's input. Of all the parts, it is the one most likely to be implicit (except when an error message is required). Each instance of a transaction (created using AIDE) has a definition for each of its parts (the output of AIDE) stored in a transaction database (TDB), individually retrievable for modification or execution.

There are two major manifestations of this transacton model, seen at run-time and at design-time. At run-time, the form of the transaction model is built into the control structure of the transaction executor. This transaction executor is data-driven at run-time to instantiate a dialogue transaction. Each part of the transaction has its own executor, called by the transaction executor: a display executor to interpret the display definition and to produce a display; a language executor to interpret the language input definition and to accept, parse, and validate the user's input; and a confirmation executor to interpret the confirmation definition and to produce the system confirmation.

At design-time, there are also control and data structures which reflect the parts of a transaction. In addition, there is a set of tools for producing each of the transaction parts. These tools are integrated into the interactive system called AIDE, which is discussed in the following section.

2.5. AUTHOR'S INTERACTIVE DIALOGUE ENVIRONMENT (AIDE)

The purpose of AIDE is to provide an automated set of tools for a dialogue author to use in creating human-factorable (flexible enough to incorporate applicable human factors results when they become available) human-computer dialogues. The structural organization of AIDE is shown in Figure 2. It consists of the dialogue author's interface, which provides the human-computer dialogue between the dialogue author and the tools, and integrates this

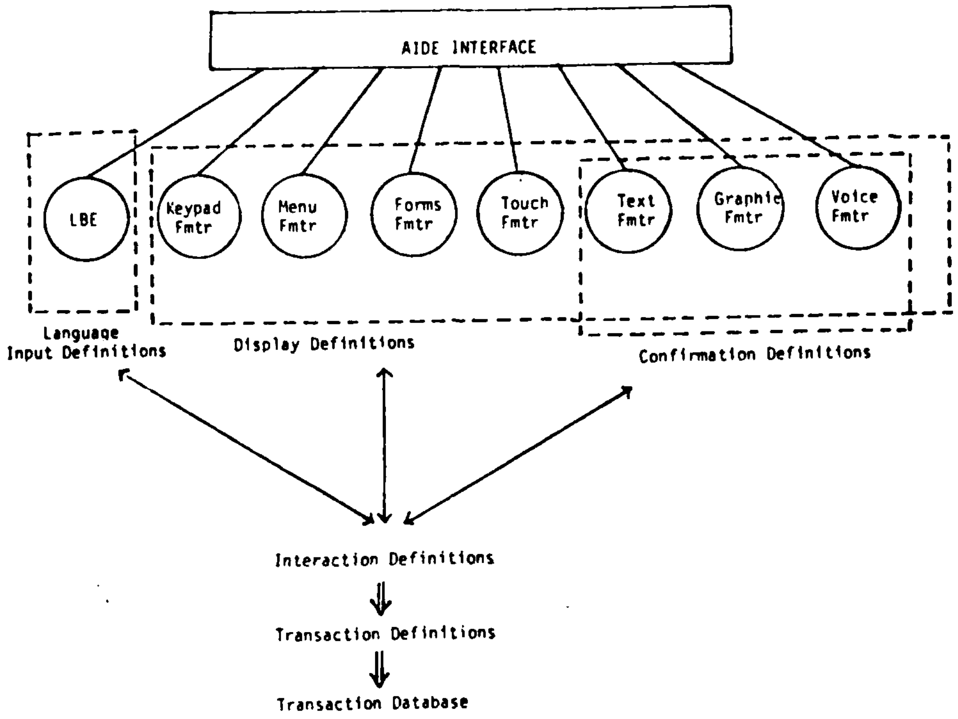


Figure 2. Author's Interactive Dialogue Environment

diverse group of interactive facilities for creating dialogue transactions. Such tools as a menu formatter, a keypad formatter, a text formatter, a graphics formatter, a voice formatter, a forms formatter, and a touch panel display formatter are used to create the display and confirmation parts of a transaction. An example-based language definition tool called Language-By-Example (LBE), discussed later in this paper, facilitates the design, specification, representation, parsing, and recognition of the language input part of a transaction for an interaction language. (Parsing and recognition are more the purview of the run-time language executor [NARAP83], and will not be discussed in depth in this paper.) Finally, the output of these tools, the internal representations or definitions of the parts of a transaction, are held during their creation, modification, or execution, in a relational transaction database. Permanent storage is provided by a dialogue database, which contains all parts of all transactions for an entire application system.

Each of these tools plays an important part in the development of human-computer dialogues under DMS. Further discussion of all except LBE, the language tool, is beyond the scope of this paper.

2.6. ROLE OF THE LBE LANGUAGE TOOL IN DMS AND AIDE

The term *interaction language* will be used herein as a general term for all types of languages which comprise the external dialogue of an application system, the means of communication between the user and the computer. Such languages include typed or spoken command languages, request/response exchanges, function keypads, touch panels, voice I/O, list menus, etc. This language, or external dialogue, is the means by which the human gives commands or queries to the computer and by which the computer responds to or queries the user. This discourse is the exchange of words, phrases, parameterized commands, and other symbols and actions, i.e., the conversation, between a human and a computer.

This human-computer language is composed of two separate parts: the computer's part, which is determined in the software, and the human's part, which is any input a user may give to the system. Thus, it is confusing and indeed erroneous to think of a human-computer dialogue as being only what is displayed to the user by the system. Instead, both the human side and the computer side of the dialogue must be considered in the design of a specific discourse, or external dialogue, of a human-computer system.

Clearly, the human-computer systems which are created under DMS and AIDE have a language for communication with the user. Since the dialogue author is not expected to be either a skilled programmer or a language specialist, an example-based language tool (LBE), to aid in the overall development, specification, and implementation of interaction languages, is being created as a part of AIDE. It will also reduce the amount of communication that must occur between the dialogue author and the application programmer who writes the semantic action routines which execute the specific requests of the user.

There are several aspects to the development of a language tool. The design, specification, and implementation of human-factorable interaction languages for application systems, as well as the parsing and recognition of user inputs in the interaction language, are of obvious concern. One of the most significant problems encountered so far is that of language specification; i.e., how can the commands of an interaction language be specified in a clear, complete, human-understandable notation? In fact, clarity and completeness seem almost mutually exclusive in many language representation schemes. As more information is conveyed in a notation, it typically becomes less human-understandable. A highly formal notation (e.g., Backus-Naur Form) that contains a great deal of information is particularly confusing to the average person without much mathematical background. This issue of information richness versus human factors considerations is one of our main areas of research in the overall design and development of the LBE language tool.

A second important issue arises in language representation; i.e., how can the definition of an interaction language be stored internally so that it is readily retrievable for modification or execution? In fact, this representation provides communication between design-time, when the interaction language is developed, and run-time, when that same interaction language is processed. The values resulting from run-time processing are mapped to language token values which are passed to the computational component at the end of the transaction, to trigger the appropriate semantic action(s). The format and content of this internal representation is another of our research topics [NARAP83].

In summary, interaction languages are specified and their representations constructed at interface design-time, by the dialogue author using AIDE. The run-time transaction executor calls a language executor which accepts, parses, and validates user inputs. Thus, under DMS, a dialogue author can develop an interaction language and it can be processed at run-time, without writing a single line of code!

3. LITERATURE REVIEW ON LANGUAGE SPECIFICATION AND IMPLEMENTATION

3.1. THE NEED FOR A HUMAN-UNDERSTANDABLE LANGUAGE SPECIFICATION

Researchers have realized for some time that severe limitations exist in the notations typically used to specify a language. Representational schemes are generally so information-packed that they become almost unreadable to the average person. Formal language definitions meet with resistance, especially from more pragmatic users, because they are so cryptic and often difficult to understand. The reader of a language definition should be able to quickly grasp the formalism and, from it, extract the information needed. This can be done only through the development of a readable notation that will describe the complete syntax of a language [LEDGH74]. Attention to the design of the language specification, so that the general user can understand it, will help overcome some of the resistance normally encountered in the use of formal language definitions [MARCM76]. Interaction languages will comprise a major portion of the interface which the dialogue author is creating, and the dialogue author is not expected to be a language specialist. The need for incorporation of human factors considerations into both the languages and their specifications is especially important in light of this new role of the dialogue author in producing human-computer systems.

3.2. EXISTING METHODS OF LANGUAGE SPECIFICATION

Several well-known methods of language syntax specification are familiar to computer scientists. However, most of these are used primarily for representation of static programming languages and are inadequate for representation of dynamic interaction languages [JACOR83]. In a static language, the entire sequence of both system and user actions is input, typically as a program. The conditions for all possible alternatives by both system and user must be explicit in the code. In an interactive language, system actions cannot occur until a user input is received and processed. This user dependency (both of content and sequence) associated with interaction languages needs to be present in any specification used for interaction languages. In addition, interaction languages have many features which programming languages do not have. The device from which the language input will come (e.g., voice recognizer, touch panel, function key, mouse), the position at which the input will be received, and the definition of the input echo (e.g., color, possibly no echo at all) are just a few characteristics which are unique to interaction languages. Obviously, none of these is provided for in traditional language specification schemes.

One of the best known systems for representing the syntax of a language is the Backus-Naur Form (BNF) [NAURP63]. While it is fairly understandable once the metalanguage symbols are learned, it has several deficiencies. For example, BNF has no provision for specifying that the declared attribute of an identifier must be compatible with its uses in a program, that the formal and actual parameters must correspond exactly in procedures, and that multiple declarations of the same identifier in a local context are illegal. Any context-sensitive requirements in an interaction language are not representable in BNF notation. In addition, BNF is difficult for humans to understand as well. It is a highly structured, hierarchical metalanguage that results in a "fan-out" problem. That is, non-terminals in an expression can be replaced by more non-terminals through several successive iterations before a terminal symbol is finally reached. This multi-level tree structure is difficult for humans to follow, since by the time the leaves (terminals) are reached, the root (highest level expression) may long be forgotten.

Jacob [JACOR83] presents formal methods for specification of user interfaces at the program module level, emphasizing behavior without committing to a particular internal implementation. Such formal specification techniques have been applied extensively to software but very little to interfaces. Jacob examines two classes of interface specification techniques: those based on state transition diagrams and those based on BNF-type definitions. He concludes that state transistion approaches provide more comprehensible language specifications, because they show surface structure better than BNF does. Language-By-Example, to be introduced in section 5.3, is a specification technique which begins with the specifics of the surface structure (examples of commands in the interaction language being specified) and works toward a general specification of an interaction language.

An attempt to improve on the basic idea of BNF has been attempted using production rules [LEDGH74]. These production rules are basically BNF with notations included to capture some of the requirements not filled by standard BNF, especially regarding context-sensitive issues. Separation of the specification of legal strings from the necessary conditions for string legality is included. The concept of a syntactic environment is introduced to insure that declared identifiers are compatible with their uses. In this syntactic environment, the declared type of each identifier can be derived and therefore its compatibility determined. While adding these features to the syntax representation conveys more implicit information to the user of that language, the resultant language definition seems increasingly complex and confusing. The notation is quite mathematical, there appear to be several confusing symbols, and development of three types of environments (explicitly declared, contextually declared, and implicitly declared) is necessary. In addition, the "fan-out" problem of standard BNF is not solved at all, and the total length of the sets of productions is formidable. The overriding concern should be with user understandability, and production systems do not seem to solve this problem, either.

Several other attempts have been made at least to standardize the numerous variants of BNF notation which seem to abound [WIRTN77, LEDGH80]. Virtually every time a new language is introduced, so is a new variant of BNF with the appropriate modifications necessary to represent that new language. Adoption of a consistent notation for representing static language definitions would greatly improve human understandability, readability, and usability.

One variant of BNF, designed specifically to represent interaction languages rather than static languages, is the multi-party grammar [SHNEB82]. The features which differentiate this extension from standard BNF are the labeling of nonterminals with a party (i.e., either human or computer) identifier, assignment of values to nonterminals when appropriate, and definition of a nonterminal which will match any input string if no other parse of that input is successful. The grammar also permits terminal string input by the user to be fed back in a later part of the dialogue. Other issues involving visual features peculiar to interactive displays are also incorporated.

State transition diagrams (essentially finite state machines) constitute another formal representation frequently used for language definition. As with BNF, this technique has most frequently been used for static languages. State transition diagrams represent the notion of states and the sequencing of transitions among states. Since user inputs provide conditions upon which transitions are made, state transition diagrams seem to be more amenable for specification of interaction languages. A comparison of the advantages and disadvantages of both BNF and state transition diagrams is given in [JACOR83], along with examples of the use of each of these for language specification. The representation of languages using state transition diagrams is extended so that the complete description of a command is possible. Pascal syntax specification has been represented using "railroad track" diagrams to show the relationships between components of that programming language [JENSK74]. While its visual aspect is appealing, it is not powerful enough to represent all the various components and relationships of an interaction language.

Several other formal definition techniques for programming languages also exist. In [MARCM76], a comprehensive description and comparison is made of four of these methods: W-grammars, Production Systems with an axiomatic approach to semantics, the Vienna Definition Language, and Attribute Grammars. The VDL is also discussed in detail in [WEGNP72]. Even a quick scan of these articles will give an indication of the overall complexity of each of these techniques; none of them is easily understandable without considerable study of the underlying concepts, formalisms, and notations. This perhaps explains why none of these four techniques to date (to the authors' knowledge) has been applied to the formal description of interactive systems. Specification techniques which are complicated and not readily comprehensible contradict the basic tenet of the current emphasis on the design of human-factorable human-computer interfaces!

Another approach to interaction language design and representation of interactive computer systems has been introduced in the Command Language Grammar (CLG) [MORAT81]. This formalism creates a framework for describing all aspects of the user interface, not merely the representation of the interaction language itself. Several components (conceptual, communication, and physical) are refined into various levels, each representing a description of the system at the appropriate level of abstraction. The CLG representation is thorough and complete, providing a representation of an interactive system ranging from the user's cognitive level to the system's representational level. However, its very thoroughness introduces a complexity that makes it difficult to use.

A very recent attempt to specify human-computer interfaces has been done using the constructs of a high-level procedural language (i.e., iteration, conditionals, etc.) to represent both the human and the computer actions which must take place [LINDT83a]. The McCabe Metric, or cyclomatic number, is then applied to the resultant algorithms to determine the potential usability of the interface.

3.3. OTHER RESEARCH AREAS IN LANGUAGE REPRESENTATION AND RECOGNITION

Most formal language specifications are used as a tool for describing a language. However, some research is being done in the use of formal grammar description as a prediction tool for use in evaluating alternative human-computer interface designs [REISP81, BLEST82, REISP82]. An "action grammar" is used to describe both cognitive and input actions, which are then converted to a time or error representation. Sentences are created which represent particular tasks or user classes (e.g., "move cursor" = time to move cursor). Then, a set of "prediction assumptions" is compared to the sentences to determine resultant comparative times. Such evaluations using formal languages allow early identification of design inconsistencies which are likely to lead to user errors and allow analysis of the interface for incorporation of human factors principles.

For implementation of languages, a number of automated systems exist. The Unix system contains two automated tools for input language recognition [JOHNS78, JOHNS80]. A program generator called Lex creates lexical analyzers for an input specification language based on the notation of regular expressions. YACC is a parser generator which uses an input specification language that describes the desired syntax for a language. These two tools can be used separately or together to automatically generate major components of a compiler. Another system is LANG-PAK [HEINL75], an interactive language design system for designing and implementing application languages. Input consists of BNF-like statements which represent both the syntax and the semantics of the language being created. This input is processed to create a translation of each user interaction.

A very exciting DMS research product is being developed as the run-time complement to the AIDE design-time system. This is a language executor called DYLEX, which dynamically processes user language inputs a character-at-a-time, based on the design-time definition which the dialogue author has developed [NARAP83].

From this brief review, many of the issues associated with interaction language specification and representation can readily be seen. While numerous formalisms for specification of static programming languages exist, these notations are not adequate for specification of interaction languages. In addition, these notations are generally not human-factorable or easily understandable by someone not completely familiar with them.

4. ISSUES IN DEVELOPMENT OF A LANGUAGE TOOL

Various issues must be considered by the dialogue author in the development of an interaction language. First, of course, is the design of the interaction language itself. If this language is not natural, flexible, and simple to learn, the user will not be able to communicate efficiently and effectively with the system. The dialogue author must also precisely and completely specify the syntax of the interaction language so that it can be recognized by the system at execution time and so that the end-user of the application system can understand and effectively use its syntax. The specification notation should be designed so that the dialogue author will not have trouble using it to create a language specification, and the user of that language will not have trouble understanding it. Finally, if the dialogue author does not have a tool to facilitate interaction language recognition (i.e., processing the language input part of a transaction), a fundamental tenet of DMS may be violated. That is, if the author cannot create a language recognizer, then an application programmer will have to be responsible for writing software to deal with an aspect of external dialogue. Each of these issues will be further discussed below.

4.1. ISSUES IN DESIGNING AND SPECIFYING AN INTERACTION LANGUAGE

4.1.1. Human Factors Considerations in Interaction Language Design

User considerations in the design of an interaction language include the choice of language functions and components, as well as the syntax of an entire interaction. Overall consistency of all interactions within a single system is critical to avoid confusing the user. A simple example is the use of a single space as a delimiter between one keyword and its parameters (e.g., "l Deb" meaning locate the first occurrence of the string of characters "Deb") and use of a slash (/) as the delimiter in another interaction (e.g., "c/Debbie/Debby" meaning change the first occurrence on the current line of the string "Debbie" to "Debby"). This kind of inconsistency can be perplexing to the user of a system in which both of these exist as possible interactions.

Human factors considerations are also important in the form of presentation chosen for interaction languages. Numerous methods of interaction input are possible, including entering the complete keyword, entering an abbreviated form of the keyword, entering enough of the keyword for the system to recognize it and perform command completion, and pressing a function key.

Responses of the system to the user, while not directly related to the interaction language itself, can greatly influence the user's understanding of the system. A powerful, well-designed interaction language is of little use if the responses, and especially the error messages, of the system to the user are ambiguous or meaningless. Poor system messages give little guidance to the user as to what action to use to correct the error, if necessary, and to continue executing.

One important point must be made here. The language tool of AIDE is not aimed at teaching a dialogue author to design an interaction language. It is not intended to be an "expert system" or CAI-type package. Rather, it is intended as a tool to elicit from the dialogue author all information necessary to completely and precisely define an interaction language, without the use of formal, cryptic specification schemes.

4.1.2. Types of Interaction Language Syntax

Human-computer dialogue has two distinct functions; i.e., communication 1) to request information (either from the user or from the computer) and 2) to transmit information (either to the user or to the computer). Interactive programs, in order to execute, must have input (e.g. "name", "SSN", "edit", numerical data) from the user. This input is typically obtained through prompts or queries from the computer which ask the user for specific information. These prompts may vary greatly in form and content, and depend heavily on the type of interaction language syntax of the system being used. Many types of syntactic forms are possible in an interactive human-computer application system, including:

1) request/response form

2) menu form

3) keypad form

4) command string form

5) touch panel form

6) voice form

7) form-filling form

8) any combination of the above syntactic forms

In a request/response dialogue, the computer prompts the user for a specific language input at a particular point in a dialogue. Any textual output from the computer which requests a response from the user is of this form. This type of human-computer dialogue is typically more appropriate for less skilled system users. The computer is basically in control of this type of dialogue, since there is generally only a small number of predetermined acceptable responses that a user can give to any particular query. Examples of systems employing such languages include an airline reservation system (e.g., system requests "Departure airport:" and user responds "Blacksburg") and a university's student records system (e.g., system requests "Please type in student id" and user responds "123-45-6789").

A menu is a special form of request/response syntax. It combines both the transmission of the prompt with the request for information by displaying possible options and then asking the user to choose one.

Another special type of request/response syntactic form is a function keypad. This is typically an auxiliary keypad (i.e., a programmed function keypad which is not a part of the main input keyboard) whose keys have prespecified functions that are invoked by pressing the correct key. Within a single application system, the same physical function key may denote different semantic functions at different times during system execution. A hierarchy of keypads can provide a powerful, effective interface for a human-computer application system. This form of syntax has been chosen for the high-level AIDE interface, in order to encourage consistency among the AIDE tools and functions, as well as to provide the dialogue author with a simple, efficient, and generally parameterless interface. In addition, it diminishes the need for modality among the tools of AIDE. This will be further discussed in the section on the LBE language tool interface.

Both menu and keypad syntactic forms are good for inexperienced users, since all possible options are presented at all points in time. A keypad, because of its brevity and conciseness, is also a good interface for expert users, who typically do not like the verbosity of a menu or request/response syntax. Additionally, in light of recent advances in their technology, both voice I/O and touch panel I/O are particularly appropriate adjuncts to menu and keypad syntactic forms.

With a command string syntactic form, the user generally does not receive textual prompts or queries from the system, but spontaneously enters a specific command with a predetermined format, in order to accomplish a particular system action at a particular point in a dialogue. In this type of dialogue, the user is more in control, since choices may be made from a large number of appropriate inputs at many points in the dialogue. This type of interaction is usually easier for more experienced users, since the inputs may be more parameterized than in a request/response language. Examples of a command string syntax interaction include a text editor (e.g., user enters "del 3" to delete three lines) and the GENIE air-traffic-control system [LINDT83b], used for human factors experimentation (e.g., participant enters "321 climb and maintain angels 40" which causes aircraft number 321 to climb to 4000 feet and hold there).

A form-filling syntactic form elicits input from the user by allowing movement among fixed fields on the screen. Once a given field is selected, the user can then enter the appropriate information. This syntactic form might be used in an inventory control system, where "part number", "customer name", "quantity required", and "delivery date" are separate fields that must be "filled in" to complete a customer order.

Finally, an interactive human-computer system whose interaction language is comprised exclusively of one of these syntactic forms is relatively rare (with the possible exception of command string syntax interfaces). The human-computer interface of a single system frequently consists of a combination of any and all possible interaction syntax types. For example, a predominantly request/response language will often include some menus and a command string language and may make use of a special function keypad.

When the syntax of a interaction language is embodied in, for example, a keypad, interactive specification of that syntax must be done for each key. That is, simply developing the keypads that comprise an application system interface and giving labels (functions) to each key produces only the display part of the transactions which the keypads represent. The language input specification must also be developed so that, at run-time, the language executor can process the user's keypad key selections. A token value must be defined for each possible language input. As a result of a user input, this token value is passed from the dialogue component to the computational component, to invoke the appropriate semantic action routine(s). Until the language input part of a transaction is developed, only its display can be activated at run-time and the user's input cannot be accepted, validated, or recognized. As a consequence, the token value(s) associated with that language input cannot be determined, so no semantic actions can be taken. The language tool allows the dialogue author to completely define the language input part of each transaction by specifying token values for each user input. Many different syntactic forms can map to the same token value, making a transaction independent of device type, specific syntax, etc.

4.1.3. Specification of Syntax

In order to specify the syntax of an interaction language, a metalanguage (i.e., a language that is used to define other languages) is required. One of the most confusing aspects of most metalanguages is the fact that the metalanguage itself contains symbols that can also be contained in the language it is being used to represent. A simple example is shown in the BNF representation of conditional statements in the following way:

```
          <comparison> ::= <operand> <conditional_operator> <operand>
             <operand> ::= <integer> | <identifier> | (<expression>)
<conditional_operator> ::=   = | <> | < | <= | >= | >
```

Obviously, the use of angle brackets as the metalanguage delimiter for category names within the language is confusing, since in the definition of <conditional_operator> the < and > represent "less than" and "greater than" in the language being defined. This same issue can even arise outside the specification of a formal language syntax, as in the computer instruction: Type "QUIT" to quit. The user is unsure, upon seeing this for the first time, whether to type "QUIT", which is, after all, what is shown, or simply QUIT, without the quotation marks.

The only way to alleviate this confusion is to define the symbols of the metalanguage so that they are disjoint from the characters of the language which the metalanguage is being used to represent. An immediate constraint arises in the definition of an interaction metalanguage to define other interaction languages. All characters in the metalanguage, if they are to be displayed on the CRT, must be in the ASCII character set. A character not in the ASCII set cannot be used as an interaction metalanguage symbol. But at the same time, the ASCII set is the domain from which the symbols for virtually all interaction languages are drawn. Thus, a notational exclusion between the two is impossible. One weak solution is to define the symbols of the metalanguage to be the most rarely used characters of the language itself, to avoid confusing the two as much as possible.

There are several other possibilities for overcoming this problem, however, if a simple graphics system is available. Such features as color, highlighting, blinking, and graphical shapes can be used to insure that the metalanguage is disjoint from the interaction language it is defining. For example, components of the metalanguage might be displayed in red, while components of the interaction itself are displayed in blue. Similarly, metalanguage symbols might be special graphical symbols created especially for use in the metalanguage, while the language symbols are the standard ASCII character set.

An initial attempt to classify the various components of interaction languages has led to two general observations. First, all interaction languages, regardless of syntactic form, appear to be comprised of two basic entities: *tokens* and *delimiters*. In command string languages, tokens and delimiters occur alternately in a command, and a command may begin and end with either a token or a delimiter. In keypad-driven languages, only a single token, without a delimiter, results from each key selection. Secondly, interaction languages appear to have features that are analogous to the context-free features of programming languages, as well as those that are analagous to the context-sensitive features. The context-free components are those that can be chosen and represented independently of the context in which they are to be used. That is, there are no limitations on the choice and use of these components at any time in a interaction. Context-sensitive components, however, are those whose choice and representation are dependent upon the context in which they are to be used. That is, their choice is constrained by a choice that has already been made, or even one that will be made later.

A simple example will help to clarify some of these issues. The specification of a simplified "change" command for a text editor might be represented as follows:

$$\text{Change} \begin{bmatrix} / \\ ? \\ , \end{bmatrix} \text{string1} \begin{bmatrix} / \\ ? \\ , \end{bmatrix} \text{string2} \left\{ \begin{bmatrix} / \\ ? \\ , \end{bmatrix} \quad * \right\}$$

This cryptic line of syntax specification is filled with numerous things that must be understood by someone attempting to learn the syntax of the "change" command. In this particular specification scheme, the first token, "Change", means that a "C" is required (but it may be either upper- or lower-case!) and "hange" is optional. The left-most set of elongated brackets containing three characters vertically positioned means that any one of those three characters may be chosen as a delimiter at this point in the command. However, the user must also know that, because this is the "change" command, it is implied that the character chosen as a delimiter may not be an element in either "string1" or "string2". This is not shown explicitly in any form in this representation. The second token, "string1" itself can consist of any sequence of characters, so long as the chosen delimiter is not one of them. Next is another set of elongated brackets containing the same three delimiter options that were given earlier. But now the implicit information is that the character chosen as the first delimiter must again be used as a delimiter at this point in the command. That is, if "?" was used for the first instance of a delimiter, then "?" must be chosen as the delimiter each time a delimiter selection is required in the same instance of the command. Thus, even though all three characters are given here as options, the choice as to which must be used has already been made, if "?" was selected earlier. This consistency requirement is not explicitly expressed in any way in this syntax specification. Next, the third token, "string2", is encountered, with the implication again being that it can consist of any combination of characters as long as the delimiting character is not in it. Finally, curly braces around a closing delimiter and the fourth token, the "*" symbol, are intended to indicate that this entire portion of the command is optional. The "*" represents a global change throughout the entire file currently being edited. So this command without a "*" effects changes to only the current line, and the third delimiter becomes optional. Including "*" at the end of the command line effects changes to all occurrences throughout the file, and the third delimiter (the one preceding the "*") is required. Not completely clear here is the fact that if "*" is used, then a delimiter must also be used, and again, that this delimiter must be the same as the first two. Finally, it is also unclear from this representation whether "string 1" will replace "string2" or vice versa.

4.1.4. Categories of Interaction Language Components

In order to resolve these kinds of ambiguity in interaction language syntax specification, an initial categorization of all interaction language components has been determined. This categorization, most applicable to the command string syntactic form, contains categories for both context-free and context-sensitive features, both of which may be either optional or required. If a component is an optional choice, its use in a command string is not required. For example, an AREA command, which displays a portion of a file above and below the current line might be entered as simply "AREA" or as "AREA 5". In the first case, the system default is used for the number of lines to be displayed, while in the second case, five lines above and five lines below the current line are displayed. A special case of optionality is that of equivalent options for which two different choices for a language input component have exactly the same meaning. For example, "DELETE 3" and "DELETE +3" would accomplish exactly the same thing, that is, deletion of three lines. Required components, on the other hand, must be included in the command string every time it is used. The command string name, for example, is obviously a required component; without it the command would be unrecognizable. Any of the categories discussed below can be either optional or required.

TOKENS -- These are the basic entities of which an interaction language is comprised. It is the run-time processing of tokens which determines what semantic actions will occur as a result of user input. A token may have context-free or context-sensitive features, depending on whether there are constraints on its choice. A simple example would be the valid inputs for "departure city" and "destination city" in an airline reservation system "reserve" command. The valid choices for departure city and destination city very likely come from the same list of cities, but once the departure city has been chosen, then the destination city must be a different city. Thus, departure city exhibits context-free behavior, but destination city has context-sensitive features. Any token may be defined in one of two ways: as a constant or exact representation, or as a variable-name or descriptor rule.

1) Constants -- These are those tokens of an interaction language that are fixed both positionally (i.e., the location within a command) and literally (i.e., the exact characters that must be entered) in a command. Thus, a constant in a command string must be entered exactly as it appears in that command string's representational syntax. Reserved words (of which the command name itself is a special case) are typical examples of constants. Human factors considerations such as choice of a mnemonically meaningful command name and choice of a reasonable abbreviation (if allowable) are not directly related to the issue of language specification. In any case, the command name and its allowable abbreviations must be clearly represented. Inclusion of presentation mode

(e.g., typed command string name or abbreviation, function key, command string completion, etc.) may also be desirable. Another example of a reserved word is in the command "MOVE x TO y". Here, "TO" is an alphabetic constant that must always appear in that form in that place in the command string.

2) Variable-names -- These are tokens of an interaction language that are user-supplied alpha-numeric strings. Thus, a variable-name in a command string can represent any string of characters the user will enter, as long as that string is comprised of characters that are legal for that particular variable string. The variable-value is the actual string of characters that is entered by the application system user at application system run-time. There are two most common types of variables: text variables and numeric variables.

   a) Text variables are comprised of any printable characters from the entire alphabet of the language. In the "change" example given earlier, "string1" and "string2" are variables of this sort. They may consist of any combination of characters (alphabetic, numeric, or special) from the total alphabet of the language.

   b) Numeric variables represent a numeric value in a command. They may be either a single digit (0, 1, 2, ... 9) or a combination of single digits, or they may be within a prespecified, allowable range (e.g., 0 to 100). For example, a typical format for a DELETE command consists of the keyword DELETE followed by a positive number which indicates the number of lines to be deleted. Thus, DELETE 3 would delete the current line and the next two lines. This might be represented in a language specification as DELETE N. Here, N is a variable that can change as needed to accomplish deletion of the proper number of lines. An example of a prespecified range might be in the specification of possible headings for an aircraft in an air-traffic-control simulation. The possible headings are in the range of 1 degree to 360 degrees, and this restriction must be indicated in the definition of the command.

DELIMITERS -- These are the characters (both printable and non-printable) that are used to separate tokens of a command. Two issues must be addressed here. The first one is the choice of delimiters for the metalanguage itself. These metasymbols are frequently denoted by such characters as { }, < >, [ ], etc. The problem discussed earlier of making these disjoint from the language being represented is perhaps more apparent here than in any other instance. The second issue is choice of a delimiter within a command string in the language itself, that is, the character that the user will actually enter to the system as a delimiter. The choice of a delimiter for a command string seems simple enough, as well as the representation of possible choices. But string delimiters may have context-sensitive features under at least two conditions: 1) if the character chosen as the first delimiter must also be used as the second delimiter and 2) if a delimiter cannot be in the string it is delimiting. Representation for the context-sensitive aspects of delimiter choice should be included in the language definition.

LISTS -- These are used to represent all possible alternative choices that can be made for an entity (either a token or a delimiter) of a command string. A list, in general, is a set of constants and/or variables that denote a specific entity in the language. For example, a list of constants that are alternative tokens that represent the "change" command might consist of the following: c, change, C, CHANGE. Context-sensitivity may arise here if limitations are imposed on the choice that can be made. For example, in the "change" command, the choice of string delimiters from the list shown in the syntax of section 4.1.3 (/ or ? or ,) is constrained by the requirement that the delimiters be disjoint from any characters in the strings. Thus, use of a "/" is perfectly allowable as a delimiter for the string "July 15, 1982", but not for the string "7/15/82". Not all list choices are context-sensitive, however. For example a NEXT command string might be allowable as simply NEXT (meaning move to the next line), or NEXT -N (meaning move up N lines), or NEXT * (meaning move to the bottom of the file). In this instance, the choice following NEXT can be made from the list consisting of a signed or unsigned integer, or a "*". A blank space denotes a default of 1. Any of these options may be chosen, without any context constraints, depending on what the user wishes to do.

CONDITIONALLY OPTIONAL CHOICES -- This involves making a choice of an input character based on specific conditions that may or may not exist at any given time in a command. For example, * is generally used to represent all occurrences. Again, using the "change" command, a change of all occurrences of the string "SP" to the string "S.P." might be done by typing the following:

```
c/SP/S.P./*
```

The third '/' would not be needed, however, if the * were not used in the command. Thus, the use of the third delimiter (/) is conditionally optional, depending upon use of the subsequent *. This makes the use of the third delimiter context-sensitive.

DEFAULTS -- These are values which are automatically system-supplied if the user does not supply a value for a required component at run-time. For example, "1983" might be the default value for the token "year". In a form-filling type of syntax, default values can be entered a priori by the system and made visible to the user.

REPETITION -- This is the number of times a particular component of a command can be repeated. As in programming languages, repetition specifications should allow for the use of any number of occurrences of the component. For example, a string can consist of concatenation of one or more occurrences of a single symbol (either alphabetic, numeric, or special). In other cases, it may be necessary to specify the exact number of times a component is to be repeated, for instance, a string consisting of exactly five characters.

DEVICE DETERMINATION (or SYNTACTIC FORM) -- This is the device (e.g., keyboard, function key, touch panel, or voice recognizer) which will be used for the language input. Including this in the specification may be unnecessarily confusing, especially if there is more than one input device which might be used for a single interaction. However, alternative I/O devices are becoming a part of the human-computer interface and should be accepted and handled accordingly.

LANGUAGE ATTRIBUTES -- These are the possible alternatives for such interaction language features as echo/no echo, carriage return/immediate command, spelling correction, error checks, token completion, input color, etc. Many such features exist and should be completely specifiable by the dialogue author as appropriate at all levels in a given application system.

Two other requirements result from context-sensitive features. One is the "look-ahead" requirement, in which what is to be chosen from among several possible options is dependent upon what will come later in the command. For example, the choice of the first delimiter of a string depends upon its not being an element in that string. But at the time when that first delimiter must be chosen, the elements of the string have not yet been encountered in the command. Thus, the user must "look ahead" in the string to determine what its characters are, so that the delimiter choice can be made to be disjoint from them. The other requirement is a "history" requirement, in which the choice to be made depends upon a choice that was made earlier. For example, the choice for the second delimiter of a string must be the same as the first delimiter. In both these cases, the time order of decisions is cognitively different from the spatial order in which the entities are entered.

In order to develop a syntactical language specification that is complete, a symbolic notation must be defined for each of the specific categories and features discussed above. In order to make the specification as human-readable and human-understandable as possible, the symbols must somehow be meaningful to the user. Choice of such metasymbols is not an easy task, particularly while trying to keep them as disjoint as possible from the languages they will be used to define.

4.2. ISSUES IN AUTOMATING LANGUAGE SPECIFICATION AND RECOGNITION

4.2.1. Role of the Author in Interaction Language Implementation

Traditional methods of recognizing computer languages necessitate the design and implementation of software to parse the language. Translators, compilers, and interpreters are the software components for programming language recognition. For interaction languages, state table-driven keyword analyzers and finite state machine parsers are frequently used. But not all application programmers have the knowledge required to write language recognizers, and certainly a dialogue author is not expected to have this type of programming ability. Thus, much research effort has gone into the development of automated tools for language design and implementation. Such systems as YACC, LEX, and LANG-PAK, discussed earlier, allow nonspecialists in languages to define and implement programming languages.

Because DMS will be used to create human-computer systems, the interfaces for these systems will consist of a human-factorable dialogue, created by the dialogue author. The inputs to the system by the user constitute external dialogue and, as such, should be dealt with by the dialogue author. But the introduction of the dialogue author into the human-computer system development process creates some unique issues in interaction language implementation. The LBE language tool provides the dialogue author with the means for specifying and representing an interaction language to the system, so that inputs in that language can be executed.

4.3. A MODEL FOR INTERACTION LANGUAGE SPECIFICATION

A comprehensive model for interaction language specification will address the special needs of the dialogue author while designing and implementing interaction languages, as well as the connection to the execution-time environment of the language. Figure 3 shows such a model, one which views human-computer communication as human-to-human communication, offset in time. The real communication here is between the dialogue author and the end-user. Figure 3 shows the path of this communication; the stored internal representation provides the time offset (somewhat analogous to a complicated store-and-forward message system). This communication link between the author and the user is the composition of transformation function (the arcs in Figure 3) that connect their conceptual models. There are five separate submodels of the interaction language being specified, each with its own (often different) specification and representational needs:

1) *Dialogue author's conceptual model* -- This is a mental design-time model of an interaction language which is formulated by consideration of the application system needs as a result of the author's early participation in the requirements analysis and design phases of the application system's life-cycle. The specific language requirements are transmitted by the programmer in an internal dialogue requirements specification document. The conceptual model is completed by the dialogue author's knowledge of human factors principles and, possibly, advice from an interaction language "expert" program.

2) *Language specification model* -- The dialogue author uses the AIDE/LBE interface to specify the interaction language design to DMS. As Language-By-Example, using its built-in specification model, guides the dialogue author, the inter-

action language is instantiated for specific language inputs and generalized into complete language specifications by LBE. The dialogue author uses this model to convey the syntax of each specific language input to the underlying system, which then creates an interpretable, storable representation or definition of that language input. A very important requirement for this specification model is that it be highly human-factorable and human-understandable, as well as information-rich.

3) *Internal representation model* -- This is the interpretable, stored representation which is the product of the AIDE/LBE tools applied to the language specification model. When the design-time specification is given to LBE, a corresponding language input definition is stored in a database as the internal representation of that language component. Then, when the completed application system is operational and is being given interaction language inputs from a user at run-time, these definitions are accessed as needed to recognize the user inputs, are immediately processed, and the corresponding semantic routine(s) are called and executed. This model is stored in a data structure called token tables, and is not necessarily in a human-readable form. The presentation of token table formats and their interpretation is beyond the scope of this paper. Again, the reader is referred to [NARAP83].

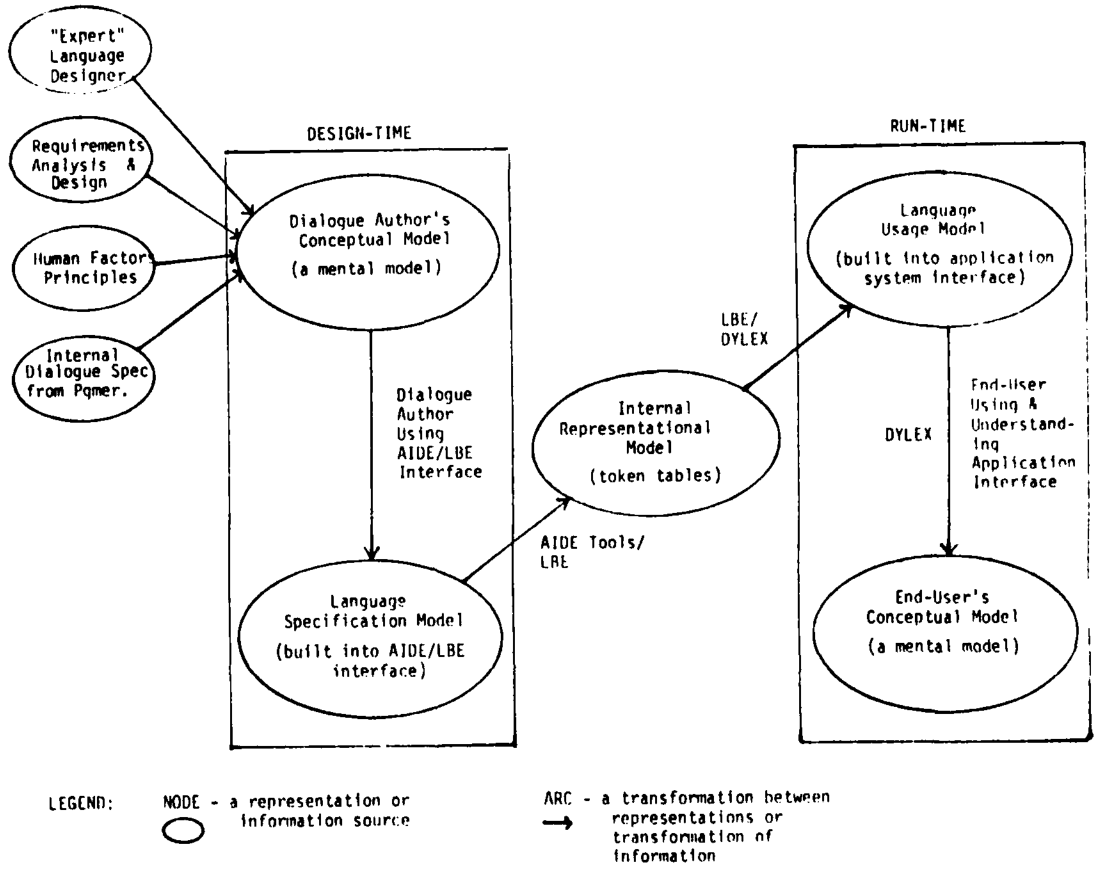

"Expert" Language Designer
Requirements Analysis & Design
Human Factors Principles
Internal Dialogue Spec from Pgmer.
DESIGN-TIME
Dialogue Author's Conceptual Model (a mental model)
Dialogue Author Using AIDE/LBE Interface
Language Specification Model (built into AIDE/LBE interface)
AIDE Tools/ LBE
Internal Representational Model (token tables)
LBE/ DYLEX
RUN-TIME
Language Usage Model (built into application system interface)
DYLEX
End-User Using & Understanding Application Interface
End-User's Conceptual Model (a mental model)
LEGEND:
NODE - a representation or information source
ARC - a transformation between representations or transformation of information

Figure 3. A Model of Communication between Author and User

4) *Language Usage Model* -- This is the descriptive model which is seen by the application system user at application system run-time. It is effectively a training and/or help-type representation of the interaction language syntax for each allowable input. The user can access this specification as

needed, and it may consist of an on- or off-line user's manual. This level should also be human-factorable and information-rich.

5) *End-user's conceptual model* -- This is the application user's own mental model of the interaction language. At execution time for the application system, the language executor, called by the transaction executor, uses the internal model to recognize an input of the interaction language. It returns token values to the computational component to invoke a procedure to produce the appropriate semantic action. As these interactions take place, the user builds a model used to understand the use of the system. This model includes syntax and semantics, as well as personal strategies for the use of commands. If this model provides a reasonable match to the author's conceptual model, the two have communicated well.

5. LANGUAGE-BY-EXAMPLE (LBE): VERSION I

5.1. GENERAL GOALS

The primary functions of the LBE language tool of AIDE are as follows:

1) to specify interactively the syntax of the interaction language of a human-computer system at system design-time,
2) to store interaction language definitions in an internal representation accessible at run-time,
3) to facilitate automation of the recognition (lexical analysis and parsing) of that syntax at application system run-time,
4) to provide a mapping from user's "raw" input to common (normalized) token values, independent of syntactic form or device type,
5) to provide a mapping of token values from any user input to the appropriate semantic action(s).

The language tool provides these functions for the syntax types discussed earlier (i.e., request/response, menu, keypad, command string, and combinations of these). It deals directly only with the syntax and therefore the recognition of an interaction, but not with the semantic action routines that cause a user request to be executed. These semantic routines, and therefore their execution, are not a part of the language tool, nor of the dialogue component of the application system. They are called as a result of language input recognition provided by the run-time language input executor, DYLEX, and are executed within the computational component. The semantic action routines are specified during system design and are implemented by an application programmer.

Recent trends in language design and specification have been toward the inclusion of semantics in the language specification itself. While this is useful for programming languages, we find it to be the wrong direction to take for interaction languages. In LBE, and indeed throughout DMS, the semantics of interaction languages are kept separate from their lexical and syntactic considerations. This separation is dictated by the principle of dialogue independence. As discussed earlier, the computational programmer is responsible for providing the semantic functions but is not concerned with the lexical or syntactic forms of the language inputs which will invoke those functions. These latter forms are decided upon by the dialogue author quite independently of the computational design. In fact, as a result of human factors testing and iterative refinement procedures, the lexical and syntactic details for a given computational function can be subject to considerable change as the interface design evolves.

The DMS philosophy espouses that any token value which is passed from the dialogue component to the computational component at run-time shall be a lexically and syntactically correct, valid value. This means that all user input must be not only accepted, but syntactically verified, in the dialogue component of an application system. If a user's input is not valid, the dialogue component must continue to elicit more input until a valid one is received. This ensures that the resultant token value is usable by the computational component, and that the application programmer does not have to implement input validation.

5.2. DESIGN AND IMPLEMENTATION

Because the AIDE interface consists of a two-screen workstation for the dialogue author, the LBE interface is also comprised of two screens: a command screen and a dialogue screen. The LBE language tool interface itself is a keypad-driven interaction language, with most commands being given by pressing a specific function key. The dialogue screen contains the components of the interaction language being created, while the command screen is used for prompting and echoing and for a few non-keypad-driven commands. It also contains a labelled outline of the currently active LBE keypad. This display changes as appropriate whenever a different level of LBE is entered, causing a change in the active LBE keypad functions.

5.3. LANGUAGE-BY-EXAMPLE

Because of the numerous difficulties with traditional and current methods of language specification, *Language-By-Example* (*LBE*) has been developed as a means of specifying the definitions for the language input part of a transaction. It is a powerful method for defining command string syntactic forms, yet is applicable also to the simpler forms such as keypad and menu. For the definition of a language input, LBE obviates the need for a cryptic, formal notation by providing the dialogue author with an *example-based specification interface*. Through a series of system queries and dialogue author responses, the dialogue author is guided through the definition of an interaction language. By starting with a specific example and working toward a general definition, LBE follows the human cognitive problem-solving process. This principle, which allows the user to work with concrete objects rather than abstractions, is also applied successfully in the Xerox Star interface [SMITD83].

In the more complicated command string syntactic form, the author is asked to enter an example of the command string to be defined (e.g., "locate abc", which means "locate the next occurrence of string 'abc' "). Then, by moving the cursor, the dialogue author delineates each entity (i.e., token or delimiter) of which that command string is comprised. As each entity is delineated (confirmed by reverse video highlighting of the current entity), the dialogue author responds to system queries, both with keypad keys and typed input. These responses comprise all information which is necessary to specify a full and general definition of that entity, as well as its relationship to the other entities in the command. Complete definition of all entities comprises all information which is necessary to fully define that command string language.

In the simpler syntactic forms of keypad and menu, the amount of information which the dialogue author must provide is much smaller, but LBE prompts in the same way as for command strings, and the author responds with either keypad keys or typed input, as appropriate.

A more complete discussion of LBE is beyond the scope of this paper. It is presented here simply to serve as an introduction to a new, alternative approach for the specification and representation of interaction languages.

5.4. SAMPLE SCENARIO OF LBE

This section presents a sample scenario to demonstrate the use of LBE by the dialogue author at language input design-time. This scenario uses the design of two command string commands to demonstrate the components, principles, and relationships which LBE is capable of handling. The first command string is the "locate" command, which might be a text editor command to locate an instance of a specific string in a file. The second command, the "reserve" command, might be part of an airline reservation system and would be used to indicate that a flight was to be reserved between a given departure city and destination city. This scenario does not illustrate all possible features of LBE, but rather gives an overview of its capabilities.

In the following scenario,

"D:" indicates dialogue author inputs,

"S:" indicates system (AIDE/LBE) responses and queries, and

"K:" indicates currently active keys on auxiliary keypad.

Words enclosed in < > (e.g., <ECHO>) indicate that the dialogue author pressed the indicated key on the currently active auxiliary keypad. Characters which are $\boxed{\textit{italicized}}$ and enclosed in a box indicate reverse video of those characters.

SCENARIO: Dialogue author, within AIDE, has indicated by an auxiliary keypad key selection a desire to <DEVELOP COMMAND STRING LANGUAGE>

S: You can now develop a command string language

S: Develop language attributes for this transaction

K: TOKEN COMPLETION, NO TOKEN COMPLETION

D: <TOKEN COMPLETION>

K: NO ERROR CHECK, ERROR CHECK WITH ECHO OF ERRONEOUS CHARS, ERROR CHECK WITHOUT ECHO OF ERRONEOUS CHARS, SPELLING CORRECTION

D: <ERROR CHECK WITH ECHO OF ERRONEOUS CHARS>

K: ECHO, NO ECHO

D: <ECHO>

S: Enter interaction name, followed by CR

D: command

S: Enter example of command string to be defined, followed by CR

D: locate abc

S: Indicate first token in the sample command string, the command name

D: $\boxed{\textit{locate}}$ abc

S: Do you wish to change the transaction attributes for this interaction?

K: YES, NO

D: <NO>

S: Press token type

K: EXACT REPRESENTATION, RULE, LIST

D: <EXACT REPRESENTATION>

S: Enter token value for this token, followed by CR

D: locate

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default value for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Indicate next entity in the sample command string

D: locate$\boxed{\ }$abc

S: Press delimiter type

K: EXACT REPRESENTATION, RULE

D: <RULE>

S: Press delimiter rule type

K: ALL BLANKS, PUNCTUATION, NON-STANDARD

D: <ALL BLANKS>

S: Enter smallest number of blanks allowable, followed by CR

D: 1

S: Enter largest number of blanks allowable, followed by CR

D: 3

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default value for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Enter interaction name, followed by CR

D: loc_string

S: Indicate next entity in the sample command string

D: locate $\boxed{abc}$

S: Do you wish to change the transaction attributes for this interaction?

K: YES, NO

D: <NO>

S: Press token type

K: EXACT REPRESENTATION, RULE, LIST

D: <RULE>

S: Press token rule type

K: NUMERIC ONLY, ALPHA ONLY, ALPHA-NUMERIC, ANY CHARACTER, RESTRICTED COMBINATION

D: <ANY CHARACTER>

S: Enter shortest string length allowed, followed by CR

D: 0

S: Enter longest string length allowed, followed by CR

D: 80

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <OPTIONAL>

S: Are there any constraints on this entity?

K: YES, NO

D: <NO>

S: Press command termination type

K: CARRIAGE RETURN, NO CARRIAGE RETURN

D: <CARRIAGE RETURN>

K: EXIT AIDE, DEVELOP ANOTHER COMMAND FOR CURRENT TRANSACTION

D: <DEVELOP ANOTHER COMMAND FOR CURRENT TRANSACTION>

S: Enter example of command string to be defined, followed by CR

D: reserve (roanoke, atlanta)

S: Indicate first token in the sample command string, the command name

D: $\boxed{\underline{\textit{reserve}}}$ (roanoke, atlanta)

S: Do you wish to change the transaction attributes for this interaction?

K: YES, NO

D: <YES>

S: Develop interaction language attributes

K: TOKEN COMPLETION, NO TOKEN COMPLETION

D: <NO TOKEN COMPLETION>

K: NO ERROR CHECK, ERROR CHECK WITH ECHO OF ERRONEOUS CHARS, ERROR CHECK WITH NO ECHO OF ERRONEOUS CHARS, SPELLING CORRECTION

D: <SPELLING CORRECTION>

K: ECHO, NO ECHO

D: <ECHO>

S: Press token type

K: EXACT REPRESENTATION, RULE, LIST

D: <EXACT REPRESENTATION>

S: Enter token value for this token, followed by CR

D: res

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default value for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Indicate next entity in the sample command string

D: reserve$\boxed{(}$roanoke, atlanta)

S: Press delimiter type

K: EXACT REPRESENTATION, RULE

D: <RULE>

S: Press delimiter rule type

K: ALL BLANKS, PUNCTUATION, NON-STANDARD

D: <PUNCTUATION>

S: Enter smallest number of blanks allowed before punctuation character, followed by CR

D: 0

S: Enter largest number of blanks allowed before punctuation character, followed by CR

D: 4

S: Press punctuation type

K: SINGLE EXACT REPRESENTATION, LIST OF REPRESENTATIONS

D: <SINGLE EXACT REPRESENTATION>

S: Enter smallest number of blanks allowed after punctuation character, followed by CR

D: 0

S: Enter largest number of blanks allowed after punctuation character, followed by CR

D: 0

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Enter interaction name, followed by CR

D: departure_city

S: Indicate next entity in the sample command string

D: reserve ($\boxed{\textit{roanoke}}$, atlanta)

S: Do you wish to change the transaction attributes for this interaction?

K: YES, NO

D: <NO>

S: Press token type

K: EXACT REPRESENTATION, RULE, LIST

D: <LIST>

S: Press token type for list entities

K: EXACT REPRESENTATIONS ONLY, RULES ONLY,
EXACT REPRESENTATIONS & RULES

D: <EXACT REPRESENTATIONS ONLY>

S: Enter exact representation, or CR if no more

D: roanoke

S: Enter token value for this token, followed by CR

D: roa

S: Enter exact representation, or CR if no more

D: atlanta

S: Enter token value for this token, followed by CR

D: atl

S: Enter exact representation, or CR if no more

D: paris

S: Enter token value for this token, followed by CR

D: par

S: Enter exact representation, or CR if no more

D: CR

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default value for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Indicate next entity in the sample command string

D: reserve (roanoke$\boxed{,\ }$atlanta)

S: Press delimiter type

K: EXACT REPRESENTATION, RULE

D: <RULE>

S: Press delimiter rule type

K: ALL BLANKS, PUNCTUATION, NON-STANDARD

D: <PUNCTUATION>

S: Enter smallest number of blanks allowed before punctuation
character, followed by CR

D: 0

S: Enter largest number of blanks allowed before punctuation
character, followed by CR

D: 1

S: Press punctuation type

K: SINGLE EXACT REPRESENTATION, LIST OF REPRESENTATIONS

D: <SINGLE EXACT REPRESENTATION>

S: Enter smallest number of blanks allowed after punctuation
character, followed by CR

D: 0

S: Enter largest number of blanks allowed after punctuation
character, followed by CR

D: 1

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Enter interaction name, followed by CR

D: destination_city

S: Indicate next entity in the sample command string

D: reserve (roanoke, $\boxed{\textit{atlanta}}$)

S: Do you wish to change the transaction attributes for this interaction?

K: YES, NO

D: <NO>

S: Press token type

K: EXACT REPRESENTATION, RULE, LIST

D: <LIST>

S: Press token type for list entities

K: EXACT REPRESENTATIONS ONLY, RULES ONLY, EXACT REPRESENTATIONS & RULES

D: <EXACT REPRESENTATIONS ONLY>

S: Enter exact representation, or CR if no more

D: roanoke

S: Enter token value for this token, followed by CR

D: roa

S: Enter exact representation, or CR if no more

D: atlanta

S: Enter token value for this token, followed by CR

D: atl

S: Enter exact representation, or CR if no more

D: paris

S: Enter token value for this token, followed by CR

D: par

S: Enter exact representation, or CR if no more

D: CR

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default value for this entity?

K: YES, NO

D: <NO>

S: Are there constraints on this entity?

K: YES, NO

D: <YES>

S: Press constraint type

K: NUMERIC VALUE, STRING VALUE, TOKEN, DELIMITER

D: <TOKEN>

S: Enter interaction name which defines contraining token, followed by CR

D: departure_city

S: Press relationship between constraining token and this token

K: <, >, =, <=, >=, NOT =

D: NOT =

S: Indicate next entity in the sample command

D: reserve (roanoke, atlanta$\boxed{)}$

S: Press delimiter type

K: EXACT REPRESENTATION, RULE

D: <RULE>

S: Press delimiter rule type

K: ALL BLANKS, PUNCTUATION, NON-STANDARD

D: <PUNCTUATION>

S: Enter smallest number of blanks allowed before punctuation
character, followed by CR

D: 0

S: Enter largest number of blanks allowed before punctuation
character, followed by CR

D: 1

S: Press punctuation type

K: SINGLE EXACT REPRESENTATION, LIST OF REPRESENTATIONS

D: <SINGLE EXACT REPRESENTATION>

S: Enter smallest number of blanks allowed after punctuation
character, followed by CR

D: 0

S: Enter largest number of blanks allowed after punctuation
character, followed by CR

D: 1

S: Is this entity required or optional?

K: REQUIRED, OPTIONAL

D: <REQUIRED>

S: Is there a default for this entity?

K: YES, NO

D: <YES>

S: Enter the default value, followed by CR

D:)

S: Are there constraints on this entity?

K: YES, NO

D: <NO>

S: Press command termination type

K: CARRIAGE RETURN, NO CARRIAGE RETURN

D: <CARRIAGE RETURN>

K: EXIT AIDE, DEVELOP ANOTHER COMMAND FOR CURRENT TRANSACTION

D: <EXIT AIDE>

S: This completes this session with AIDE.

These two examples purposefully ignore the development of the display and/or confirmation parts of the interactions being created for these command strings. Explanation of AIDE for this purpose is well beyond the scope of this paper. This sample scenario has been presented without accompanying explanation simply to give the reader an idea of how the dialogue author would interact with LBE to define two different command strings.

6. SUMMARY, CONCLUSIONS, AND FUTURE WORK

One area of formal languages in which little research has been done is that of specification and representation of interactive languages and interfaces for human-computer systems. Existing representational schemes (e.g., BNF, state transition diagrams) are usable for static languages, in which all actions are predetermined by the software. But in interaction languages, the added dimension of the human makes interaction highly varied and its representation therefore more difficult. Users of these systems must have a readable, understandable, and complete specification notation for interaction language syntax, in order to use the system effectively. Also, the addition of a dialogue author to the system development cycle emphasizes the need for a human-factorable specification scheme.

This paper has presented numerous issues in interaction language specification and representation. It has also presented a taxonomization of the components and features of interaction languages, various syntactic forms, and a model for interaction language specification. Finally, a new approach, Language-By-Example, has been presented as a simple, yet effective alternative to traditional methods of defining interaction languages.

The need for a human-factorable specification technique for interaction languages in human-computer systems is well-recognized. This research is an attempt to codify some of the problems and to propose such a technique. Its merit and effectiveness will be known only after it has been thoroughly evaluated.

REFERENCES

[BLEST82] Bleser, T. and J. Foley. "Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces," Prcc. Conference on Human Factors in Computer Systems. Gaithersburg, Md. (March 1982).

[HARTH83] Hartson, R., R. Ehrich, and D. Johnson. "The Management of Dialogues for Human-Computer Interfaces," to be submitted for publication (1983).

[HEINL75] Heindel, L. and J. Roberto. "LANG-PAK: An Interactive Language Design System," Elsevier Computer Science Library: Programming Languages Series; 1. American Elsevier Publishing Co.,Inc., New York (1975).

[JACOR83] Jacob, R. "Using Formal Specifications in the Design of a Human-Computer Interface," Communications of the ACM. 26,4 (April 1983).

[JENSK74] Jensen, K. and N. Wirth. "Pascal User Manual and Report," Springer-Verlag, New York (1974).

[JOHND82] Johnson, D. and R. Hartson. "The Role and Tools of a Dialogue Author in Creating Human-Computer Interfaces," VPI&SU, Department of Computer Science Technical Report, (May 1982).

[JOHNS78] Johnson S. and T. Lesk. "UNIX Time-Sharing System: Language Development Tools," Bell System Technical Journal. 57,6 (July-August 1978).

[JOHNS80] Johnson, S. "Language Development Tools on the UNIX System," Computer. 13,8 (August 1980).

[LEDGH74] Ledgard, H. "Production Systems: or Can We Do Better than BNF?" Communications of the ACM. 17, 2 (February 1974).

[LEDGH80] Ledgard, H. "A Human Engineered Variant of BNF," Sigplan Notices. 15, 10 (October 1980).

[LINDT83a] Lindquist, T. "The Application of Software Metrics to the Human-Computer Interface," Proc. IEEE COMPCON Fall 1983 Conference. Washington, D.C. (September 1983).

[LINDT83b] Lindquist, T., R. Fainter, and M. Hakkinen. "GENIE: A Modifiable Computer-Based Task for Experiments in Human-Computer Interaction," submitted for publication (1983).

[MARCM76] Marcotty, M., H. Ledgard, and G. Bochmann. "Sampler of Formal Definitions," Computing Surveys. 8, 2 (June 1976).

[MORAT81] Moran, T. "The Interaction Language Grammar: a representation for the user interface of interactive computer systems," International Journal of Man-Machine Studies. 15 (1981).

[NARAP83] Narang, P., R. Ehrich, and D. Johnson. "Dynamic Languages for Human-Computer Interaction," to be submitted for publication (1983).

[NAURP63] Naur, P., editor. "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM. 6 (January 1963).

[REISP81] Reisner, P. "Formal Grammar and Human Factors Design of an Interactive Graphics System," IEEE Transactions on Software Engineering. SE-7, 2 (March 1981).

[REISP82] Reisner, P. "Further Developments Toward Using Formal Grammar as a Design Tool," Proc. Conference on Human Factors in Computer Systems. Gaithersburg, Md. (March 1982).

[SHNEB82] Shneiderman, B. "Multi-Party Grammars," IEEE Trans. on Systems, Man, and Cybernetics. (March 1982).

[SMITD83] Smith, D., et al. "Designing the STAR User Interface," BYTE Publications Inc. (April 1983).

[WEGNP72] Wegner, P. "The Vienna Definition Language," Computing Surveys. 4,1 (March 1972).

[WIRTN77] Wirth, N. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?" Communications of the ACM. 20, 11 (November 1977).

[YUNTT84] Yunten, T. and R. Hartson. "Supervisor-Based System Development Methodology," to appear in Advances in Human-Computer Interaction, Ablex Publishing Co. (1984).

November 1983

OFFICE OF NAVAL RESEARCH

Engineering Psychology Group

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CAPT Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Dr. Dennis Leedom
Office of the Deputy Under Secretary
of Defense (C^3I)
Pentagon
Washington, D.C. 20301

Department of the Navy

Engineering Psychology Group
Office of Naval Research
Code 442 EP
Arlington, VA 22217 (2 cys.)

Manpower, Personnel & Training
Programs
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Sciences Division
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Special Assistant for Marine Corps
Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

CDR James Offutt, Officer-in-Charge
ONR Detachment
1030 East Green Street
Pasadena, CA 91106

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. J.S. Lawson
Naval Electronic Systems Command
NELEX-06T
Washington, D.C. 20360

Dr. Neil McAlister
Office of Chief of Naval Operations
Command and Control
OP-094H
Washington, D.C. 20350

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

Combat Control Systems Department
Code 35
Naval Underwater Systems Center
Newport, RI 02840

Human Factors Department
Code N-71
Naval Training Equipment Center
Orlando, FL 32813

November 1983

Department of the Navy

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training & Equipment Center
Orlando, FL 32813

CDR Norman E. Lane
Code N-7A
Naval Training Equipment Center
Orlando, FL 32813

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Dr. L. Chmura
Naval Research Laboratory
Code 7592
Computer Sciences & Systems
Washington, D.C. 20375

Dr. Edgar M. Johnson
Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Human Factors Technology Administrator
Office of Naval Technology
Code MAT 0722
800 N. Quincy Street
Arlington, VA 22217

Department of the Navy

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 334A
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design
NAVAIR 5313
Washington, D.C. 20361

Mr. Philip Andrews
Naval Sea Systems Command
NAVSEA 61R2
Washington, D.C. 20362

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 81323
Washington, D.C. 20360

Larry Olmstead
Naval Surface Weapons Center
NSWC/DL
Code N-32
Dahlgren, VA 22448

Mr. John Impagliazzo
Code 101
Naval Underwater Systems Center
Newport, RI 02840

Navy Personnel Research and Development Center
Planning & Appraisal Division
San Diego, CA 92152

Dr. Robert Blanchard
Navy Personnel Research and Development Center
Command and Support Systems
San Diego, CA 92152

CDR J. Funaro
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

November 1983

Department of the Navy

Mr. Jeffrey Grossman
Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Human Factors Engineering Branch
Code 4023
Pacific Missile Test Center
Point Mugu, CA 93042

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

CDR C. Hutchins
Code 55
Naval Postgraduate School
Monterey, CA 93940

Department of the Army

Director, Organizations and
Systems Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D.C. 20310

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

AFHRL/LRS TDC
Attn: Susan Ewing
Wright-Patterson AFB, OH 45433

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Department of the Air Force

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks Air Force Base, TX 78235

Foreign Addresses

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
P.O. Box 2000
Downsview, Ontario M3M 3B9
Canada

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314

Dr. Clinton Kelly
Defense Advanced Research Projects
Agency
1400 Wilson Blvd.
Arlington, VA 22209

Other Organizations

Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

Dr. J.O. Chinnis, Jr.
Decision Science Consortium, Inc.
7700 Leesburg Pike
Suite 421
Falls Church, VA 22043

Dr. Paul E. Lehner
PAR Technology Corp.
P.O. Box 2005
Reston, VA 22090

Dr. Robert T. Hennessy
NAS - National Research Council (COHF)
2101 Constitution Avenue, N.W.
Washington, D.C. 20418

November 1983

Other Organizations

Dr. Amos Freedy
Perceptronics, Inc.
6271 Variel Avenue
Woodland Hills, CA 91364

Dr. Deborah Boehm-Davis
General Electric Company
Information & Data Systems
1755 Jefferson Davis Highway
Arlington, VA 22202

Mr. Edward M. Connelly
Performance Measurement
Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. Marvin Cohen
Decision Science Consortium, Inc.
Suite 721
7700 Leesburg Pike
Falls Church, VA 22043

Dr. Richard Pew
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

Dr. Alan Morse
Intelligent Software Systems, Inc.
Amherst Fields Research Park
529 Belchertown Rd.
Amherst, MA 01002