THE STRUCTURE AND DEVELOPMENT OF HUMAN-COMPUTER INTERFACES

by

Deborah Hix Johnson

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in
Computer Science

APPROVED:

H. Rex Hartson

Roger W. Ehrich  Timothy E. Lyndquist

Robert C. Williges  Vinod Chachra

January, 1985
Blacksburg, Virginia
THE STRUCTURE AND DEVELOPMENT OF HUMAN-COMPUTER INTERFACES

by

Deborah Hix Johnson

Committee Chair: H. Rex Hartson

Department of Computer Science

(ABSTRACT)

The Dialogue Management System (DMS), the setting for this research, is a system for designing, implementing, testing, and modifying interactive human-computer systems. As in the early stages of software engineering development, current approaches to human-computer interface design are ad hoc, unstructured, and incomplete. The primary goal of this research has been to develop a structural, descriptive, language-oriented model of human-computer interaction, based on a theory of human-computer interaction. This model is a design and implementation model, serving as the framework for a dialogue engineering methodology for human-computer interface design and interactive tools for human-computer interface implementation.

This research has five general task areas, each building on the previous task. The theory of human-computer interaction is a characterization of the inherent properties of human-
computer interaction. Based on observations of humans communicating with computers using a variety of interface types, it addresses the fundamental question of what happens when humans interact with computers. Formalization of the theory has led to a multi-dimensional dialogue transaction model, which encompasses the set of dialogue components and relationships among them. The model is based on three traditional levels of language: semantic, syntactic, and lexical. Its dimensions allow tailoring of an interface to specific states of the dialogue, based on the sequence of events that might occur during human-computer interaction.

This model has two major manifestations: a dialogue engineering methodology and a set of interactive dialogue implementation tools. The dialogue engineering methodology consists of a set of procedures and a specification notation for the design of human-computer interfaces. The interactive dialogue implementation tools of AIDE provide automated support for implementing human-computer interfaces. The AIDE interface is based on a "what you see is what you get" concept, allowing the dialogue author to implement interfaces without writing programs.

Finally, an evaluation of the work has been conducted to determine its efficacy and usefulness in developing human-computer interfaces. A group of subject dialogue authors
using AIDE created and modified a prespecified interface in a mean time of just over one hour, while a group of subject application programmers averaged nearly four hours to program the identical interface. Theories, models, methodologies, and tools such as those addressed by this research promise to contribute greatly to the ease of production and evaluation of human-computer interfaces.
DEDICATION

This work is dedicated to my Father,

to whom education was always so important.
I think he would have been proud.
ACKNOWLEDGEMENTS

The completion of a Ph.D. dissertation is a monumental goal in anyone's life. There are many people without whom I would not have been able to reach this goal. My committee chair, Dr. H. Rex Hartson, has been my mentor, sometimes my tormentor, and always my friend. His support and encouragement have been one of the few constant factors in the hectic life of this Ph.D. student. Dr. Roger W. Ehrich has given invaluable assistance in my research efforts, especially in its language aspects, and has instilled in me a marvelous appreciation for Italian cuisine. Dr. Timothy E. Lindquist always showed confidence in me and urged me to continue, even when others did not. Dr. Robert C. Williges and Dr. Vinod Chachra took time from their busy schedules to participate on my committee; their comments and time are greatly appreciated. The members of the DMS research group, especially , , and , have also provided much technical advice, implementation effort, and fun during the course of this research. Dr. John Whiteside and at Digital Equipment Corporation made possible an internship which gave me valuable insight into the "real world" aspects of human-computer interfaces. Funding has been provided by
the Office of Naval Research, Dr. John J. O'Hare, Technical Director, and the National Science Foundation, Dr. H. E. Bamford, Jr., Director.

On a personal level, my friends and family have helped make all this tolerable. saw me in my best moments and my most insane moments, and loved and supported me in spite of them. The times we've shared are among my fondest memories. were willing listeners and understanding supporters. Thanks go to for serving as my technical editor through several rounds of this enormous document, as well as being my friend through several rounds of internal conflicts. My mother, , and brother, have given me unfailing love and caring and held down the fort on the farm while I struggled through all this. I only hope I can start repaying them now. And finally, thanks to , who for so many years was "sailing right behind" me; those times will not be forgotten.

The lines from a song that was popular when I was finishing high school seem appropriate for this time as I am finishing my dissertation:

"...Sail on, Silver Girl, sail on by,
Your time has come to shine,
All your dreams are on their way..."

Paul Simon, Bridge Over Troubled Waters
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION AND GOAL OF THIS RESEARCH</td>
<td>1</td>
</tr>
<tr>
<td>Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>Purpose of This Research</td>
<td>4</td>
</tr>
<tr>
<td>Scope of This Research</td>
<td>7</td>
</tr>
<tr>
<td>II. THE DIALOGUE MANAGEMENT SYSTEM</td>
<td>9</td>
</tr>
<tr>
<td>Overview and Background of DMS</td>
<td>9</td>
</tr>
<tr>
<td>New Concepts and Roles in DMS</td>
<td>10</td>
</tr>
<tr>
<td>Dialogue Independence</td>
<td>10</td>
</tr>
<tr>
<td>Dialogue Author</td>
<td>12</td>
</tr>
<tr>
<td>DMS as the Context for This Research</td>
<td>13</td>
</tr>
<tr>
<td>III. LITERATURE REVIEW</td>
<td>17</td>
</tr>
<tr>
<td>Approaches to Modeling Human-Computer Interaction</td>
<td>19</td>
</tr>
<tr>
<td>Existing Methods of Language Specification</td>
<td>25</td>
</tr>
<tr>
<td>Static versus Dynamic Languages</td>
<td>26</td>
</tr>
<tr>
<td>BNF Specifications</td>
<td>28</td>
</tr>
<tr>
<td>State Transition Diagram Specifications</td>
<td>33</td>
</tr>
<tr>
<td>Other Specification Techniques</td>
<td>37</td>
</tr>
<tr>
<td>Other Research in Language Specification</td>
<td>42</td>
</tr>
<tr>
<td>Language Implementation and Recognition Tools</td>
<td>43</td>
</tr>
<tr>
<td>Interactive Tools for Dialogue Design and Implementation</td>
<td>44</td>
</tr>
<tr>
<td>IV. A THEORY OF HUMAN-COMPUTER INTERACTION</td>
<td>48</td>
</tr>
<tr>
<td>The Role of Theory in Research</td>
<td>48</td>
</tr>
<tr>
<td>A Scenario of Human-Computer Interaction</td>
<td>49</td>
</tr>
<tr>
<td>A Theory of Human-Computer Interaction</td>
<td>52</td>
</tr>
<tr>
<td>Future Research on Theory of Human-Computer Interaction</td>
<td>54</td>
</tr>
<tr>
<td>V. A MULTI-DIMENSIONAL DIALOGUE TRANSACTION MODEL OF HUMAN-COMPUTER INTERACTION</td>
<td>55</td>
</tr>
</tbody>
</table>
Chapter I

INTRODUCTION AND GOAL OF THIS RESEARCH

"'Where shall I begin, please your Majesty?' he asked. 'Begin at the beginning,' the King said, very gravely, 'and go on till you come to the end: then stop.'" Lewis Carroll, *Alice's Adventures in Wonderland*

"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."

This advertisement from 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 productivity 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.

1.1 PROBLEM STATEMENT

During the last decade or so, the fields of human-computer interaction and human-computer dialogue design have become recognized as not only viable, but necessary, research areas. However, initial research efforts have emerged without a unified framework within which to design, implement, evaluate, and modify human-computer interfaces. Directives from workshops on human-computer interaction mandate the need for "a model of interaction and a language for specifying user interactions...which have been subjected to experience in real world applications" [GIITW83]. Such models must, of course, be "sufficiently simple to be accepted" as workable paradigms [MOIW80]. Much like the early stages of software engineering development, current approaches to hu-
man-computer dialogue design are ad hoc, unstructured, and incomplete. And like more recent advances in software engineering, the jumble of work on human-computer interaction needs models and methodologies to structure it to the point where it can justifiably be called dialogue engineering.

As long as there have been computers, there have been human-computer interfaces. And as long as there have been computers, there have been poor human-computer interfaces. One goal of research in the area of human-computer interaction is to produce quality interfaces. This goal involves a two-step process. First, the phenomena and elements of human-computer interaction must be observed and their relationships understood. Then, and only then, can the means for producing quality interfaces be provided. Without a framework, the elements of human-computer interfaces have no cohesion, but are simply a group of random, unconnected messages, displays, and user actions. Consequently, dialogue development procedures are unstructured and random as well.

Subdividing the interface into its components "is extremely helpful in user-interface design because it enables us both to categorize the problems arising in design and to be more thorough in addressing them" [NEWMW79]. Thus, observation allows formulation of a theory of human-computer interaction upon which the structuring and modeling of in-
terfaces can be based. This, in turn, allows interface design, through the use of a dialogue engineering methodology, and construction, through the use of interactive tools. Finally, an evaluation is needed to determine whether the observations and structuring produced viable design and construction mechanisms.

1.2 PURPOSE OF THIS RESEARCH

The flurry of research in this relatively new field of human-computer interaction has produced numerous models for many facets of human-computer systems. There are models for the complete human-computer system, for control flow within the system, for dialogue simulation and/or rapid prototyping of the system, and for the architecture of the system. Indeed, there seem to be models for everything except the human-computer interface. Thus, a primary goal of this research has been to develop and evaluate a structural, descriptive, language-oriented model of human-computer interaction, based on a theory of human-computer interaction. This model is a design and implementation model, serving as the framework for a dialogue engineering methodology for the design of human-computer interfaces and interactive tools for the implementation of human-computer interfaces.

This research is broken down into five general task areas, each building on the previous task. The theory of hu-
man-computer interaction is a characterization of the inherent properties of human-computer interaction. It was formulated by observing and analyzing what happens when people interact with computers in a large variety of situations. The multidimensional dialogue transaction model is the heart of this research effort. It is a formalization of the theory, to explain and structure what happens when people interact with computers. It presents a formal representation of the elements of human-computer interaction and the relationships among these elements. The dialogue engineering methodology provides the procedures and a specification notation for the design of human-computer interfaces, based on the model. The dialogue implementation tools of the Author's Interactive Dialogue Environment (AIDE) provide automated support for the methodology, facilitating interface construction. Finally, an evaluation of the research has been carried out to determine its efficacy and usefulness in developing human-computer interfaces.

The progression among these research tasks was such that the theory of human-computer interaction identified the elements and the structure of dialogue, formally represented by the model. The model, in turn, dictated the requirements for both the methodology and the AIDE tools for dialogue development. Each component of dialogue in the model required
a representational notation in the methodology, as well as a corresponding AIDE tool for its implementation. Evaluation was the final step in the progression.

Because the five tasks followed this logical progression, they could be viewed as analogous to life cycle stages in system development. The theory is similar to requirements specification, the model can be thought of as the design phase, the methodology and tools represent the implementation phase, and evaluation is analogous to testing and modification. Because this progression was not strictly sequential, but rather was more like the iteration that occurs during the life cycle, there were numerous feedback cycles.

The overall research effort has a strong language orientation. This language orientation assumes that the phenomena and problems of human-computer interactions are basically phenomena and problems in language design. Thus, in DMS, the problem of human-computer interface design is viewed as a problem of interaction language design. A wide variety of devices, using multiple sensory channels, is the medium through which communication in an interaction language is made between the human and the computer. As in human-to-human discourse, that communication can be highly dynamic, with instantaneous responses by both parties. In fact, this dynamism and its manifestations are the main differentiating
features between human-computer interaction languages and traditional programming languages. The designer of an interaction language must specify both the language of the human and of the computer, in addition to their dynamic interrelationships. This research has produced an approach to interface design that addresses the semantic, syntactic, and lexical levels of an interaction language. The language orientation is the fabric that ties all five research tasks together at all levels. Because all tasks were addressed with the view that the problem is one of language research, all have a common concept, terminology, and approach.

1.3 SCOPE OF THIS RESEARCH

The scope of this research is the modeling of human-computer interaction and specification of human-computer interfaces, not design or execution of interfaces. All five task areas (theory, model, methodology, tools, and evaluation) were approached from a computer science viewpoint, rather than a human factors viewpoint. The research effort did not address directly such interface design issues as principles and guidelines. However, one goal was to support the human factors activities of evaluation and application of design principles at each step of the interface design process, without presuming to base the approach on any specific principles.
In addition, the research did not address directly the transaction executor which executes the interface at application system run-time. However, the results of this research generated both requirements and solutions for many of the dialogue executor needs, thereby advancing the design of that tool, as well.
Chapter II
THE DIALOGUE MANAGEMENT SYSTEM

"We shall never understand one another until we reduce the language to seven words." Kahlil Gibran, Sand and Foam

2.1 OVERVIEW AND BACKGROUND OF DMS

The Dialogue Management System "is a comprehensive system for designing, implementing, testing, and maintaining large software systems with human-computer interfaces" [HARTH84]. The DMS project inspired the research reported herein and is the implementational setting for the interactive tools of AIDE. The theory, the modeling, and the methodology, however, are independent of a specific supporting environment or system such as DMS. Initially funded by the Office of Naval Research, DMS began as one task of a three year joint project between the Departments of Computer Science and Industrial Engineering at Virginia Tech. This program was devoted to the research and development of effective human-computer interfaces. The National Science Foundation is currently funding a three year extension of this work in the Department of Computer Science. DMS is an extensive research effort currently involving two faculty members, several graduate research assistants, and one programmer.
2.2 NEW CONCEPTS AND ROLES IN DMS

2.2.1 Dialogue Independence

The DMS philosophy emphasizes 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 may be 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, as shown in 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 form and content of the dialogue.
Figure 1: Internal and External Dialogue
component. *Internal dialogue*, on the other hand, has no direct
collection to the user of the system, but serves as a link
between the dialogue component and the computational compo-
nent 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 com-
putational components of software systems, individuals in
separate roles are responsible for each of the two compo-
nents. An application programmer writes computational com-
ponents, but writes no dialogue. In DMS, an individual in a
new role, that of a *dialogue author*, has sole responsibility
for developing the dialogue which comprises the human-compu-
ter 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 inter-
face development. The dialogue author creates the dialogue
so that its form and content reflect principles of good hu-
man-computer interaction.

The major reason for this separation of roles between the
dialogue author and an application programmer is that an ap-
lication programmer is generally not skilled in writing ef-
fective 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 logically complex piece of software, to write code to specify user interaction dealing with such things as input data checking or message format consistency. The main objective of a dialogue author is to develop 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. The hypothesis is that this separation produces more effective human-computer interfaces, and it allows these interfaces to be quickly and easily modified as end-user needs change.

2.3 DMS AS THE CONTEXT FOR THIS RESEARCH

In DMS, a human-computer system is viewed as consisting of three major components, as shown in Figure 2: a dialogue component (what the computer system presents to the human and what the human inputs to the computer), the computational component (the processing mechanisms of the system, with which the human does not directly interact), and the global
Figure 2: Structure of a Human-Computer System under DMS
control component (which governs the logical sequencing among the dialogue events and computational events). A holistic system development methodology, the SUPERvisory Methodology And Notation (SUPERMAN) [YUNTT85] is used for requirements specification, system design, implementation, and modification. SUPERMAN, discussed in Chapter 7.3, emphasizes dialogue independence, representing the entire interactive system in supervised flow diagrams (SFDs). Symbols of the SFDs are elements in a graphical programming language (GPL), which produces the global control component. The computational component is produced by an application programmer using conventional programming environments.

The external dialogue component is the human-computer interface for an interactive system and is a collection of dialogue transactions. These dialogue transactions are developed by a dialogue author without writing programs, using a set of interactive tools called the Author's Interactive Dialogue Environment (AIDE). This dialogue component is a separate, clearly delineated part of the entire human-computer system. This research has produced a theory, a model, a methodology, and tools which apply to the design of this dialogue component as an integrated part of the total system. This research has not addressed either the global control or computational components, except as they relate to the dialogue component.
At execution-time for the systems which DMS produces, a DYnamic Language EXecutor (DYLEX) [NARAP85] is the main element of the transaction executor, processing the user inputs as the application system executes. A multi-process execution environment supports DMS tools (AIDE and GPL) at design-time, as well as the DMS-produced application systems at execution-time. DMS is currently implemented on a VAX 11/780 under the VMS operating system.
Chapter III
LITERATURE REVIEW

"The sole substitute for an experience which we have not ourselves lived through is art and literature." Alexander Solzhenitsyn, Nobel Lecture

The need for models and descriptive techniques has long been recognized in well-established areas of Computer Science. Such techniques as methodologies for system design and formal language specification of programming languages have received much research attention. Numerous specification techniques have been developed, including Backus-Naur Form (BNF), regular expressions, context-free grammars, state transition diagrams, augmented transition networks, Petri nets, and finite state automata.

Since this research views a human-computer interface as an interaction language, modeling and specifying the interface can be viewed as a problem of language specification. However, researchers have realized for some time that severe limitations exist in the models and notations typically used to formally specify a language. Formal language definitions meet with resistance, especially from more pragmatic users, because they are often so cryptic, information-packed, and difficult to understand. The reader of a language defini-
tion, whether a systems programmer or a system user, should be able to quickly grasp the formalism and, from it, to extract the information needed. Moreover, there are many well-documented deficiencies in existing specification schemes.

Several workshops have addressed issues of interface modeling and specification in detail. One of the first of these was the Seillac II Conference on the Methodology of Interaction [MOIW80]. It raised a wide spectrum of topics and questions for discussion and future research. The definition of interaction and of an interactive system was pursued. The formal description and the construction of user interfaces, as well as models of interaction, were addressed. Steps toward the development of a methodology for user interface design were outlined. Methods for user interface evaluation were explored, as well as future trends in interactive systems. This workshop raised thought-provoking issues that set the stage for research in human-computer interaction in the '80s.

The Graphical Input Interaction Techniques Workshop was held in Seattle in 1982. A summary of its discussions [GIITW83] revolved around techniques for the design, implementation, and evaluation of the human-computer interface for a graphics application system. It proposed the develop-
ment of a software tool called a User Interface Management System (UIMS). Like Seillac II, the scope of this workshop was far-reaching, addressing such issues as separation of dialogue from computation (dialogue independence) in an application system; the need for interdisciplinary work among computer scientists, cognitive psychologists, graphics designers, and representative users; and the identification of several specialist roles in producing an interactive system (e.g., application programmer, interaction designer (dialogue author), and a UIMS manager). Models, tools, and techniques were all included in the discussions. While this workshop addressed a broad range of topics, its approach to solutions was heavily influenced by graphical interaction concerns, making it somewhat less than general in its approach.

3.1 APPROACHES TO MODELING HUMAN-COMPUTER INTERACTION

Wasserman identified the Input-Process-Output (I-P-O) or "turn-taking" paradigm of control which is common to interactive information systems [WASSA80]. Many dialogue models are based at least in part on this fundamental sequencing. This sequencing actually describes a behavioral model of the entire system (because of the "process" component), not just the human-computer dialogue (the "input" and "output" components).
Few explicit structural models of human-computer interaction seem to exist; the ones that do are often overly simplistic and incomplete. One working group of the Seillac II Conference proposed a high-level model of interaction based on a processing paradigm. A control level and a performance level between human and computer represent, respectively, "what" interaction occurs and "how" it occurs. While basically sound, the model, because of its high level, provides little insight into the specific task of designing interfaces.

A model for expressing functional requirements of human-computer interaction is presented in [CASEB82]. This model describes the interface specification in terms of a finite state machine (or state transition diagram), but does not relate it to a language-oriented interface model. This finite state machine model divides an interface into classes of stimuli and responses. The addition of checkpoints to validate input and timers for performance measurements extends the model. Application-specific vocabulary and semantics are used to specify system requirements in a Real-Time Requirements Language (RTRL). A comparison of RTRL to an informal English prose version of the requirements specification for a system showed RTRL to produce more complete, consistent specifications [CASEB82].
General Transition Networks (GTNs) also use state transition diagrams as the basis for modeling an interface, again without specifically relating them to language [KIERD83]. Nodes of GTNs represent states and arcs are labeled with both conditions and actions, with examination of the conditions being done in a specified order to trigger transitions. The GTN's key feature, according to [KIERD83], is its ability to describe hierarchies of modes or states of the system. The work done with GTNs appears only to be theoretical. While the authors claim that GTNs are powerful enough to easily describe very complex systems, the example given is a "simplified form of portions" of a specific word processor. Also, no mention is made of their use in a real implementation and simulation.

An interface processor is the basis for another human-computer interaction model [EDMOE82]. This interface processor consists of input, output, and dynamics processes which perform simple transformations (e.g., from keyboard input to character strings) and determine the actions of the computer. This model appears to dwell on physical processes at the expense of providing insight into the essential nature of human-computer communication. The author postulates that, using this model of an interface, "we could clearly arrive at...a description of the system" and "arrange
that the user's model matched construction of the interface." However, it is not made clear how this might be accomplished. The author observes the need for a specification scheme that also serves as an implementation language.

Four distinct stages of human activity during interaction with a computer have been proposed: intention, selection, execution, and evaluation [NORMD84]. Because each stage has different implications for system design, different supporting tools are needed. The premise here is that the four stages can be used to guide screen design (e.g., evaluation is essentially a feedback stage, so appropriate information should be given to the end-user). While the stages appear realistic, this model is much more philosophical and less pragmatic than some of the others, without specific means for direct application.

Borufka, Kuhlmann, and Pfaff at the Technische University of Darmstadt and ten Hagen at the Amsterdam Mathematical Center have created the concept of a "dialogue cell" as a model around which human-computer interactions can be described and developed [BORUHS1, BORUHS2, PFAFG82]. A dialogue author develops dialogue, but in a dialogue language, programming it much the same way an application programmer would do. This approach, which does not seem to be language-based, places a strong emphasis on graphics; the prem-
ise is that data is more easily represented pictorially than numerically. Tools which are provided for the dialogue author include an input/output system for accessing graphics devices (through the Graphical Kernel System) and a "dialogue language" for specifying data structures and control flow. Dialogue cells consist of four basic elements which comprise the dialogue specification. The "prompt" is a description of the type of data to be entered by the human; the "symbol" is the input of the user; the "echo" is the interpretation of the entered symbol; and the "value" is derived from the symbol for further use. Dialogue cells are combined to represent sequences of interactions.

Examples of dialogue cells imply that they contain not only dialogue information, but also computational code to carry out the semantics of that dialogue, which violates the concept of dialogue independence. Also, despite several pages of these lengthy example dialogues, their source is never explicitly stated; the assumption is that the dialogue author programmed them. Viewport definitions, data types, and control constructs are all contained in these dialogue cell examples, implying that the dialogue author must, despite the high-level dialogue language, be a fairly sophisticated programmer! The modeling of interfaces as dialogue cells, and structuring of cells into prompt, symbol, echo,
and value is a worthwhile approach to understanding the structure of human-computer dialogue. Unfortunately, the examples of dialogue cells do not revolve around this structure, and very little indication is given as to the basis for their construction.

An "interaction event" has been proposed as the basis of a dialogue model [BENBI84], presented as a structured approach to the definition of human-computer dialogues. The basic premise is that an interaction event is the main component of a dialogue and that a dialogue is composed of a sequence of these events. An interaction event is composed of a system "prompt", a user "input", a system processing "action" based on the input, and "flow control" which determines the next interaction event. "Escape", "help", input "check", and unique event "ID" are also included in the model. A dialogue generator is proposed as a software tool for implementing human-computer dialogues. A fully functional dialogue generator has been implemented, the execution of which is based on a tabular form of interaction events, called "reference sets".

While this model seems more thorough than most, the concept of the dialogue generator is not completely clear. Examples of text, check, action, and flow control tables, which the dialogue generator processes, are presented. How-
ever, it is not made clear how these tables are produced. Also, the definition of the user input seems, at least from the examples, to be limited to data-typing, without provisions for more complicated aspects of input definition. The definition of an interaction event encompasses more than just the dialogue. The action and flow control components are really part of the computational and global control components of the system, making this a model of the complete human-computer system, not just its interface. Despite these ambiguities, the basic benefits of this research are sound: practically, to produce a dialogue generator to facilitate human-computer dialogue implementation; and theoretically, to provide a better understanding of these dialogues through a common set of concepts.

3.2 EXISTING METHODS OF LANGUAGE SPECIFICATION

The approach to modeling human-computer interaction which has received the most research attention is in specifying an interface as a formal language. This idea serves as the basis for several systems which actually use formal specification techniques for defining and implementing an application system's interface. The idea of viewing a user interface from a language viewpoint, at the conceptual, semantic, syntactic, and lexical levels, was pioneered by Foley [FOLEJ74,
FOLEJ80, FOLEJ82]. Foley's "conceptual level" is the collection of basic system concepts the user must understand. The "semantic level" encompasses input operations and output presentation techniques. The "syntactic level" contains specific token sequences to invoke semantic actions, as well as specific form and content of output. The "lexical level" defines the token structure in terms of hardware. Many others have used this approach as the basis for interface modeling. Semantic, syntactic, and lexical levels provide the framework for the model of human-computer interaction that is the core of this dissertation research, but with somewhat different definitions for each level. The numerous ways in which language-oriented specification techniques have been used to model and define human-computer interaction will be described in the following sections.

3.2.1 Static versus Dynamic Languages

Most well-known methods of language syntax specification are used primarily for representation of static programming languages and are inadequate for representation of dynamic interaction languages [JACOR83a]. In a static language, the entire sequence of both system and user actions is input, typically as a source code program. The conditions for all possible alternatives by both system and user must be expli-
cit in the code. Processing mechanisms are bound at language design-time. 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, location, possibly no echo at all) are just a few characteristics which are unique to interaction languages. Such questions as the scope of an interaction language and the difficulty of separating syntax and semantics make definition of interaction languages particularly difficult [LEDGH81]. Many of these are not provided for in traditional language specification schemes.

Two classes of interface specification techniques are most prevalent: those based on BNF-type definitions and those based on state transition diagrams. Jacob appears to have done the most extensive survey and thoughtful comparison of the advantages and disadvantages of both BNF and state transition diagrams, along with examples of the use of
each of these for language specification [JACOR83a]. He concludes that state transition approaches provide more comprehensible language specifications, because they show surface structure of the human-computer interface better than BNF does.

3.2.2 BNF Specifications

One of the best known systems for representing the syntax of a language is the Backus-Naur Form (BNF) [NAURP63]. While it is useful as a specification system, BNF has several well-known deficiencies, one of which is its inability to represent context-sensitive requirements. More importantly, BNF is often difficult for humans to understand. 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. In particular, this makes it very difficult to determine legal expressions in a language from looking at its BNF specification.
An improvement 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. The concept of a syntactic environment is introduced to insure that declared identifiers are compatible with their uses. While adding these features to the syntax representation conveys more 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. In dialogue management, 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 [WIRTN77, LEDGH80]. An extension of the notation for regular expressions has been proposed to give flexibility to syntax definition [WILLM82]. 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 understandability, readability, and usability.

Despite these well-recognized drawbacks, however, BNF has been used extensively in specification of human-computer interfaces. One formal approach is intended specifically to describe the human factors of an interface [FOLEJ81, BLEST82]. It defines the lexical and syntactic aspects of both the input and output of an interface. The input definition defines tokens and their relationships, while the output definition defines screen characteristics and content. While this is a thorough approach to defining input and output in a system, if automated tools were available to develop the output definition (i.e., the screen form and content) directly, its formal specification might not be necessary. Once the interface is formally specified, evaluative metrics are applied to the specification in order to determine potential design flaws.

Extended LL(1) grammars with added graphical information have been used as interface specifications [OLSED83, DEMPE83] in a system for automatically generating interactive systems. SYNGRAPH (SYNtax directed GRAPHICs) generates
the user interface for interactive graphics applications. An ELL(1) grammar describing the interaction language is used as input to the generator, as well as Pascal code which is invoked to perform the related semantic actions. Output is a recursive descent parser for the interaction language, as well as a scanner and a screen manager. SYNGRAPH has produced the concept of an "interactive pushdown automaton" as the basis for describing the interface syntactic components [OLSED84a]. While this system precludes coding of the interface, an ELL(1) grammar which describes it must still be produced by the system designer.

Another system that is an outgrowth of the SYNGRAPH research is a Menu Interaction Kontrol Environment (MIKE) [OLSED84b]. MIKE can be used to generate text-based interactions; interaction languages are represented as Pascal procedures and functions. Defining the interface with expressions differs significantly from the SYNGRAPH grammatical approach, and has proven to be much easier to learn. A menu-driven interface is generated with automatic support for help, command cancelling, rubout, and prompting.

Simulation systems have been developed which accept as input BNF production rules with associated actions, and produce a prototype of the human-computer interface [HANAP80]. Association of actions in BNF-like expressions is also found
in TBNF, a specification tool used to define an operator console configuration [LAWSH78]. TBNF was used to specify a Power Network Telecontrol System in Barcelona, Spain. The specification was small (only sixteen productions) but unfortunately they were given in Spanish in the article!

Set-grammars have been proposed as yet another BNF-like scheme to represent the user's "grammar in the head" [PAYNS83]. Syntax is represented as a production rule, a set definition, and a selection rule. Task-action grammars (TAG) are an approach to representing the user's mental image of a task (interaction) language [PAYNS84]. This meta-language for defining interaction languages has two levels of description: concepts, which specify grammatical objects, and rule-schemata, which specify mappings from task descriptions to action specifications.

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 non-terminals with a party (i.e., either human or computer) identifier, assignment of values to non-terminals when appropriate, and definition of a non-terminal which will match any input string if no other parse of that input is successful. The grammar 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.

3.2.3  *State Transition Diagram Specifications*

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.

One of the earliest uses of state transition diagrams was in the specification of a compiler for a programming language [CONWM63]. Actions associated with each state transition indicate what is to happen when that transition occurs. Similar augmented transition network grammars have been used to analyze natural language structure [WOODW70].

The top level design of an interactive computer system using state transition diagrams [PARND69] evolved in response to the increased need for consideration of human interaction in the system design process. Specifically, ter-
minal state transition diagrams specify appropriate messages at each state of an interactive system. Current research has now progressed far beyond this point, but the concepts were novel when first proposed.

The Abstract Interaction Handler (AIH) of Foley, Feldman, et al. [FOLEJ74, FELDM81, FOLEJ81, FELDM82, KAMRA83] is so named because the user interface is seen by the computational component of the application system as an abstraction rather than as style-dependent and device-dependent details. The key concept of their approach is its language orientation. The system, its language design tools, and development methodology are all based on Foley's three levels of interaction languages: lexical, syntactic, and semantic. Interactive tools are used for designing user interfaces at each of these three language levels, as well as the top-most conceptual level. Interaction language specification is accomplished with a variation of augmented transition networks (ATNs) that includes strong typing for data abstraction. The syntactic definitions of the ATNs are stored in an interaction language program. The structure of the ATNs is a formal way to represent the syntax of interactive dialogues for both input and output. At execution-time the ATNs are interpreted by the Interaction Language Interpreter (ILI), so that user inputs are processed according to the grammar.
Semantic action modules are called by an interpreter at appropriate times for computational functions, and the interpreter passes user input values to the semantic modules. Language definitions of nodes are implemented as code in a high-level programming language, rather than as declarative definitions.

Many others have used state transition diagrams to represent the human-computer interface [DENEE77, DWYEB81, GREEM81]. However, these have all been used illustratively; none has applied the diagrams to any sizable real world application.

An evaluation of tools for designing human-computer interfaces has shown interesting results [GUESS82]. One tool, a powerful syntax-directed translator (SYNICS) was compared to a dialogue description language. SYNICS had an input structure based on BNF-like production rules, while the dialogue description language was based on a system similar to state transition diagrams. The results pointed to the diagrams of the dialogue description language as a much easier method of defining dialogue than production rules.

Jacob's use of state transition diagrams and a textual description language is also based heavily on Foley's three language levels. State transitions are associated with an input or an output token, but not both [JACOR83a, JACOR85].
That is, output is treated as a separate token, rather than as a special action, allowing specification of output dynamics. The output tokens include prompts, acknowledgements, and echos. Through a process of step-wise refinement, states are added to the state transition diagrams, making the specification of the interface more precise. The resultant specifications are detailed and voluminous. However, all specification levels use the same type of state transition diagram notation, employing sub-diagrams to control complexity. The state transition diagrams are converted to a corresponding text form (a state transition textual language) much like a high-level programming language. This textual form is executable, providing rapid prototyping capability for the interface. Application of these techniques to a real world Military Message System provides an interesting example.

Wasserman's User Software Engineering (USE) methodology [WASSA79, WASSA84a, WASSA84b] is a system for specifying not only the user interface but an entire interactive information system. Augmented transition diagrams are used to describe the language of the user. The machine representation of the state transition diagrams is a textual encoding in a "node and arc" language. As in most of the other systems and methods of interface representation (e.g., AIH, Jacob,
multi-party grammars), semantic control is intermixed with lexical and syntactic control at the same level of abstraction. This causes complexity problems in the state transition diagrams, shared with Jacob's approach, when syntactic functions such as token-level error handling are included in the diagrams. Overall, however, USE is a holistic (including both software engineering methodology and dialogue design) approach to system design, with emphasis on the interface.

3.2.4 Other Specification Techniques

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 author's knowledge) has been applied to the formal description of interactive systems. Specification techni-
ques which are complicated and not readily comprehensible contradict the basic tenet of the current emphasis on the design of quality 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.

The lexical level of Foley's conceptual, semantic, syntactic, and lexical levels has been further decomposed into two levels: lexical and pragmatic [BUXTW83]. The criticism is that Foley's lexical level encompasses too broad a definition, including such diverse features as composition of tokens, spatial display concerns, devices, and physical gestures. The new "lexical level" addresses only token compo-
sition information, while the "pragmatic level" subsumes issues of layout, devices, and gestures. Buxton states that this pragmatic layer is the main level of interaction between a human and a computer system. It therefore has the greatest influence on the user's perception of the system and should be given special attention.

A proposal for the formal specification of human-computer interfaces is written in first order logic using the rule-based language Prolog [ROACJ83]. This method of modeling, designing, and developing dialogues allows a uniform syntactic and semantic description of the interface. Because Prolog translators are available, this specification is also executable, and allows for rapid specification, implementation, and modification of an interface. An application example involved specification of a carrier-based air traffic control system, which took about 100 Prolog rules versus more than 5000 lines of Pascal code. While the idea of using Prolog as both a specification and an implementation tool is promising, the complexity of learning to create Prolog programs needs to be investigated.

The COUSIN-SPICE system of Hayes, Ball, and Reddy at Carnegie-Mellon University has a strong artificial intelligence flavor and deals with natural language understanding. Their concept of a quality interface is one that supports graceful
interaction in the way that human-to-human communication is graceful and robust [HAYEP81]. The basic concept is an attempt to provide interfaces to numerous different application systems, rather than providing separate interfaces to individual applications [HAYEP83]. Recent work on COUSIN-SPICE has evolved an interface definition centered around form-based interface abstractions, expressed in a language which is interpreted [HAYEP85]. Such an interface definition consists of a declaration of the form name followed by a sequence of field definitions containing attributes, as a textual encoding of the interface. External interface definitions expressed in form-based abstractions are used by COUSIN-SPICE to provide a wide variety of applications with consistent, quality interfaces. A significant contribution of the COUSIN-SPICE work is that their model of dialogue control departs from the standard, linear I-P-O model and involves the concurrency of possibly many communication media (e.g., simultaneous pointing, speaking, and typing).

The IBM syntax notation for specification of a command language is fairly well-known. It uses upper case letters (for keywords), lower case letters (for syntactic variables), brackets (to indicate optional symbols), and braces (to indicate choice from a list). Also, Pascal syntax spe-
Cification 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.

An empirical comparison of the IBM notation to the Pascal notation was conducted to determine which was easier to understand [EVER82]. When subjects were given extensive training in both schemes, no appreciable differences were found. However, when subjects were given only one page of instructions, the performance of both programmers and non-programmers was consistently better with the Pascal syntax diagrams. The experimenters' conjecture was that the one-page instructions were typical of the type and depth of information a person would get in a real world situation.

A graphical approach to the representation of the entire interface, not just its interaction language component, attempts to extend data flow diagrams to specify the end-user's model of the dynamic nature of a system [WELLM84]. Both user inputs and system outputs, as well as control flow, tasks, and files involved in processing, have notational symbols. Once a system is represented using this scheme, a dialogue simulator and a dialogue processor can
execute the specification. A first version of a dialogue prototyper has been implemented, but no comments on its performance were given. Details of such issues as screen handling, formatting, input definition and validation, were also missing.

3.3 OTHER RESEARCH IN LANGUAGE SPECIFICATION

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. Attempts to predict ease of use through the use of analytical tools is also being researched [REISP83a, REISP83b]. Formally speci-
fying the constructs of a language with an axiomatic approach has also been used to evaluate strengths, weaknesses, and inconsistencies in interactive systems [EMBLD78].

Another approach to specifying human-computer interfaces has been done algorithmically using the constructs of a high-level procedural language (i.e., iteration, conditionals, etc.) to describe both the human and the computer actions which must take place, and the flow of control between them [LINDT83, LINDT85]. This representation, called a "dialogue structure", procedurally defines interface semantics and protocols (rules which determine interrelationships of interface components), but not the syntax. The McCabe Metric, or cyclomatic number, a measure of software complexity, is then applied to the resultant algorithms to determine the potential usability of the interface.

3.4 LANGUAGE IMPLEMENTATION AND RECOGNITION TOOLS

For implementation of programming languages, a number of automated systems exist. The Unix system contains two of the best known automated tools for 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.

3.5 INTERACTIVE TOOLS FOR DIALOGUE DESIGN AND IMPLEMENTATION

Among the tools being developed to design, implement, simulate, test, and maintain dialogue are special functions to format text, generate graphics, produce keypad outlines and touch-panel configurations, and manage voice input/output interactions. Other tools allow definition of user input, so that at run-time the user's actions can be accepted, validated, and processed. Many of these are dialogue simulators, or rapid prototypers (e.g., ACT/1 [MASOR83], SIMIC [CLARI83]), and often do not produce an interface that will be used in a production version of the system. Other tools are used as dialogue implementation aids to help a dialogue author produce interfaces with a particular content. Many of the models and specification schemes already discussed
have such interactive tools associated with them (e.g., AIH, USE, [JACOR83a], [BENBI84]). Some other examples of tools for dialogue development are discussed below.

A Dialogue Generator (DIAGEN) [KAISP82] is a generalized software tool for creating an interactive interface which separates the dialogue from the dialogue-driven application program. A specialized DIAGEN language is used by a dialogue author to "program" a scenario which describes the dialogue. This scenario is then interpreted, but its relationship to the computation is not made clear. While many of these ideas were quite progressive for their time (1976-1977), one important issue was not thoroughly resolved. A single run-time message for erroneous user input was hard-wired; the system could only respond "Wrong answer. DIAGEN repeats the question." and the whole sequence was repeated!

TAXIS [MYLOJ80], another early system for designing and programming interactive information systems, has been extended [BARRJ80] with a tool for the description of processes that construct dialogues, modeled with Petri nets.

A system that provides for some dialogue design is Screen Rigel [ROWEL83], a set of input/output features for Rigel, a high-level database programming language. This data-field-level capability for specifying data content, position, and
format is provided in the run-time facility of Rigel, as a design-time activity with respect to a database application program. These specifications are terminal independent and data independent and, to the extent that they produce stored definitions of an interface, they provide a measure of dialogue independence. Screen Rigel, however, is intended for use by the database application programmer, not a dialogue author.

A design for a display management system is proposed in [CARLE83]. This design is based on decomposition of the display management software into three levels of abstraction, each representing part of the overall interface for a user interaction. Definition of each of these parts (frame, structured input/output, and device interfaces) provides the information necessary for an application system to create displays on the screen and to process user inputs. The proposed design attempts to provide a software structure which hides implementation details of data structures, algorithms, and device information, as well as separation of display specifications and input/output functions. The concept appears quite thorough and viable, and an example describes the processes which occur between various elements in the system. However, an explanation of how these elements (e.g., frame specifications) are created is not given; it
can be inferred that they must be programmed in the conventional manner.

A system called BLOX Graphic Builder [BLOX84] is now commercially available for the automatic creation of a graphical human interface, with on-line help and end-user documentation. Screens, menus, graphics input, dynamic displays, and finite state table-driven control are interactively specified; only the application-dependent "action routines" are coded. This appears to be a reasonably powerful system for interactive production of an interface.
Chapter IV

A THEORY OF HUMAN-COMPUTER INTERACTION

"It is a capital mistake to theorize before one has data." Sir Arthur Conan Doyle, *The Memoirs of Sherlock Holmes*

4.1 THE ROLE OF THEORY IN RESEARCH

What really happens when a human sits down to use a computer? The answer to this question is the basis for the formulation of a theory of human-computer interaction. And this theory is the basis for studying the structure and development of human-computer interfaces. The importance of theory in research is incontestable. A plausible general body of principles is needed to explain observable phenomena and their interrelationships. Theory systematizes knowledge; it presents a way of organizing and representing facts [KAPLA64]. It heuristically serves as a tool for understanding those phenomena which can be observed. Theory, then, "is more than a synopsis of the moves that have been played in the game of nature; it also sets forth some idea of the rules of the game, by which the moves become intelligible" [KAPLA64].

48
4.2 A SCENARIO OF HUMAN-COMPUTER INTERACTION

Development of a theory of human-computer interaction began by careful real world observation of humans interacting with computers. During the course of this research, literally dozens of interfaces were observed, to cover the widest possible range of interface types. Some examples of the kinds of interfaces include micro systems (e.g., Macintosh, Lisa, and Star), text editors (e.g., XEDIT, SAM, EDT), database query languages (e.g., DEC's Datatrieve and a dBase II application for the Virginia Tech Computer Science Research Consortium database), real-time simulation systems (e.g., the GENIE carrier-based air traffic controller), large text retrieval systems (e.g., the Virginia Tech Library System and a DEC product for on-line retrieval of documentation), and unreleased operating system interfaces (e.g., the DEC micro-VAX interface), to name a few. These interfaces cover a large variety of styles, techniques, and devices. That which is observed in such interfaces is the surface structure of human-computer interaction; the hidden rules that help to organize and explain what is observed produce its deep structure. These are the basis for the theory.

Following is what an observer might see on the surface of a typical scenario of human-computer interaction, representative of the many interfaces mentioned above:
The human is observed reading some text on the screen. The human then types some characters, following them with a carriage return. Or the human might move a mouse and click one of its buttons. The human then peers intently at the screen, as something is displayed; some objects that an observer might see appearing on the screen include text and graphics. The human might now proceed to type some more characters, and so on.

From this scenario, an observer could claim that human-computer interaction approximates a turn-taking paradigm, alternating between the human and the computer in a continuous sequence. This scenario is expanded a bit further to refine the paradigm:

\[
\begin{array}{c|c|c}
\text{Computer's Turn} & \text{Human's Turn} \\
\hline
[1] \text{Produce display} & \text{See display} \\
[2] \text{Wait} & \text{Think} \\
[3] \text{Accept typing} & \text{Type} \\
[4] \text{Compute} & \text{Wait} \\
[5] \text{Produce display} & \text{See display} \\
[6] \text{Wait} & \text{Think} \\
[7] \text{Accept typing} & \text{Type} \\
\end{array}
\]

Analysis is now added to these observations, seeking patterns and relationships:
[1] seems to be telling the human what input to enter and/or how to enter it. This is the prompt of the dialogue transaction model to be presented in the next chapter.

[2] is a time when the human decides what to do next. Thinking is observable through a pause by the human before typing. It is not explicitly included in the dialogue transaction model because there is nothing the system can or needs to do during this step.

[3] appears to be a request by the human for the computer to perform a task. This is the input of the model.

[4] appears to be where the computer attempts to perform the requested task. Computing is observable through a pause by the system before producing a display, during which time an observer might see flashing panel lights, hear the disk head, etc. These pauses are often, of course, nearly imperceptible. The processing is the *lexical and syntactic validation* and possibly the *semantic (computational) processing* of the human's input in the model.

[5] seems to be the results of the attempted task; sometimes the computer responds that the task could not be understood or could not be performed. This is
either the confirmation or the output transaction of the model, depending upon the specific situation.

[6] is again a time when the human thinks about how to proceed. If [5] indicated that the system did not understand or could not do the requested task (what will be called errors at the syntactic and semantic levels, respectively, in the model), the human must try again in [7]. If [5] produced task results, the human must decide upon the next task in [7]. Again, this thinking process is not explicitly modeled.

Note that while this scenario uses typed input as the way in which the human gives requests to the computer, any form of input is equally applicable.

4.3 A THEORY OF HUMAN-COMPUTER INTERACTION

From this simple scenario of a human using a computer, many of the phenomena of human-computer interaction are observed and from such observations a theory of human-computer interaction proposed. This theory addresses such fundamental questions as: What are the constituents of human-computer interaction? How do these constituents behave, both individually and together? Can a structure be hypothesized to explain the relationships among the constituents?
This theory of human-computer interaction, then, postulates that the observable components of human-computer interaction include those which are essentially human input, those which are essentially computation, and those which are essentially computer output. During the course of human-computer interaction, these components occur in various sequences. One common sequence is user input, followed by computation, followed by display of results, or an Input-Process-Output configuration. Further decomposition of each of these components reveals that it is also composed of other events. The user input, for example, is often preceded by a computer prompt. The input is not always correct or understandable by the computer, and additional dialogue is required to clarify the misunderstanding, until the computer eventually confirms that the input is understood and processes the input request. This confirmation is often implicit; i.e., no explicit computer response means that no error in human input was detected.

Both within and beyond this typical Input-Process-Output (I-P-O) paradigm, the theory of human-computer interaction expands to handle many different sequences and phenomena. Even when an interface is not organized explicitly around this turn-taking approach, the I-P-O sequence still applies at a level of finer granularity, eventually approaching con-
currency among the I, P, and O components. An example of this would be a video game in which human inputs and computer outputs appear, at least on the surface, to be concurrent. Deeper analysis, however, would reveal that inputs and outputs are still occurring sequentially, but in a somewhat finer grained time frame. In addition, humans may interact with the computer through numerous types of input devices, including keyboards, function keys, mice, touch panels, voice equipment, joysticks, bit pad and cursor, etc. Similarly, the computer can respond to the human in numerous formats and with multiple devices; technological advances make the possible realm of input/output techniques almost limitless.

4.4 FUTURE RESEARCH ON THEORY OF HUMAN-COMPUTER INTERACTION

In order to insure that these technological advances do not make this research obsolete, it is important that the theory of human-computer interaction be kept current. This can be done by continuing to observe interfaces and humans using them. Frequent reassessment of the theory of human-computer interaction and extensions to it will be made as necessary to explain each new type of interface.
Chapter V

A MULTI-DIMENSIONAL DIALOGUE TRANSACTION MODEL
OF HUMAN-COMPUTER INTERACTION

"The Rabbit could not claim to be a model of any-
thing, for he didn't know that real rabbits exist-
ed...but once you are Real, you can't become unre-
al again. It lasts for always." Margery
Williams, *The Velveteen Rabbit*

5.1 MOTIVATION AND REQUIREMENTS FOR A MODEL

The surge of interest and technological advances in hu-
man-computer interfaces have led to the development of com-
plex, sophisticated interfaces. But the advance of research
to support interface development has lagged behind. Few mo-
dels, either theoretical or practical, exist to guide in hu-
man-computer interface design. Without such models, the
components of an interface are an overwhelming collection of
unrelated prompts, displays, messages, expressions, inputs,
and devices with little cohesion.

The *multi-dimensional dialogue transaction model* presented in
this chapter is a manifestation of the theory presented in
Chapter 4, and is the core of this dissertation research.
This model is a structural, descriptive representation of
what happens when humans interact with computers. It is
based on three traditional levels of language: semantic,
syntactic, and lexical. As already discussed (in Chapter 3.2), this approach is not new, but it has not heretofore been exploited to its full potential. While the model is general in nature, it is also used as the specific framework for a dialogue engineering methodology (presented in Chapter 7) and a set of automated tools (discussed in Chapter 8) for use by a dialogue author under DMS.

5.1.1 Levels of Such a Model

A comprehensive model for human-computer interaction must address, at several levels, the specific needs of the dialogue author while designing and implementing interfaces. Figure 3 shows five such levels. While most dialogue models portray interaction as being between a human user and a computer, this abstraction 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. Each of the five levels has its own special needs:

1. **Dialogue author's conceptual level** -- A mental design-time image of an interface which is formulated from the dialogue author's knowledge of application system-specific interface requirements and of human factors principles.
Figure 3: Levels of a Model of Human-Computer Interaction
2. *Interface specification level* -- The design-time mechanism by which the dialogue author defines each specific interface component. If automated (e.g., as in AIDE), the system then creates a storable, interpretable representation of each interface component definition. BNF and state transition diagrams are typical forms of such a specification level.

3. *Internal representation level* -- The interpretable, stored representation which is the product of the automated tools applied at the interface specification level.

4. *Interface documentation level* -- The means used for documentation of the interface specification. The application system end-user may access it at application system run-time, as a training and/or help-type representation of the interface. It may consist of both an on- or off-line user's manual and other specific documentation. IBM or Pascal "railroad" notation are typical forms of this level.

5. *End-user's conceptual level* -- The application end-user's mental image of the interface. As run-time communication between user and system takes place, the user builds a model to understand the use of the system, including syntax and semantics, as well as personal strategies for use of the system. If this image pro-
vides a reasonable match to the author's conceptual image, the two (dialogue author and end-user) have communicated well, through the interface.

The model presented in this dissertation is essentially level (2), the dialogue author's implementation specification level. The thesis of this research is that the best model for a dialogue author to have at interface design-time is the conceptual run-time model of the end-user. The multi-dimensional dialogue transaction model was developed by observing end-users interacting with numerous existing interfaces, taxonomizing the interface components, and establishing relationships among these components. Then, by "working backwards", the multi-dimensional dialogue transaction model emerged as the description of what the dialogue author must produce at interface design-time. This resultant model of human-computer interaction, when embodied in dialogue author support tools, has been shown in preliminary studies (discussed in Chapter 9) to facilitate the design of user interfaces.
5.1.2 Requirements for Such a Model

The multi-dimensional dialogue transaction model can be used to express the external viewpoint of the end-user and the human factors expert studying the end-user, the dialogue-design-oriented viewpoint of the dialogue author, and even the computationally-oriented viewpoint of the programmer. From the end-user's viewpoint, the purpose of a dialogue transaction is to provide the means for entering a command or other information to the system, or to display an output or computational results to the end-user. This external view also conveys to the human factors person the behavioral nature of the interface. To the dialogue author, a dialogue transaction is a mechanism for defining those things (such as user prompt, instructions, "help" information, input definitions, error messages, etc.) necessary to extract a command or other information from the end-user, or to display output results. The computational programmer sees a transaction as the means for determining what command to process, without having to think about syntax checking or how best to interact with the user. The perspective taken herein is that of the dialogue author.

An important concern for a human-computer interface model and resultant specification scheme is that it be human-oriented. In addition, there are several other re-
requirements which must be addressed. While many of these are obvious, they bear mentioning. Any model and specification scheme must be powerful and unambiguous, yet should not pose any limitations on implementation. Model structure should reflect the user's cognitive model of the system, and be able to produce at least a prototype (automatically, if specifications are executable [JACOR83b]). The model and specification must be able to represent all types of interaction sequences and user actions, including those not normally found in programming languages [GIITW83, JOHND83, KAMRA83].

5.2 A MULTI-DIMENSIONAL DIALOGUE TRANSACTION MODEL

5.2.1 Basic Components and Their Relationships

Every exchange of information between a computer and its user follows some specific sequence. The theory of human-computer interaction has shown that one way of viewing this exchange at the highest, or semantic, level is as an Input-Process-Output (I-P-O) sequence. This I-P-O sequence is called a dialogue transaction sequence, which is defined by the ordered triple

< dialogue input transaction, computational processing, dialogue output transaction >
Figure 4: The Multi-Dimensional Dialogue Transaction Model
as shown in Figure 4. A *dialogue input transaction* prompts the user, accepts, and validates the resultant input. The validated inputs are passed to the computational component, processed, and the results are then communicated from the computer to the user through a *dialogue output transaction*. This type of transaction is non-interactive, and involves the development of transactions whose exact form, content, and visual attributes are not known until run-time, as the result of computation. They are discussed in Chapter 5.4. *The dialogue input transaction, referred to hereafter simply as a transaction or dialogue transaction, is the component with which this research effort has been primarily concerned.*

A *transaction* is a sequence of one or more interactions, to extract a valid user input (such as a complete command). This valid user input is a set of linguistically related tokens. The set of all language input definitions for a single transaction of an interface is the *interaction language* (e.g., set of commands) of that transaction. An interface can be considered to be the collection of all dialogue transactions for an application system. Therefore, an interface is also a set of interaction languages.

One command of an interaction language is used in a given run-time instance of that transaction. A transaction is made up of a sequence of *interactions*, each of which is com-
prised of three parts: system syntactic prompt, followed by human token input, followed by system syntactic confirmation. Thus, an interaction is an ordered triple

\[
< \text{syntactic prompt, token input, syntactic confirmation} >
\]

also shown in Figure 4. Any of these parts may be implicit (null) within a given interaction. If the token input is not valid, another pass is made through the same interaction, until a valid input is received from the end-user. The result of each interaction is a single valid token value. Each syntactic prompt can consist of any type of display component, including such pieces as menus, labeled function key outlines, text, graphical objects, forms to be filled in, spoken words from a voice synthesizer, as well as combinations of these. Its function is to request a user input, possible informing the user how properly to make that input. Correspondingly, the token input can be comprised of pieces defined for menu selection, function key selection, command string input, forms completion, or voice input. The function of this part is to accept, process, and validate all user inputs. Syntactic confirmation parts give the user information about the syntactic validity of the input. These confirmations can be comprised of such pieces as text, graphics, and voice.
There are four types of confirmations: positive confirmation, negative confirmation, neutral confirmation, and help. Positive confirmation tells the user that the input was accepted as syntactically valid; it is often implicit (null), with the prompt of the next interaction or the results of processing implying that the input was valid. A negative confirmation is a syntactic error message, which may be explicit (e.g., "Invalid input") or implicit (e.g., fill-in-the-blanks that won't!). It is usually followed by another instance of the interaction until a valid input is received from the user. Neutral confirmation expresses uncertainty about the input and asks the user to verify the system's interpretation of assumptions about the intended input. It might be considered a form of negative confirmation, meaning that the system accepted the input but was not able to validate it without further information. Help is treated as a type of confirmation because it is logically similar to the handling of input errors. That is, neither a help request nor an error produces a token value to be returned to the computational component. Rather, the dialogue component displays a message (help or error) and then makes another attempt to obtain a valid token value from the end-user. By handling error messages and help requests as confirmations in the dialogue component, they are removed from
the global control structure of the system, reducing the clutter often found in other representation schemes (e.g., state transition diagrams) which include them there.

An action within an interaction is also an ordered triple

\[ < \text{lexical prompt, lexeme input, lexical confirmation} > \]

which is the expansion of the token input part of an interaction, shown in Figure 4. At this level, a system lexical prompt can be as simple as cursor positioning, and the human lexeme input is a single character (for typed keyboard input) or other lexeme (e.g., function key selection or spoken word). The system lexical confirmation can be as simple as a beep for erroneous input. Any of these can be implicit within a given action. The result of each action is a single valid lexeme value.

In terms of language, a transaction represents the semantic level. This is the end-user's view of what the system, through its dialogue transactions, does. The purpose of a transaction instance is to elicit from the end-user a sequence of one or more linguistically related valid token values necessary to invoke one semantic function. Each token value is the result of an interaction instance, which is the syntactic level. Token values are comprised of one or more lexeme values, derived from action instances which are at
the *lexical* level. A *lexeme* is the smallest independent unit of end-user input. A token, which is a group of one or more lexemes, is the smallest syntactically independent unit of end-user input. This comparison of the dialogue transaction model with languages is presented in more detail in Chapter 6.

5.2.2 *Dimensions of the Model*

In order to allow the form and content of a human-computer interface to be tailored to specific needs of the user and to the state of the dialogue, the basic parts of the model (prompt, input, confirmation) at both the syntactic and lexical levels have five dimensions:

- **f** - the form (device) of the last previous user input (e.g., keyboard, touch panel, voice, and combinations of these)
- **t** - the type of the last previous user input error (i.e., the specific kinds of input errors that can occur)
- **p** - the pass (count) of the last previous user input error (i.e., the number of times a specific type of user input error has occurred)
- **d** - the delimiter of the last previous user input (e.g., <CR>, /. blanks)
- **u** - the user expertise level

Thus, at the syntactic level, there can be multiple alternative forms of interaction parts for a single interaction.
That is, by varying the values of \( f, t, p, d, \) and \( u \) along these dimensions, a single interaction can have a large variety of prompts, input definitions, and confirmations. At the lexical level, similar variations along the dimensions can occur, exactly as they do at the syntactic level. Because of this similarity, dimensions at the lexical level are not addressed here.

These five dimensions allow the development of an interface that is precisely tailored to the needs of the end-user and which will specifically address each situation arising in the interaction between the user and the system. Each dimension represents an aspect of dialogue context at runtime. Combinations of these orthogonal dimensions represent dialogue states resulting from the course which the dialogue has taken. These dimensions are surely not the only ones needed to completely specify all application system interfaces; any dialogue state information upon which form and content depend is potentially another dimension.

5.2.3 Scenarios Showing Use of the Dimensions

Following are two short run-time scenarios from an example system, showing how values vary along the dimensions as the user interacts with the system. The notation used shows each interaction part as a function of the dimension parame-
ter values, e.g. Computer Prompt \([f, t, p, d, u]\). (Note: No claim is made to the "human factoredness" or efficacy of dialogue in these scenarios; their purpose is to illustrate the use of the five dimensions. These commands were chosen to be illustrative of a variety of occurrences in interaction languages; this example will also be used in explaining the dialogue engineering methodology in Chapter 7. The design of the interaction parts of these scenarios by the dialogue author is also omitted.)

Those parts shown in quotes (" ") appear on the screen at run-time (i.e., prompts, confirmations, and what the user enters). Other parts are internal definitions which do not appear on the screen at run-time but are given here for completeness (i.e., input definitions). A zero (0) used as a parameter value means that parameter has been initialized to a null value and has not yet been set to something else.

**Scenario 7**

The purpose of this scenario is to illustrate the use of the \(f, t,\) and \(p\) dimensions. Note that this scenario shows repeated instances of the same interaction, finally resulting in a valid input. It shows how negative confirmation (error) messages within a single interaction can become more and more informative as the end-user continues to make erroneous inputs, and how the input definition can even be changed as a result of repeated errors.
Interaction 1:

Computer Prompt [0, 0, 0, 0, novice] :

"Please enter a command from this menu, followed by a <CR>:

cop copy
def define
res reserve

You may also use the labeled function keys or speak the command."

Input Definition [0, 0, 0, 0, novice] :

cop <CR>, def <CR>, res <CR> from the keyboard function keys labeled 'C', 'D', 'R' spoken words 'copy', 'define', 'reserve'

User enters :

"re2" <CR>

Computer Negative Confirmation [keybd, numeric input, 1, <CR>, novice] :

"This is not a valid input."

Computer Prompt [keybd, numeric input, 1, <CR>, novice] :

"Please try again."

Input Definition [keybd, numeric input, 1, <CR>, novice] :

same as Input Definition [0, 0, 0, 0, novice]

User enters :

"re3" <CR>

Computer Negative Confirmation [keybd, numeric input, 2, <CR>, novice] :

"Numeric characters are not valid inputs."
Computer Prompt [keybd, numeric input, 2, <CR>, novice]:

"Please enter one of the choices from the menu."

Input Definition [keybd, numeric input, 2, <CR>, novice] :
same as Input Definition [0, 0, 0, 0, novice]

User enters :

"re2" <CR>

Computer Negative Confirmation [keybd, numeric input, 3, <CR>, novice] :

"You have entered a numeric character again."

Computer Prompt [keybd, numeric input, 3, <CR>, novice] :

"Typed input is no longer valid. Please press one of the labeled function keys 'C', 'D', or 'R', or speak a command.

Input Definition [keybd, numeric input, 3, <CR>, novice] :

function keys labeled 'C', 'D', 'R'
spoken words 'copy', 'define', 'reserve'

User enters :

unlabeled key on function keypad

Computer Negative Confirmation [funct-key, unlabeled key, 1, 0, novice] :

system beeps once

Computer Prompt [funct-key, unlabeled key, 1, 0, novice] :

"Please press a labeled key."

Input Definition [funct-key, unlabeled key, 1, 0, novice] :

same as Input Definition [keybd, numeric input, 3, <CR>, novice]

User enters :

key labeled 'HELP'
Computer Help [funct-key, unlabeled key, 1, 0, novice]:

...good help message...

Computer Prompt [funct-key, unlabeled key, 1, 0, novice]:

"Please press a labeled key."

Input Definition [funct-key, unlabeled key, 1, 0, novice]:

same as Input Definition [keybd, numeric input, 3, <CR>, novice]

User enters:

key labeled 'R'

Computer Positive Confirmation [funct-key, unlabeled key, 1, 0, novice]:

"You wish to reserve a seat."

Computer Prompt [funct-key, 0, 0, 0, novice]:

"Enter the departure city."

Scenario 2

This is a different instance of an end-user interacting with the same interface as in Scenario 1 (note that the initial Computer Prompt and Input Definition are the same). The purpose of this scenario is to illustrate the use of dimension d. Unlike Scenario 1, this scenario goes through several interactions. It shows how (in Scenario 2a) the interface can prompt the user for the next input token (interaction) of a command when the user enters <CR> before
all tokens of the command are entered, and how (in Scenario 2b), if the user enters all needed tokens without <CR> between them, the system will not issue the prompts.

Scenario 2a.

Interaction 1:

Computer Prompt [0, 0, 0, 0, novice] :
　"Please enter a command from this menu:

   cop     copy
def     define
res     reserve

You may also use the labeled function keys or speak the command."

Input Definition [0, 0, 0, 0, novice] :

   cop <CR>, def <CR>, res <CR> from the keyboard
   function keys labeled 'C', 'D', 'R'
   spoken words 'copy', 'define', 'reserve'

User enters :

   "cop" <CR>

Computer Positive Confirmation [keybd, 0, 0, <CR>, novice] :

   none

Interaction 2:

Computer Prompt [keybd, 0, 0, <CR>, novice] :

   "Enter filename to be copied from:"

Input Definition [keybd, 0, 0, <CR>, novice] :

   alpha-numeric string of less than 20 characters
User enters:

"xyz" <CR>

Computer Positive Confirmation [keybd, 0, 0, <CR>, novice] :

none

Interaction 3:

Computer Prompt [keybd, 0, 0, <CR>, novice] :

"Enter filename to be copied to:"

Input Definition [keybd, 0, 0, <CR>, novice] :

same as Interaction 2 Input
Definition [keybd, 0, 0, <CR>, novice]

User enters :

"abc" <CR>

Scenario 2b.

Interactions 1 and 2:

Computer Prompt [0, 0, 0, 0, novice] :

same as Interaction 1 Computer
Prompt [0, 0, 0, 0, novice]

Input Definition [0, 0, 0, 0, novice] :

same as Interaction 1 Input
Definition [0, 0, 0, 0, novice]

User enters :

"cop xyz" <CR>

Interaction 2:

Note that the user entered the input for Interaction 2 when "xyz" was entered; note that no prompt appeared afte-
without <CR>. The definition of this prompt for Interaction 2 is:

Computer Prompt [keybd, 0, 0, blanks, novice] :

    none

Comparing this with the previous Interaction 2 Computer Prompt [keybd, 0, 0, <CR>, novice] shows the difference to be, of course, the value of dimension d.

Computer Positive Confirmation [keybd, 0, 0, <CR>, novice] :

    none

Interaction 3:

Computer Prompt [keybd, 0, 0, <CR>, novice] :

"Enter filename to be copied to: "

Note that if the user had initially entered "cop xyz abc" the prompt for Interaction 3 would have also been defined not to appear.

The dimension f of a single interaction allows the dialogue author to design prompts and confirmations that respond to the end-user in a way that is appropriate for the form of input (f) the human used in the previous interaction. In Scenario 1, the negative confirmations respond specifically to erroneous keyboard (menu) input and erroneous function key input. Without dimension f, the form of input would not be known and only a more general, less in-
formative error message such as "Erroneous input" could be given.

The dimensions \( t \) and \( p \) allow various confirmation messages for an interaction to be based on specific user input error types \( (t) \) and the number of times, or passes, \( (p) \) a specific error type has occurred consecutively. Because the language input is explicitly defined for each interaction, the exact set of possible input error types can be determined for each interaction. Again in Scenario 1, the language input is defined (in part) to be function key "R", typed characters "res", and spoken word "reserve". Entering in a numeric character is obviously one possible type of input error. Dimension \( p \) allows different negative confirmations as a user continues to repeat the same type of error, shown in the "re2", "re3", "re2" sequence of errors in Scenario 1.

The dimension \( d \) of a single interaction allows the design of parts of subsequent interactions to be based on various values that can be used for delimiters. In Scenario 2, use of \(<\text{CR}>\) as a delimiter at the end of each interaction caused a prompt to appear to give the human information about what the next input should be. When the user entered tokens consecutively (without \(<\text{CR}>\) between them), the prompts did not appear. Combining this dimension with dimensions \( f, t, \) and
\( p \) obviously allows great flexibility in the amount of information given to the user.

Dimension \( u \) allows multiple interfaces for several levels of user expertise. The initial value of \( u \) is taken from a user profile and can be varied by the user during execution of the application system. This dimension also facilitates adaptive interfaces by allowing the execution-time system to change the level of user expertise as the system executes.

5.2.4 Extensions to the Basic Model

Dialogue Validation

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 value which is at least lexically and syntactically valid. 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 (i.e., repeat the interaction or action) until a valid one is received. This ensures that the resultant token value is usable by the computational component.

Therefore, under this model of human-computer interaction, interaction language processing involves action-at-a-time (i.e., lexical) checking of each user input action.
This type of processing will be discussed in detail in Chapter 6. Because lexical and syntactic errors are inseparable, no distinction is made between lexical validation and syntactic validation; both are referred to as dynamic dialogue validation.

Interruptible Transactions

Often, semantic checking (e.g., to make sure a token value exists in an application database) is also desired. This is not the invocation of the semantic action defined by the functionality of the command, but rather a check to determine whether a token is semantically valid. When this occurs, computation (e.g., to search the database for the input token value) is needed; this semantic check is not done on the dialogue side. This results in a significant extension to the syntactic level of the model, as shown in Figure 5. (The special notation used here is part of the dialogue engineering methodology, explained in Chapter 8). The transaction in which this semantic check is desired is called an interruptible transaction. This current transaction is interrupted (temporarily suspended) and a special sub-transaction state is entered in which the semantic checking is done. A sub-transaction control structure temporarily assumes control (i.e., global control of the application system is not cluttered with this special sequencing).
Figure 5: Interruptible Transaction for Semantic Validation
At this point, the semantic validation is made. If the token is semantically valid, control returns to the interrupted transaction where it was suspended and normal execution resumes. If the token is not semantically valid, action is taken to resolve the problem. Resolution may be as simple as returning directly to the interrupted transaction without attempting to solve the problem or as complicated as unconstrained dialogue transaction sequences (i.e., effectively another special system of dialogue and computation during the interruption of this transaction). Once resolution is complete, control again returns to the interrupted transaction and execution continues.

An important constraint exists on this sub-transaction semantic error resolution. It can be used only to determine how the end-user wishes to proceed when a semantic error has occurred. That is, it cannot be used to get a token value for the interaction of the transaction which was interrupted. The system will return to the interrupted transaction and initiate another instance of the interaction for which the token value was semantically invalid. This new instance of the interaction will extract another token value which will be syntactically and semantically checked again.
Confirmation Points

The time at which dialogue validation is done and reported to the end-user is called the confirmation point for a given input. Dialogue validation at the action level can only have an action confirmation point. At the interaction level, dialogue validation can have either an interaction confirmation point or a transaction confirmation point. An example of each of these confirmation points is shown in the example below, where "inxo" has been incorrectly entered for token value "info":

```
show inxo vpi
  ↑↑  ↑
   |   _ transaction confirmation point
   |   _ interaction confirmation point
   _ action confirmation point
```

The action confirmation point informs the user of an error as soon as the first erroneous character (action) is input, based on the prefixes in the set of all possible values for this token. An interaction confirmation point informs of the error immediately at the end of any erroneous token (interaction). An interaction confirmation point always results in a (possibly implicit) syntactic confirmation; it may also produce a (possibly implicit) semantic confirmation if semantic validation was also specified in an interruptible transaction. Note that syntactic validation must occur
before semantic validation. A transaction confirmation point reports the error only at the end of the complete command (transaction). The complete multi-dimensional dialogue transaction model, including the sub-transaction semantic validation and all confirmation points, is shown in Figures 6 and 7. Figure 6 shows the decomposition of a dialogue input transaction at the syntactic level, and Figure 7 shows the decomposition of the token input at the lexical level.

While the emphasis of this discussion has appeared to be for user inputs through a keyboard, it is again important to note that it is equally relevant to the definition and processing of other user input forms. For input forms other than keyboard, lexical and syntactic processing are equivalent. The lexeme produced, for example, by a touch panel selection, a pointing device, or a mouse click, is also the corresponding raw token value for the current interaction. Because of this, these inputs are not received a character-at-a-time, but rather a token-at-a-time, and thus the lexical (character) and syntactic (token) processing levels are identical. Interaction and transaction confirmation points are also equally applicable to all types of user input devices.
Figure 6: Complete Multi-Dimensional Dialogue Transaction Model: Syntactic Level
Figure 7: Complete Multi-Dimensional Dialogue Transaction Model: Lexical Level
Dialogue Variables

While the dimensions allow the dialogue author great flexibility in the design of interfaces, other conditions on the state of the dialogue are sometimes needed to completely define an interface. Like the dimensions, a dialogue variable gives information about the state of the dialogue upon which dialogue form and/or content depend. In addition to dimension-like conditions, input token values are another form of dialogue variables. An example of this would be a user input of "cat" that was not syntactically valid with a corresponding negative confirmation of "'cat' is not a valid input". The input token value "cat" had to be stored in a dialogue variable in order to be echoed to the end-user in the error message at run-time. These dialogue variables, as well as the dimensions, could be implemented using production rules, with dialogue form and content determined by the run-time evaluation of a set of these rules.

Chaining

Often parts of a new transaction may make use of the form and content of a previous transaction. This is called chaining, when a dialogue output transaction (or modification of it, such as produced by adding to it or erasing part of
it) is used as (part of) the prompt and succeeding input definition for a dialogue input transaction. The transaction level is the highest level of chaining; chaining can also occur between two interactions, and even from output transactions to interactions. When the input definition is a result of chaining, connections must be made from the chained prompt to the input value.

5.3 IMPLICATIONS OF THE MODEL

As previously stated, the multi-dimensional dialogue transaction model has been used in DMS to provide the framework for a dialogue engineering methodology and a set of interactive dialogue design tools to be used by a dialogue author. The dialogue engineering methodology provides the dialogue author with procedures and a specification notation for developing human-computer dialogues; i.e., for developing dialogue transactions. By explicitly specifying each part of each interaction, the dialogue author can completely define a transaction. By varying the values of the dimensions appropriately, a precise definition of each prompt, input, and confirmation is given. An example of the use of the dialogue engineering methodology is given in Chapter 7.
The Author's Interactive Dialogue Environment (AIDE) of DMS is an integrated set of interactive tools for defining all the components described in the dialogue transaction model. Sequencing in AIDE is guided by the relationships among model components. AIDE contains such tools as a menu formatter, a function key formatter, a graphics formatter, a text formatter, a voice formatter, a forms formatter, and corresponding tools for defining the language input for each of these formatters. Again, AIDE is presented in detail in Chapter 8.

Although it is used as the basis for developing the dialogue of human-computer interfaces, the model itself is independent of specific human factors principles. Other models of human-computer interaction often attempt to incorporate specific principles of dialogue design, to guide the dialogue author. But these principles are subject to evaluation and evolution. Thus, as the principles change, models based on them become invalid. This multi-dimensional dialogue transaction model does not, therefore, presume to incorporate human factors guidelines. Because of this, the dialogue author can, for example, create a "bad" negative confirmation (e.g., "Invalid input"). This means that the dialogue author, and not the model, did something wrong, according to "good" dialogue design principles. The dialogue
author could just as easily have created a negative confirmation message that provided detailed feedback (e.g., "Only alphabetic input is valid here."). A dialogue design expert system is being considered for AIDE, to help with such design decisions while remaining independent from the transaction model. This model attempts to describe the function and structure of human-computer dialogue, and not its specific form and content based on design principles which are subject to change and personal interpretation.

The complexity of the multi-dimensional dialogue transaction model results from the complexity which is potential in state-of-the-art human-computer interfaces. This complexity is offset by the completeness with which the model describes the elements of human-computer interaction, and is relieved by using this model to organize the task into clearly defined components using a structured, cohesive approach. The five dimensions account for numerous major aspects of human-computer interfaces that have not been explicitly included in existing models of human-computer interaction.
5.4 DIALOGUE OUTPUT TRANSACTIONS

Computational results are presented to the end-user through non-interactive dialogue output transactions, which was shown in Figure 4. These transactions must be formatted dynamically because their form, content, and visual attributes are not typically known until execution-time. These computational results must be formatted as text, graphics, histograms, pie charts, graphs, and so forth. The general form of these transactions is known by the dialogue author at design-time, but the data-dependent values which determine their content are the results of run-time computation (e.g., the length of the bars in a histogram or the results of a database search). The AIDE design-time interface for dialogue output transactions and the corresponding run-time dynamic display executor are not in the purview of this research effort. They are being developed by other members of the DMS research team [SIOCA84], using a rule-based approach to specification.

5.5 FUTURE MODELING RESEARCH

Numerous aspects of the modeling research will provide future research issues. Foremost among these is the granularity of the I-P-O paradigm for those systems in which the interface is not so easily delineated from the computational
functionality as the strict I-P-O sequence implies. The modeling of dialogue output transactions is also very important, so that the design of this type of transaction by the dialogue author will be consistent with the design of dialogue input transactions. The calling mechanisms for interruptible transactions and the sub-transaction control structure must be solidified. Further work on dialogue variables and chaining is also needed to better understand the scope and power of these concepts. Other future additions to the modeling will include time dynamics issues, an example of which is expirable transactions that terminate automatically if the end-user does not enter an input within a certain time.
Chapter VI

THE RELATIONSHIP OF THE MODEL TO PROGRAMMING LANGUAGES

"The greater part of this world's troubles are due to questions of grammar." Montaigne, Essays

6.1 INTERACTION LANGUAGES VERSUS PROGRAMMING LANGUAGES: THEORETICAL COMPARISONS

6.1.1 Languages, Grammars, and Tokens

In formal language theory, a language is a (possibly infinite) set of finite length symbol (character) strings formed from a specific finite alphabet. A language may be defined by a generative grammar which defines the relationships among the symbols for all possible syntactically correct strings. In programming languages those symbols are grouped into syntactic elements called tokens. A language can also be defined by a parser (recognizer) that determines whether sentences are members of the language. The generative rules of a grammar are collectively called the syntax of the language. This syntax determines which character string formations are allowable, but gives no indication of the meaning (semantics) of those character strings.
6.1.2 *Lexical versus Syntactic Rules*

In programming languages rarely is there made a distinction between lexical and syntactic rules. Both kinds of rules appear in the grammars of languages. Consider this example grammar for a very limited hypothetical programming language:

\[
\begin{align*}
(1) & \quad \text{<statement>} ::= \text{<name>} \text{<value>} \\
(2) & \quad \text{<name>} ::= \text{ADD} \mid \text{SUBTRACT} \\
(3) & \quad \text{<value>} ::= \text{<digit>} \mid \text{<digit>} \text{<value>} \\
(4) & \quad \text{<digit>} ::= 0 \mid 1 \mid 2 \mid \ldots \mid 9
\end{align*}
\]

Syntax is the description of allowable *inter-token* relationships. If the rules of this example are to be syntax only, it must be the case that individual "digits" are tokens. But it is more useful to consider tokens as being word-like groups of characters (e.g., ADD, SUBTRACT, 9999). Under this grouping interpretation, only rule (1) is purely syntax and rules (2), (3), and (4) are *lexical* rules which govern *intra-token* composition. That is, the lexical rules provide *data typing* definitions for the tokens. The individual characters are *lexeme* values and are grouped together to make up the token values ("names" and "values") according to rules (2), (3), and (4).
6.1.3 Lexical versus Syntactic Token Values

So far, token values have been described as strings of lexeme values and as syntactic elements of interaction language grammars, with no clear distinction being made between the roles of these language elements. It is, however, useful to make such a distinction. An immediate advantage is that it allows the interaction language grammar (syntactic) definition to be independent of device type, input form, interaction style, and user actions. Another advantage is that it allows multiple input forms to correspond to the same syntactic token. For example, BLUE, X'FF', 137, PF1-key, the coordinates from a region of a touch panel, and a specific output from a voice recognizer can be different input forms which all map to the same syntactic token value. One can define many-to-one transformations from the lexical (raw) token value representations to the corresponding syntactic (normalized) token values. The broad variation of possible lexical token values occurring in the external dialogue (between the interface and the end-user) is relevant only to the dialogue author, not to the programmer. An interaction language grammar is defined in terms of the syntactic token values; the dialogue author and the application programmer must agree that only the syntactic token values occur in the internal dialogue (between the interface and the application...
software). This supports dialogue independence by allowing the author and the programmer to agree on an interaction language independently of format and input/output devices. If, for example, the author decides to change the input form of an interaction from programmed function keys to a menu, the normalization transformation hides the change from the programmer.

6.2 RELATIONSHIP OF THE MODEL TO PROGRAMMING LANGUAGES

The relationships between the multi-dimensional dialogue transaction model and programming languages are summarized here:

<table>
<thead>
<tr>
<th>Interaction Languages</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>set of languages</td>
</tr>
<tr>
<td>transaction (a set of commands)</td>
<td>language</td>
</tr>
<tr>
<td>transaction instance (a specific command)</td>
<td>specific string or sentence</td>
</tr>
<tr>
<td>sub-command</td>
<td>non-terminal</td>
</tr>
<tr>
<td>interaction (a syntactic token)</td>
<td>terminal symbol</td>
</tr>
<tr>
<td>interaction instance (a token value)</td>
<td>specific word</td>
</tr>
<tr>
<td>action (a lexeme)</td>
<td>alphabet</td>
</tr>
<tr>
<td>action instance (a lexeme value)</td>
<td>specific symbol or character</td>
</tr>
</tbody>
</table>
6.3 INTERACTION VERSUS PROGRAMMING LANGUAGES: PROCESSING COMPARISONS

6.3.1 Static versus Dynamic Processing

Source code analysis for a program has two aspects: lexical analysis and syntactic analysis, or parsing. In single pass compilers the lexical analyzer is called whenever the parser needs another token. The process illustrates the close link between the lexical and the syntactic composition of programming languages. The lexical analyzer reads the next token (the input string lexemes up to the next delimiter), determines the data type of the value, and, for a given token name, returns a pair \(<\text{token type}, \text{token value}\>\) [AHOA77]. The syntactic analyzer then checks this pair against the syntactic rules for that language, as shown in Figure 8.

Programming languages are usually interpreted or compiled a whole statement (sentence) at a time and the difference between lexical rules and syntactic rules is not important. In interaction languages, however, processing can be an action(lexeme)-at-a-time and the distinction, again, is useful. For example, in DMS, under the multi-dimensional dialogue transaction model, the validation of interaction languages is done dynamically, an action-at-a-time (e.g., as each character is typed). The difference is that each character is compared against an expected token type for a spe-
Figure 8: Programming Language (Static) Processing
pecific position in the input string, as shown in Figure 9. That is, for interaction languages, the syntactic component of the definition tells the lexical processing what token type to expect. For example, for a simple "copy" command:

```
  copy filename filetype newfilename newfiletype
```

the value provided by the end-user at run-time for "filename", in the second syntactic token position, can be constrained lexically to be an alphanumeric string, beginning with an alpha character, and of length less than or equal to eight characters. Because of this interplay between lexical and syntactic processing, a dialogue error can be viewed as either lexical or syntactic. For example, if "8abc" is entered by the end-user for "filename" in the copy command illustrated above, it is a lexical error because it does not follow the lexical rules for "filename". It also can be seen as a syntactic error, since "8abc" cannot be a "filename" and a "filename" is required syntactically in that position of the command. Thus, lexical and syntactic errors are inseparable. A lexical error automatically causes a syntactic error, because the wrong character type produces the wrong token type. Similarly, a syntactic error means that a lexical error has occurred.
Figure 9: Interaction Language (Dynamic) Processing
6.4 OTHER PROCESSING COMPARISONS

Another area of comparison between interaction language and programming language processing is that of delimiters. Delimiters in a programming language are typically pre-defined, such as ; or , or ( or ) or "begin" or "end" [AHOA77]. During lexical analysis, they are sorted out of the input string along with the tokens. In an interaction language, under the multi-dimensional dialogue transaction model, delimiters are used to denote the end of a token. They are not, however, pre-defined for an entire interface or even a whole command, but can be specified for each token (interaction) by the dialogue author at design-time.

Several other differences between programming languages and interaction languages exist, and contribute to the complexity of interaction language definition and processing. Programming languages do not have interleaved input and output, as do interaction languages. For example, dialogue input transactions also contain output (e.g., prompts and error messages) which alternates with user input, but the sole purpose of the transaction is nonetheless still to obtain end-user input. Specification of tokens which may have no input time sequence ordering (e.g., in a form-filling interface) is unique to interaction languages. Also, interaction languages may have different processing algorithms at the
lexical and the syntactic levels (e.g., token completion at the lexical level, and spelling correction at the syntactic level). Other attributes of interaction languages which programming languages do not have include input device type, input position, input color, input echo, whether tokens are required or optional, and default values for tokens.
Chapter VII

A DIALOGUE ENGINEERING METHODOLOGY FOR INTERFACE DEVELOPMENT

"Man must evolve for all human conflict a method which rejects revenge, aggression, and retaliation. The foundation of such a method is love."
Martin Luther King, Jr., accepting the Nobel Peace Prize

7.1 HOW THIS METHODOLOGY RELATES TO SOFTWARE ENGINEERING METHODOLOGY

A dialogue engineering methodology is to dialogue design and construction what a software engineering methodology is to software design and construction. A software engineering methodology for system development is a set of procedures for constructing a system and a notation for the representation of its design. It allows the top-down decomposition of a system into various levels of abstraction. It provides a notational scheme for representing system functions, and the data flow and control flow among them. It provides for the basic constructs of sequencing, iteration, and conditional control flow. It helps control the complexity of system design by breaking the system down into cognitively tractable pieces, encouraging modularity. As an example, the DMS SUPERvisory Methodology And Notation (SUPERMAN), discussed in
Chapter 7.3, directs the development of the structure of the entire human-computer system. Similarly, a dialogue engineering methodology provides these same features for the development of the dialogue component of the total human-computer system.

7.2 HOW THIS METHODOLOGY RELATES TO THE TRANSACTION MODEL AND AIDE

The dialogue engineering methodology presented in this chapter is based on the multi-dimensional dialogue transaction model. The model codifies the elements of dialogue and their relationships; the dialogue engineering methodology provides the dialogue author with an approach for developing these elements at dialogue design-time. Like a software engineering methodology, the dialogue engineering methodology helps the dialogue author control the task of developing the dialogue for a human-computer system by breaking it into the manageable parts specified in the model. It has appropriate representational notations for the various dialogue elements, their relationships, and properties. It allows the dialogue author to decompose dialogue transactions (circles in SUPERMAN's notation) into their interactions, and interactions into their parts: prompts, inputs, and confirmations. It provides a forms-based scheme for specifying all dimensions, attributes, and properties of these parts.
This discussion of a forms-based approach to the dialogue engineering methodology is admittedly not complete. These forms are a vehicle for describing the methodology. They could be implemented either as a set of paper documents, or as a series of screens in an interactive tool such as AIDE. The forms shown herein do not cover all the details that are elicited interactively by AIDE, including such things as specific device information, echoing, more precise delimiter definitions (including some context-sensitive-like information), and handling of semantic checks (interruptible transactions). This information could be incorporated into the paper forms, by making every screen in AIDE (future versions) a form. The goal here, however, is not to be able to specify every minute detail of an interface. Rather, the goal is to introduce a conceptual approach to a dialogue engineering methodology, one that can be expanded and modified as its viability is tested. Future research will attempt to extend this dialogue engineering methodology to make it independent of AIDE or other automated tools.
7.3 AN OVERVIEW OF SUPERMAN

Because this dialogue engineering methodology is a complement to SUPERMAN, a brief overview of SUPERMAN is necessary. A thorough description of SUPERMAN, including examples of its use, is given in [YUNTT85]. SUPERMAN's main features are that it directly supports the separation of the dialogue author and application programmer roles and that it embodies both data flow and control flow in a single unified system representation at all levels of development.

A representation called the supervisory structure is the basic component of SUPERMAN. The supervisory structure is a hierarchy of supervisory cells, shown in Figure 10 (from [YUNTT85]), each of which represents the subfunctions of a single supervisory function. The sequence of subfunctions is shown as a supervised flow diagram (SFD) which indicates both control flow and data flow among the subfunctions. The supervisory function defines "what" is to be done; the SFD defines "how" it is to be done. A key concept is that the administration of data flow and control flow among subfunctions is performed in the supervisory function of each SFD, by making decisions and calling the subfunctions. Each subfunction can then be a supervisory function of supervisory cells at the next level down in the hierarchy. Terminal nodes in the supervisory structure are worker func-
Figure 10: A Structure of Supervisory Cells
tions, which perform single dialogue or computational operations. The set of all supervisory functions comprises the control structure of the entire application system.

The basic graphical function symbols of SUPERMAN, shown in Figure 11 (from [YUNTT85]), are very simple. A circle inscribed within a square is called a *dialogue-computation function* (*D-C function*) and is a high-level software function composed of both dialogue and computational functions. A D-C function is always a supervisory function, and as such, may contain other D-C functions, dialogue functions, and computational functions. A *computational function*, represented by a square, is a software function that performs only computation; it is constructed by an application programmer. A *dialogue function* is represented by a circle, and is a software function that provides communication between the human and the computer. Each circle in an SFD represents a dialogue transaction (either an input transaction or an output transaction); the dialogue author uses AIDE to construct dialogue transactions. It is the refinement of the dialogue transactions that the dialogue engineering methodology addresses.
Figure 11: Graphical Function Symbols of SUPERMAN
7.4 A DIALOGUE ENGINEERING METHODOLOGY

7.4.1 Hierarchy of Dialogue Elements

As already discussed, the use of this dialogue engineering methodology to develop dialogue transactions is guided by its organization around the dialogue elements of the multi-dimensional dialogue transaction model. Figure 12 shows the hierarchy of dialogue elements which establishes the terminology necessary for discussion of each element and its definition, and illustrates the relationships among them. Understanding of the dialogue engineering methodology (as well as AIDE, in Chapter 8) will be enhanced by this figure. Each transaction can contain several interactions, and each interaction is composed of up to three parts (prompt, input, confirmation). Each prompt part can be made up of various pieces, including pieces having any or all of these styles: list menu, labeled keypad outline, text, graphics, forms to be filled in, and voice output. Correspondingly, the language input part can be composed of pieces featuring menu selection, keypad key selection, command string input, form completion, and voice input. Confirmation parts are comprised of textual, graphical, and voice pieces.
Figure 12: Hierarchy of Dialogue Elements within the Model
7.4.2 Notation

The conventions used in SUPERMAN have been adapted to the dialogue engineering methodology whenever possible, to maintain consistency and provide an integrated approach to design throughout system development. In particular, the supervisory concept still applies. Control flow is represented by solid lines connecting the functions. Control flow in an SFD can consist of the three fundamental constructs of sequencing (denoted by solid lines), decision (indicated by a diamond shape), and iteration (denoted by a feedback loop). Decision predicates for control flow are written near the control lines and are enclosed in <brackets>. A triangle represents the return of control from an SFD function to a higher-level supervisory function.

To completely specify a dialogue transaction, special notation is needed for several components of dialogue, as shown in Figure 13. An interaction is represented by a pair of concentric circles. A dashed circle represents a group of interactions (not an entire transaction) that will be decomposed into single interactions. A box inscribed in a circle is a sub-transaction dialogue-computational function, representing the semantic validation and resolution function of an interruptible transaction (discussed in Chapter 5.2.4).
Figure 13: Special Notation for Dialogue Components
Two control flow considerations specific to dialogue are order independence and ellipses. *Order independence* means that some tokens of a command can be entered by the end-user in any order. An example is seen in the "reserve" command, explained in the example transaction of Chapter 7.4.4. The corresponding notation is a small solid circle on the control flow line at the point at which the order independence is applicable. This small solid circle indicates that every path out of it must be taken, but it does not matter in what order the paths are taken. In comparison, a conditional, represented by a diamond, indicates that only one path out of it can be taken. *Ellipses* are a special case in which "noise words" may be inserted as tokens into an instance of a command (e.g., the "copy" command in the example in Chapter 7.4.4). These "noise words" are ignored by the run-time processor; they simply let the end-user use words to make the command more English-like. The notation for this is three dots inserted in the control flow line between the tokens which may have "noise words" between them.

7.4.3 Procedure

Based on the elements of the multi-dimensional dialogue transaction model, the following procedure outlines the steps for defining a dialogue transaction. At the SFD lev-
el, transactions have no specific information about their dialogue content, form, or control. The next section contains a scenario in which use of this procedure is explained in detail, including the figures which are referred to in these steps.

Step 1. Define the global transaction-wide attributes for the transaction, using the form shown in Figure 14.

Step 2. Decompose the transaction into its interactions, with each interaction represented by a double circle. At this level, the representation is still a graphical, SFD-like notation, with appropriate modifications and enhancements for dialogue-specific needs, as explained in the previous section. If there is a set of several commands in the transaction, the first interaction is the root interaction, with predicates that identify the commands, indicating possible choices out of this interaction, as shown in Figure 15. The root interaction contains all possible input values that are valid for the first interaction of the transaction; it is, conceptually, all possible command names in the interaction language of this transaction. If
the transaction is a simple response to a request, it will not have a set of command names in its root interaction, but will have only a single sequence of one or more interactions (e.g., only one interaction such as responding 'Y' or 'N', or only one response such as entering name and id). It is simply decomposed into its interaction(s) as shown in Figure 16.

Step 3. Define the interaction-wide attributes for each interaction, using the form shown in Figure 17.

Step 4. Decompose each interaction into its prompt, input, and confirmation parts. (If there is a root interaction, begin with it.) The graphical representation is no longer used; each part is defined by filling in a separate form which elicits information needed to specify each part. This includes the dimensions, interaction attributes, input definitions, delimiters, syntactic constraints on tokens, visual features, etc. These forms are shown in Figures 18, 19, and 20; there is one form for each possible instance of an interaction part.
7.4.4 Scenario Showing Use of the Methodology

The sample interface of Chapter 5.2.4 will be used here to show use of the methodology. Several other commands will be added simply to make the interface richer and therefore more illustrative. This group of commands would not necessarily be cognitively related in a real world transaction; they have been chosen to represent a variety of occurrences in interfaces. Assume that the transaction to be designed is "get valid command". Figure 15 shows the commands in this transaction. This is actually Step 2 in the dialogue engineering methodology procedure described above, decomposing the transaction into its interactions. Notice the use of the root interaction ("get valid root"), the labels on the paths coming out of the root interaction indicating all possible commands, conditional decision (in the "define" command), ellipses (in the "copy" command), order independence (in the "reserve" command), and semantic checking (in the "copy" command).

A brief description of each command is given below:

COPY - copies contents of existing-file into new-file. The ellipses specify that

\[
\text{copy } \text{x}yz \text{ to } abc
\]

is a valid input, where "to" is a noise word. Likewise,

\[
\text{copy } \text{x}yz \text{ abc}
\]

is also valid.
PAS or FOR or BAS or CC - invokes the appropriate compiler (Pascal, Fortran, etc.) for the specified file; compile options include DEBUG, NOLINK, etc.

number of lines - an integer which indicates the number of lines to move in the current file.

DEFINE - defines color or video mode for a display.

CHANGE - change string1 to string2 in the current line of a file, with an option to change all occurrences in the file indicated by a '*'.

RESERVE - reserve a seat on a flight from departure-city to arrival-city on the given date. The order independence specification indicates that either

reserve (roanoke, paris, 01/10/84)

or

reserve (01/10/84, roanoke, paris)

is a valid input string. Note that departure-city and arrival-city are not order independent; departure-city must be entered before arrival-city.

Following is a detailed explanation of each step of the procedure for defining the transaction containing these commands.

Step 1. Define transaction-wide attributes. These include choice of confirmation point, some visual and processing attributes, and a list of the commands in the transaction. (Note: The scope of attributes is local to where they are defined; i.e., transaction attributes are in effect throughout a transaction unless superceded by interaction attributes for a specific inter-
action in Step 3. These interaction attributes are no longer in effect once the interaction is finished.) A completed form for the example transaction is shown in Figure 14.

Step 2. Decompose the transaction into its interactions. A simple transaction, with only one interaction or only one command, is represented as shown in Figure 16. For a set of commands, as in the sample transaction, "get root interaction" is the first interaction, followed by a branch to each command, broken down into its interactions, shown in Figure 15. Sequencing from interaction to interaction within each command is indicated by the solid control flow lines.

Step 3. Define interaction-wide attributes, in the same manner as attributes were defined in Step 1. A completed form for the interaction "get valid root" is shown in Figure 17.

Step 4. Decompose each interaction into its prompt, input, and confirmation parts. Cognitively, it is easiest to begin with the root interaction, if there is one, since this interaction contains all possible command roots. It is also easiest to cycle through the prompt, input, and confirmation parts of a form in that order.
Confirmation point:  ✔ action (default)
                      _ interaction
                      _ transaction

Visual attributes:  Blink all text     _ Y _ N
                    Blink all graphics _ Y _ N
                    Reverse video all text _ Y _ N
                    Reverse video all graphics _ Y _ N
                    __ Color of all text
                    __ Color of all graphics
                    _ BLUE_ Background color

Processing attributes:  Token completion _ Y _ N
                        Echo _ Y _ N
                        Error beep _ Y _ N
                        Type of error check:  ✔ Spelling correction
                                              _ Echo erroneous input
                                              _ Do not echo erroneous input

Type of transaction:  ✔ Set of commands   _ Request/response

If the transaction is a set of commands, list them below:

COPY
PAS
FOR
CC
BAS
AREA
DEFINE
CHANGE
RESERVE

Figure 14: Form for Defining Transaction-Wide Attributes
Figure 15: Decomposing the Example Transaction Into Its Interactions
Figure 16: Decomposing a Simple Transaction Into Its Interaction(s)
Figure 17: Form for Defining Interaction-Wide Attributes
Step 4a. The "prompt" form contains blanks to vary the dimensions, as well as a space for sketching a screen layout. It also elicits some device and visual attribute information. The completed form for the "get valid root" interaction is shown in Figure 18.

Step 4b. The "input" form is the most complicated. Each possible valid input value (the lexical token values) must be specified ("input specification"), either as an exact representation or as a rule. An exact representation is entered exactly as it appears in the specification (e.g., "copy"); a rule is a description of a value to be input (e.g., integer between 0 and 50, or alphanumeric string less than 32 characters). A special language for the specification of rules can be developed if the textual descriptions prove to be too cumbersome. For each input specification, its input form, its (syntactic) token value to be returned to the computational component, its delimiter (which can also be an exact representation or a rule), and its abbre-
Figure 18: Form for Defining a Prompt
viations must also be given. Because processing attributes such as token completion, echoing, and error checking can vary with the dimensions, these are defined here rather than on the interaction-wide attribute form. Some context-sensitive information is elicited, if needed, by specifying that the values for this interaction must be operationally related (e.g., <, >, =, etc.) to the value of another interaction. "User must input _ to _ value(s)" allows specification of the number of token values that can be input for each interaction instance. If 0 values may be input, the interaction is optional; if at least one value is required it must have a default value. A completed form for definition of the input for the "get valid root" interaction is shown in Figure 19.

Step 4c. The "confirmation" form allows connection of error types to error messages (for negative confirmation), positive and neutral confirmation messages, and help. All but the negative confirmation are straightforward, similar to the
Figure 19: Form for Defining an Input
"prompt" form with a sketch of the screen, the dimensions, and some visual attributes. For negative confirmations, the possible error types that can occur for each interaction input must be identified. Each error type becomes a value for dimension \( t \) (type of error) and appropriate error messages created for each. A completed negative confirmation form for an error message for an input of greater than 9999 is shown in Figure 20. This error could occur in the root interaction if the user tries to enter a value greater than specified (i.e., the number of lines must be between 0 and 9999).

The decomposition of this small example transaction will easily fit on one page. However, for a large transaction such as would be expected in a real system, the commands will need to be spread across several pages. In fact, it is convenient and greatly contributes to understanding to put each command on a separate page. This can be done using the following modifications to the procedure, while maintaining the integrity of the methodology:

Step 2a. Decompose the transaction into two interactions: "get valid root" and "get valid root parameters", as shown in Figure 21.
System Name Sample
Transaction Name but valid command
Interaction Name but valid root
Is this the root interaction? Y N

NEGATIVE CONFIRMATION (Q, 7**99, 1, (C2'>, «¤•=«..) t p d u

Give a rough sketch of the screen format and contents below:

Clear screen before putting up this confirmation? Y N
Leave this confirmation on the screen for _ seconds. (If this is not specified, the confirmation will remain on the screen until the next interaction.)

Visual attributes:
- Blink all text Y N
- Blink all graphics Y N
- Reverse video all text Y N
- Reverse video all graphics Y N
- Color of all text R Y G
- Color of all graphics Y N
- Background color

Figure 20: Form for Defining a Confirmation
Figure 21: Decomposing a Large Transaction
Step 2b. Decompose interaction "get valid root parameters" into all its separate commands, each with its own interactions, also shown in Figure 21. (Assume each command is on a separate page.)

7.5 SPECIFICATION OF INTERNAL DIALOGUE

As already discussed, internal dialogue is the flow of data between the dialogue and computational components of a system. It is formally specified and represents the communication interface between the dialogue author and the application programmer. The flow of data from computation to dialogue is related to dialogue output transactions and is addressed in [SIOCA84]. The flow of data from dialogue to computation, however, appears on the forms of the methodology. Specifically, the application programmer must know the token values that will be sent from the dialogue component to the computational component. This information is the "token value" shown on the input definition form of Figure 19. The application programmer must also know whether those token values have been semantically checked, and, if so, the nature of that semantic check (e.g., existence in a database). This is indicated on the interaction-wide attribute form of Figure 17.
7.6 FUTURE DIALOGUE ENGINEERING METHODOLOGY RESEARCH

The dialogue engineering methodology is the newest part of this research and, as such, has not been used extensively. It has been tried on numerous sample interfaces, but not on an interface that is to be implemented using AIDE. Co-ordinating it with a new version of AIDE, so that the two are well-integrated and cohesive is a rich future research area. Continuing to exercise the dialogue engineering methodology, to determine its completeness and efficacy, incorporating appropriate extensions, is a major portion of the future research.
Chapter VIII

THE AUTHOR'S INTERACTIVE DIALOGUE ENVIRONMENT (AIDE)

"I sit beside my lonely fire
And pray for wisdom yet:
For calmness to remember
Or courage to forget."
Charles Hamilton Aide, Remember or Forget

8.1 MOTIVATION FOR AIDE

As already discussed, the dialogue author, while creating the dialogue component of an interactive system, does not program the dialogue transactions in the way that an application programmer programs the computation components. The motivation for this approach was presented in Chapter 2.2. Instead, the dialogue author uses a special environment, called the Author's Interactive Dialogue Environment (AIDE) which is a set of interactive high-level tools to facilitate dialogue design, implementation, testing, and evaluation without writing source programs [JOHND82]. The dialogue author, using AIDE, directly manipulates objects on the screen, based on a "what you see is what you get" (wysiwyg) principle.

Most interfaces are composed of objects and communication forms that belong to identifiable classes of interface com-
ponents. Rather than repeatedly coding that same type of interface component, AIDE, in DMS, provides an automated tool for developing as many of these classes as possible. That is, the DMS team programs tools which can be used repeatedly to develop specific types of interfaces, rather than programming the interfaces themselves. By managing dialogue development as an activity separate from computational program development and by providing the means for rapid modification of dialogue, AIDE facilitates the production of highly human-factorable interfaces.

8.2 AIDE ARCHITECTURE

The architecture of AIDE contains several levels, as shown in Figure 22. At the top are the author workstation and the AIDE interface. These form the author level, which (in the first version) is primarily a set of keypad displays and the software to control their interaction with the dialogue author. The author level represents the operations used to create and modify dialogue transactions.

The next level, the tool level, is a set of functional tools that allows development of prompts, language inputs, and confirmation messages. Under this is the representation level which contains a transaction database (TDB). The TDB for AIDE Version 1 is a host-programmed interface using
Figure 22: AIDE Architecture
DEC's Datatrieve. In future versions, it will be a fast relational database which holds tuples representing the contents of, and the attribute values for, each transaction/interaction and its objects. They will be retrievable by any of their attributes, including textual content, as needed for modification using AIDE.

Under the representational level is the **executor level**. The executors interpret the definitions currently in the TDB into an image on the dialogue screen for author feedback during the design of transactions. These same executors also interpret the definitions at run-time. Displays and user input are accomplished using DMS device driver services, providing device independence down to the lowest, **device level**.

### 8.3 HOW AIDE RELATES TO THE TRANSACTION MODEL

Like the dialogue engineering methodology, the use of AIDE to develop dialogue transactions is guided by its organization around the dialogue elements of the multi-dimensional dialogue transaction model. The tools of AIDE are dictated by the elements in the model; navigation (the control structure) within the AIDE interface is guided by the relationships among those elements. Figure 12 showed the hierarchy of dialogue elements and their relationships.
In AIDE Version 1, each dialogue element has a keypad from which the author chooses the appropriate functions to develop that element. The control structure among the keypads provides both vertical and horizontal movement among elements. For example, the keypad for developing the "PROMPT" part of an interaction has the following functions:

- EXIT AIDE
- DELETE CURRENT PROMPT
- DEVELOP MENU
- DEVELOP KEYPAD
- DEVELOP TEXT
- DEVELOP GRAPHICS
- DEVELOP FORMS
- DEVELOP ANOTHER INTERACTION PART
- RETURN AND DEVELOP ANOTHER INTERACTION
- HELP

At this point, the author may choose to go vertically down the hierarchy by pressing (for example) "DEVELOP MENU", to go vertically up by pressing "RETURN AND DEVELOP ANOTHER INTERACTION", or to move horizontally in the same level by pressing "DEVELOP ANOTHER INTERACTION PART".

Analogous keypads exist for each dialogue element. Each of the common functions (e.g., DELETE, EXIT AIDE, RETURN, HELP) is always located in the same position on each keypad. All keypads also allow one-step EXIT from AIDE, without having to "back out" a level at a time.

Dialogue elements can be developed in any order. AIDE is designed to allow the dialogue author to move freely among transactions, interactions, parts, and pieces. For example,
if a menu is being designed, as each menu choice is added to the prompt, its (relatively simple) language input definition can be given and the corresponding confirmation messages can be composed, if desired. Alternatively, the entire menu prompt can be designed and then the inputs and confirmations, in turn. Similarly, it is easy to move from one formatter to another to combine, for example, a keypad, some text, and some graphics within the same prompt definition.

Transactions must be given unique names when they are created. The transaction//interaction name is the identifier by which specific interactions and their parts are later recalled from the TDB.

8.4 AIDE VERSION 1 OVERVIEW

While the general goal of AIDE is to provide an interactive tool for use by a dialogue author in developing human-computer interfaces, AIDE Version 1 had a very specific goal. Its purpose was to prove that the theory and concepts of DMS -- dialogue independence, dialogue author, and an interactive tool for the dialogue author -- could be implemented. Indeed, this first version is, of necessity, limited in the types of interfaces it can produce. Nonetheless, it has been used to produce a demonstration application system having two different interfaces (one is keypad-driven,
the other is menu-driven) with identical functionality. At execution-time, both of these interfaces run on a single computational component, thus demonstrating that dialogue independence is a viable concept. This goal, coupled with AIDE's carefully considered design decisions, has led to a dialogue author interface that is functionally integrated, flexible and consistent, yet with a minimum of modality. The preliminary evaluation of AIDE, discussed in Chapter 9, has shown promising results, indicating that a very diverse group of tools can be integrated into a single, cohesive, usable interactive system for developing interfaces.

8.4.1 AIDE Interface

The structural organization of AIDE Version 1 is shown in Figure 23. The dialogue author's interface provides the human-computer dialogue between the author and the functional tools. The dialogue author's workstation consists of a command screen (a VT100) with a touch-sensitive panel, a color graphics dialogue screen (a GIGI) with a tablet and cursor, and a standard keyboard with auxiliary keypad. The command screen is windowed for a keypad outline for command selection, a user prompt area, a help area, an error message area, and a user input area. The dialogue screen presents an image of the interaction part begin constructed, exactly as
Figure 23: Author's Interactive Dialogue Environment
it will be seen by the end-user at application system execution-time. AIDE uses an auxiliary keypad (programmable function keys on the VT100) as its primary means of function selection. The currently active functions of a keypad are displayed in a labeled keypad outline on the command screen so that the dialogue author always knows the current meaning for each key. The display changes as appropriate whenever a different level of AIDE is entered, causing a change in the currently active AIDE functions. A key can be selected either by directly pressing it on the auxiliary keypad or by touching its image on the touch-sensitive screen.

8.4.2 Tools of AIDE

The first version of AIDE incorporates tools for constructing prompts consisting of menus, keypads, forms, text, and/or graphics (in any combination); inputs for menus and keypads, based on a concept called Language-By-Example (LBE), discussed in detail in Chapter 8.5; and confirmations consisting of text. Tools which are being considered for future versions of AIDE include a touch panel formatter, a window formatter, a dialogue design "expert", and a significant extension to LBE to include more complicated input definitions such as command strings and combinations of input forms within a single interaction.
Menu and Keypad Formatters

Certain combinations of text and graphics, such as menus and labeled keypad outlines, that can be standardized and for which reasonable human factors principles exist, can be created using AIDE Version 1. The formats of these types of prompts are hard-wired as templates that the dialogue author fills in using the appropriate formatting tool. The menu template includes fields for the title and purpose of the menu, the menu options and their selection codes, and the query that the end-user should answer by the choice of a selection code. For keypad templates, textual icons can be used as labels to indicate function choices to the end-user. Attributes for all fields are also modifiable in real time by toggling through the possible choices. In each of these formatters, the author at design-time is constrained to cursor movement only within the predefined fields. One of the major principles of human-engineered display design is consistent formatting throughout an entire application system. Use of such formatters encourages this consistency while simplifying the dialogue author's job.

Forms Formatter

A forms, or "fill-in-the-blanks" format provides an effective type of interface, especially for data entry. A forms formatter in AIDE allows the author to label and de-
fine blank fields on a form on the screen. Functionally similar to IBM's Display Management System [IBMDMS1], this formatter, like all other tools of AIDE, does not require programming by the dialogue author.

**Text Formatter**

This tool allows the author to develop and edit text in an arbitrary screen window. This is useful for formatting textual pieces of prompts, as well as confirmations. AIDE also makes direct use of the text formatter for text within the menu, keypad, and form formatters. In these formatters, AIDE automatically sets the window size and position to match the appropriate predefined template fields.

**Graphical Formatter**

A graphical formatter provides a set of graphical editing functions for the construction of simple shapes (lines, circles, arcs, boxes, polygons) as well as manipulation and modification of their attributes (e.g., size, color, and screen position). It also allows the development of composite objects from these simple shapes. The graphical formatter, consistent with the rest of the AIDE interface, uses keypads for function selection. However, through informal testing, a tablet with cursor was found to be more natural than incremental-movement "arrow" keys for cursor positioning in this formatter.
Language-By-Example

As previously discussed, all human input to human-computer interfaces, in DMS, is viewed as expressions in an interaction language. The language input definition tool of AIDE assists the author in the design, specification, and implementation of interaction languages. Because it is more complex than the other AIDE tools, it is discussed in the next section in some detail.

8.5 LANGUAGE-BY-EXAMPLE

8.5.1 Motivation and Philosophy of LBE

Language-By-Example (LBE) has been developed in DMS and AIDE as an approach to specifying the definitions for the language input parts of a transaction. As a tool of AIDE, LBE is an alternative implementation of the forms-based approach. It is a powerful method for defining command string inputs, yet is applicable also to simple input forms such as keypads and menus; AIDE currently contains tools, based on the LBE concept, for defining keypad and menu inputs. 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 (AIDE) queries and dialogue author respons-
es, the dialogue author is guided through the definition of an interaction language. The definition of each input includes specifications of such details as how to receive the input (device(s), position, and channel-related details), and on the presentation modes (e.g., token completion, spelling correction). By starting with a specific example and working toward a general definition, LBE follows the human cognitive problem-solving process. A "stand-alone" (i.e., not yet integrated into AIDE) prototype for command strings is partially implemented.

In LBE, and throughout DMS in general, the semantics of interaction languages are kept separate from their lexical and syntactic considerations. This is, however, for semantic validation only, and not for processing a semantic action.) This separation is dictated by the principle of dialogue independence. The lexical and syntactic composition of interaction languages is of no concern to the application programmer; it is decided upon by the dialogue author independently of the computational design. In fact, as a result of human factors testing and iterative refinement, the lexical and syntactic details for a given computational function can be subject to considerable change as interface design evolves.
As previously discussed, under DMS, token values passed from the dialogue to the computational component at run-time have been at least lexically and syntactically validated. Often the tokens have been semantically checked as well. One goal of AIDE is that the application programmer does not have to implement input validation. It is the definition for a lexically and syntactically correct input that LBE elicits from the dialogue author.

8.5.2 A Brief Example of LBE

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., "copy existing-filename new-filename"). Then, by moving the cursor, the dialogue author delineates each entity (i.e., token or delimiter) of which that command string is comprised. In this example, there are five entities to be defined: three tokens ("copy", "existing-filename", and "new-filename") and two delimiters (the blanks following both "existing-filename" and "new-filename"). When an entity is delineated (confirmed by reverse video highlighting of the current entity), the dialogue author responds to system queries about that entity, 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. The dialogue author then delineates the next entity and gives appropriate responses about it. The dialogue author responds to such questions as whether the entity is an exact representation or a rule, whether it is required or optional, etc. 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.

8.6 INTERFACES AT RUN-TIME: DYLEX

While the run-time processor for dialogue transactions is not specifically in the scope of this dissertation research, a brief discussion of it is appropriate here, since the issues which it addresses are the same ones that are addressed by the design-time research. The heart of this run-time dialogue transaction processor is a DYnamic Language EXecutor (DYLEX) which has the capability of processing the interfaces which AIDE produces. It determines the input devices and
parses the end-user inputs of exact representations, rules, and delimiters. It provides algorithms for such features as token completion and spelling checking. Transaction and interaction attributes such as various types of input error checking, scanning (e.g., for processing ellipses), echoing, positioning of the cursor, defaults, and visual characteristics are also processed.

Constraints on token values are processed as inclusion relations and exclusion relations. An inclusion relation is specified on a token (interaction) value when that value must be the same as the value for a different token. Conversely, an exclusion relation is specified on a token (interaction) value when that value cannot be the same as the value for a different token. It does the mapping from the lexical token value (input by the end-user) to the syntactic token value (returned to the computational component). The execution of DYLEX is data-driven, based on the control structure of the dialogue transaction model; this control is hard-wired and invariant within the dialogue component of an application system.
8.7 FUTURE RESEARCH ON AIDE

AIDE Version 1 has proven to be an excellent medium for testing ideas and for gaining understanding of the complexity of such interactive tools for dialogue design. The goal of AIDE Version 2 will be to produce a system which can be taken to an industrial test-bed for use in a real world situation. The next version of AIDE will be significantly different from AIDE Version 1. There are four main features that will, at least to the dialogue author, be different: the hardware, the features, the tools, and the navigational approach.

8.7.1 Hardware

The hardware will be a single large-screen high-resolution bit-mapped graphics station. The interface will probably be window-oriented, possibly with "window shades". The hardware of AIDE Version 1 has been unsatisfactory with respect to its resolution and flexibility. Modern equipment should greatly enhance the AIDE interface.

8.7.2 Features

The functionality of AIDE Version 2 will be greatly extended to include features which the first version does not have. The approach to using AIDE will be oriented toward
the dialogue engineering methodology as a guide for dialogue
design and implementation. AIDE Version 2 will incorporate
the characteristics of the multi-dimensional dialogue trans-
action model, including the five dimensions and confirma-
tion points. It will have a powerful high-speed relational
database underlying it, so that performance will be in-
creased and so that the dialogue author can retrieve dia-
ologue transactions by name, by form, by content, or by visu-
al characteristics. Design-time templates (e.g., for
elements such as menus and keypad outlines) will be provid-
ed, so that the dialogue author can insure a consistent for-
mat; these templates will also be modifiable to allow tai-
loring to specific situations. Design of dialogue output
transactions (dynamic transactions) will be incorporated.

AIDE Version 2 will allow the dialogue author to specify
run-time metering of an interface at several levels (i.e.,
lexical/action, syntactic/interaction, and semantic/transac-
tion) so that the interfaces that AIDE produces can be eval-
uated. System development "smarts" for accounting purposes
will allow such functions as monitoring completed work, en-
forcing consistency among transactions, and detecting errors
and problem areas. A help/training facility will be includ-
ed in AIDE, to give the dialogue author information on what
AIDE can do. A mechanism for the end-user to use to design
and/or modify an interface at run-time will also be needed. AIDE will be integrated with GPL, so that the DMS design-time facility for both dialogue and computational components is a reality. While all these features may not be in AIDE Version 2, they will be considered and will undoubtedly appear in some future version.

8.7.3 Tools

Future versions of AIDE will also have several tools which AIDE Version 1 does not. These include a mouse formatter, to design interfaces with multi-button mouse inputs; a window formatter, to design multi-window screens and "window shades"; a touch panel formatter, to design interfaces which have touch-sensitive screens; a voice formatter, to allow interfaces with voice input/output; and a dialogue design "expert", to provide the dialogue author with human factors guidelines and principles when designing an interface.

8.7.4 Navigation

AIDE Version 1 allowed the dialogue author complete flexibility in the order in which parts of a transaction/interaction were developed. The enhanced features of future versions of AIDE will impose some restrictions on this
flexibility; some precedences will be necessary. Designing
using the dimensions is an example. In order to design
meaningful negative confirmation messages, the dialogue au-
thor must first have defined the input part of an interac-
tion. Otherwise, the possible types of end-user input er-
rors will not be known, and informative negative
confirmation (error) messages are not possible. Other such
precedences are also implied by various new features; they
will be incorporated into the next version of AIDE. The
loss of some of the flexibility of AIDE Version 1 is offset
by the structure and control which the imposed precedences
will provide. These constraints will allow the dialogue au-
thor fewer options (too many of which can be confusing) and
more guidance. The interfaces which can be developed will
also be much more complex and flexible.
Chapter IX
EVALUATION OF THE RESEARCH

"To measure you by your smallest deed is to reckon the power of ocean by the frailty of its foam. To judge you by your failure is to cast blame upon the seasons for their inconsistency." Kahlil Gibran, *The Prophet*

9.1 ISSUES IN SUCH AN EVALUATION

Proposing a model, a methodology, and tools for human-computer interaction is only a portion of the total task of formulating a complete framework in which to develop human-computer interfaces. In addition, the model, the methodology, and the tools must be evaluated for their efficacy and usability. However, there are several significant problems with making such an evaluation. The first task is to decide how such evaluations should be done. The evaluation of human-computer interfaces during the past few years has evolved away from formal empirical human factors studies toward observation and iterative refinement. "Point testing" of discrete parts of an interface is becoming less prevalent; most interface testing techniques are now attempting to evaluate the interface holistically. Numerous companies have recently adopted this more informal approach to test-
ing, despite relatively unlimited resources and with much at stake for their commercially available systems [BEWLW83].

Thus, two types of testing are evolving in the field of human-computer interface evaluation: formative and summative [DICKW78, WILLR83]. Formative evaluation is the collection of data during product development to make the final product as effective and efficient as possible. Summative evaluation is a more formal testing to determine whether the final product performs as desired. The evaluation described in this chapter is primarily summative.

Issues that need evaluation in DMS include such questions as whether the multi-dimensional dialogue transaction model is adequate for describing a variety of interfaces, and therefore, whether AIDE is adequate for implementing a wide variety of interfaces. Other questions include inquiry into whether AIDE can be used by non-computer professionals, and the extent to which AIDE can be expanded so that it can still be used to develop a variety of interfaces without writing programs. These are non-parametric questions and, as such, do not lend themselves to conventional controlled experimentation. They are too wide-reaching to be answered by point tests, and formulating all possible hypotheses to test such questions would take years.
Another significant problem is how to separate the model being evaluated from the tools through which that model is instantiated. Because of this, the empirical study presented in this chapter should be considered to describe a summative evaluation of an overall approach, primarily the model and the direct manipulation tools.

A final, pragmatic consideration in this evaluation is the current status of AIDE. AIDE Version 1 was based on early stages of the dialogue transaction model research. In particular, it allows development of only the syntactic level (prompt, input, confirmation) without any dimensions. A new version of AIDE, incorporating more of the model, is beyond the scope of this dissertation research. However, the syntactic level without dimensions which is reflected in AIDE Version 1 is the heart of the multi-dimension dialogue transaction model which has emerged, and, as such, is worth evaluating.

9.2 AN EMPIRICAL EVALUATION OF AIDE VERSUS PROGRAMMING

The purpose of the testing of AIDE is not a rigorous empirical evaluation. Rather, this evaluation is a preliminary study to begin determining the usefulness of the dialogue transaction model and of AIDE for developing human-computer interfaces.
9.2.1 Experimental Hypothesis

The hypothesis of this research is that creation and modification of human-computer interfaces is faster and easier using AIDE than using a conventional programming language. The independent variable is the mechanism used for developing the interface task (i.e., AIDE or a programming language); the dependent variable of primary concern is the length of time it took subjects to complete the task.

9.2.2 Methods

Subjects

People who had been working on the DMS project were asked to be subjects. Three people were dialogue authors and three others were application programmers. The dialogue author subjects were people who had implemented various parts of AIDE and were generally very familiar with its use. The application programmer subjects were people who had implemented other parts of DMS (e.g., the graphical programming language and the database), had used the DMS special services for input/output screen handling, and had executed programs under the DMS multi-process execution environment. The extremely small sample size is due primarily to the fact that the people who were chosen as subjects were the only possible ones who could be termed "expert" AIDE users and
"expert" DMS-environment programmers. It was desired that all subjects initially be "experts" at their task so that the issue of training on AIDE or DMS services/multi-processing was not a confounding variable. All subjects were familiar with the overall DMS/AIDE philosophy.

*Equipment Setup*

The experiment was conducted in one of the testing rooms in the Human Factors Lab in the basement of Whittemore Hall. The dialogue authors used an AIDE workstation (GIGI color monitor dialogue screen, VT100 with touch-panel command screen, standard keyboard, tablet with cursor). The application programmers used only the GIGI monitor and keyboard. A video camera and microphone were connected to a video recorder, headphones, and black and white TV monitor. The TV monitor and headphones were in an adjoining room so the experimenter could observe and hear all subjects. Dialogue authors were taped as they performed the task; application programmers were not. The reason for this was to capture information on how the dialogue authors approached the task using AIDE and what navigational procedures were used, to be later analyzed for feedback on improving AIDE and the model (formative evaluation). Recording the programmers was not considered necessary. A special account was set up on VAX1 for subjects to use during the experiment.
Experimental Design

The experiment was a two-level between-subjects design with one independent variable. The two levels were use of AIDE and use of a programming language. The dialogue author subjects were asked to use AIDE to create a keypad-driven interface and then to modify that interface to be menu-driven. The application programmer subjects were asked to create and then modify that same interface, using their choice of either Fortran or C. No particular randomization or counterbalancing techniques were needed for this study.

Before they were given the task description, all subjects were asked to sign an "informed consent" release and to answer some demographic questions concerning their computer science education, their previous use of DMS, AIDE, and the Fortran and C programming languages.

The interface design both for the creation task and the modification task was completely defined for both groups of subjects. All six subjects were given the same basic written task description. Special notes were added as needed for each subject group (e.g., dialogue authors were given specific transaction and interaction names to use). Subject programmers were given written documentation on DMS services, how to execute programs under DMS, and VAX Fortran and C. Subject authors were not given written AIDE documentation.
All subjects were given the same procedural instructions. They were told the task had two parts: creation and modification; they were given first the creation task, and then, when it was completed, the modification task. They were told to read the task and ask questions before they began; timing did not commence until a subject started the task (subjects did not know they were being timed). Checking the correctness of the interface was explained; the experimenter inspected and tested the interface when the subject believed the task was finished. Subjects were told they were being observed and that they could ask questions of the experimenter at any time during the experiment. Subjects were also asked to talk aloud as they worked, and to make notes on any problems they had. All subjects were given pencil and paper.

Task Description

Creation of the Transactions

Creation of a transaction involved three subtasks featuring, respectively, creation of the computer prompt, the human input, and the computer response parts.

Prompt Display: The transaction began with the GIGI display shown in Figure 24. The small squares and circles represented aircraft in the GENIE environment. Aircraft were each labeled with a three-digit identifier. The keypad
keys were labeled with the same three-digit aircraft identifiers. Colors for all parts of the display were explicitly shown in Figure 24.

*Human Input:* The system was to accept only labeled keypad keys. A key press was to be an immediate command, with no carriage return after it. The system was to return, to the computational program, a token value which was the character string labeling the corresponding key (i.e., the aircraft identifier itself). The system was to respond to all inputs, other than labeled keys, with a 'beep', and wait for another input.

*Computer Response:* The computer response to the pushing of a keypad key labeled with an aircraft identifier was to make the corresponding aircraft symbol (circle or square) blink. It was not required to erase the screen and re-display it, but was acceptable to do so. In terms of the current version of AIDE, this response was a new transaction, with display only (i.e., no input or confirmation parts).

*Modification of the Transactions*

The description of this part of the task was not given to the subjects until they had completed the creation task.

*Prompt Display:* The display for the modified version of the task was the same as that for the original task, except that the keypad was to be replaced with a menu for aircraft
Figure 24: Display with Keypad Prompt
selection, as shown in Figure 25. The colors and attributes (shading) of the circles and squares were to match the corresponding description given in the menu itself.

**Human Input:** The system was to accept only aircraft identifiers in the menu. Input was to be terminated with a carriage return (i.e., not immediate). The system was to return, to the computational program, a token value which was the character string for the selected aircraft identifier (same as for the creation task). However, the system was to reject any other input as soon as it could be determined to be erroneous. This required *character-at-a-time input validation*. For example, if the user typed "13", the "1" was to be accepted, but the "3" (not being a character which can validly follow the "1") was to be rejected with a 'beep'.

**Computer Response:** The computer response to the menu selection of a valid aircraft identifier was to make the corresponding aircraft symbol turn red. Again, it was not required to erase the screen and re-display it, but was acceptable to do so. In terms of the current version of AIDE, this response was, again, a new display-only transaction.
The following are valid aircraft identifiers:

- 118 Green Unshaded Square
- 121 Cyan Unshaded Circle
- 317 Yellow Shaded Square
- 715 Magenta Shaded Circle

Please enter an aircraft identifier, followed by carriage return.

Figure 25: Display with Menu Prompt
9.2.3 Results

The mean times, in minutes, for each group of subjects to perform the creation, modification, and total tasks are shown in Table 1.

The difference between the two groups was found to be significant for the creation task, $t(4) = 11.9, p < .005$; for the modification task, $t(4) = 5.2, p < .005$; and therefore for the total task, $t(4) = 9.9, p < .005$.

The small demographic survey completed by each subject before beginning the tasks showed that all had more than ten computer science courses. The programmer subjects all considered themselves to be "expert" in Fortran or C, and to have a "near expert" knowledge of DMS services and running programs under the DMS multi-process execution environment. Author subjects rated themselves as "expert", "near expert", and "moderately knowledgeable" in the use of AIDE.

In addition to the quantitative results given above, subjective results were acquired by observing subjects as they performed the tasks and by asking them several questions upon final completion of the tasks. Among the programmer subjects, the experimenter observed numerous signs of frustration (e.g., tapping fingers, impatient mumbling, "argh!", "this is a pain!") and tiredness (e.g., asking for a break, "this is exhausting"). When asked, at the end, how it had
<table>
<thead>
<tr>
<th>Task Type</th>
<th>Dialogue Authors</th>
<th>Application Programmers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation Task</td>
<td>43</td>
<td>168</td>
</tr>
<tr>
<td>Modification Task</td>
<td>29</td>
<td>63 *</td>
</tr>
<tr>
<td>TOTAL</td>
<td>72</td>
<td>231</td>
</tr>
</tbody>
</table>

* One application programmer subject became too tired to complete the modification task; that subject's time until quitting the modification task was used in the calculations, since a time to completion for the task was unavailable.
been, all three immediately responded "Tiring!". The author subjects, on the other hand, responded "That's all?", and "Fine", and showed no signs of frustration or exhaustion. None of the subjects thought the task was too difficult. All three programmer subjects said they had the most trouble with positioning objects on the screen and doing the character-at-a-time validation for the menu inputs. Author subjects named no real problems other than response time of the underlying database. The author subjects were asked how long they thought it would have taken them to code the same task; they estimated anywhere from 4 to 8 hours and all expressed great relief that they did not have to do it.

9.2.4 Interpretation of Results

These results support the hypothesis that creation and modification of an interface is faster and easier using AIDE than using a programming language, at least for those types of interfaces AIDE was designed to develop. This is clearly demonstrated in the mean time to complete the tasks by each group of subjects. The dialogue author subjects performed the creation task 3.9 times faster than did the application programmer subjects, the modification task 2.2 times faster, and the total task 3.2 times faster. The subjective observations made by the experimenter during the tasks and questioning the subjects after task completion support the hypothesis as well.
There are several limitations in this experimental design that must be discussed here. The number of subjects was very small; this was because of the very limited population from which to choose subjects that were experts at either AIDE or DMS-style programming. It is also recognized that the task was clearly developed so that it could be accomplished using AIDE. AIDE Version 1 is limited in its functionality, and can produce only interfaces comprised of menus, keypads, text, and graphics. However, defining a task that could not be accomplished using AIDE would have not shown anything other than AIDE's inadequacies.

Despite the superior performance of the author subjects, if anything, the task favored the programmer subjects. Because AIDE does not yet have capabilities for dynamic displays, the author subjects had to create a new transaction with four separate interactions for the second display of each task, one interaction for each of the four possible aircraft. Each of these four interactions was identical except for the requested change to the aircraft (i.e., blinking for the creation task and turning red for the modification task). Also, because the programmer subjects used the DMS input/output services, they were removed from many of the problems of complicated screen handling. Had they not been able to use these services, they would have had to do
screen handling through the VAX system services. Interestingly, however, as noted above, all programmer subjects claimed to have the most trouble with positioning objects on the screen. This is, of course, one of AIDE's strong points, since AIDE embodies a "wysiwyg", direct manipulation approach.

The one application programmer subject who did not complete the modification task obviously caused a problem with the data. Because of the small sample size, rather than discarding that subject's data completely, it was decided to use the elapsed time until the subject abandoned the modification task. The effect of this on the final results would, of course, favor the application programmer group; had this subject completed the task, the mean time to perform the task would have been even higher for the programmer group.

One remaining question which must be asked is how these results reflect on the efficacy of the dialogue transaction model upon which the AIDE interface is based. Presumably a bad model would not have a good instantiation. Thus, since the design of AIDE was directed by the model and since this study indicates that AIDE performs well, it is can be inferred that the model is a reasonable representation of human-computer interaction. Just as it is difficult to separate a model from its instantiation, it is also difficult, espe-
cially in this study, to isolate other significant factors that might affect results of the study. An example would be segregating the effect of the model from the obvious advantages of direct manipulation that are incorporated into the AIDE interface. Thus, it is evident that this study is an evaluation of an approach to interface design, especially using the model and the tools of AIDE, and not a clear-cut point test.

Finally, this study was not a comparative evaluation of this model versus other models; rather it was simply an attempt to determine whether this model has led to a workable, useful system. The results of this study definitely indicate that interactive tools for interface development are worth more research.

9.3 A SUBJECTIVE EVALUATION OF THE MODEL

Because of the limitations involved in evaluating a model separately from its instantiation, as discussed above, a more subjective evaluation of the model itself is appropriate. This involves evaluating how well it serves its purpose of describing human-computer interfaces. This translates into at least the following criteria: applicability of the model, scope of applicability, precision, and uniqueness.
The *applicability of the model* evaluates whether the model is useful for describing real world interfaces. This model definitely is useful for representing the types of interfaces that exist in real world situations. The model was developed by observing humans interacting with computers using many varieties of interfaces; these observations, and therefore the corresponding interfaces, are accounted for in the model. For example, an interface as simple as a command-string-driven system, such as VAX DCL, can readily be described using the model. The end-user, responding to a single character prompt ($), types in the command string, the system processes it, and returns the results to the end-user. The results can either be a negative confirmation, in the case of a user input error, or a dialogue output transaction giving computational results, in the case of a valid user input. To represent this situation, in fact, takes only a small subset of the complete model. Other, more complicated interfaces, are also describable by the model; numerous examples are given in the discussion of scope of the model below.

The *scope of applicability of the model* evaluates the variety of classes of interfaces that the model can represent. Most common classes of interfaces can, in fact, be described by the model. The class of simple command strings was ex-
Another class of interfaces is menu-based, including list menus, pull-down menus, iconic menus, and networks of menus. For list menus, the prompt is the menu displayed on the screen, the token input is the code which the end-user selects, and confirmation might be a beep for erroneous input. Pull-down menus use a small temporary window for the menu list; that window, when displayed, is the prompt, the token input is often chosen with a mouse-type cursor, and confirmation is usually a beep. Iconic menus are typically a combination of graphics and text in a single display which is the prompt; the selection device may be a touch panel or cursor, such as arrow keys, mouse, or bit pad. The Lisa and Macintosh interfaces represent classes of both pull-down and iconic menus. For both these types of interfaces, the end-user input still produces a single token value; the prompt is more complicated. Menu networks are often very difficult to handle; in this model the control structure for a menu network is the underlying grammar of the interaction language, described directly by the syntactic level of the model. This level gives the sequencing from token to token within a command. No special treatment is needed for even the most complex control networks within this model.
Another class of interfaces which the model represents well is a forms-driven interface. The entire form is one transaction and each blank corresponds to a single interaction (token). Applications such as those that can be written using dBase II are examples of this type of interfaces.

Classes of interfaces that are heavily graphics oriented often carry over one part of a transaction to be a part of the prompt for the subsequent transaction. The chaining concept of the model is the mechanism which provides this connection from one transaction to the next, allowing the end-user to interact with graphical as well as textual objects on the screen.

Other classes of interfaces, including those which utilize voice input/output, pointing devices, and multi-windowing also fit the model. For voice devices, each spoken word is typically an interaction (token); the device simply provides a different source for the tokens. Similarly with pointing devices, the tokens merely come from a non-keyboard source. For interfaces with multiple windows, the appearance and behavior of any window is the same as that of a single screen which has no windowing; the general model applies to each window.

One type of interface for which more study of the model is needed is those which are time dynamic. That is, the di-
alogue output transaction develops dynamically even while a dialogue input transaction is still on the screen, and the system is concurrently activated for end-user input as well. If no input occurs within a specific time, the input transaction expires, possibly causing a change in the screen. If an input does occur, it is sent to the computational component, processed, and it, too, may cause a screen change. A typical example of such an interface is a video game; it represents the most extreme departure from the Input-Process-Output paradigm which the model best represents. In such an interface, the components of the model are the same as in a non-time-dynamic interface; the approach to token acceptance and passage to the computational component is generally the same. However, the turn-taking sequencing is altered, and more concurrency and interleaving of the components occur. The model needs shared data structures between the dialogue and computational components and a mechanism for signaling synchronization between these two components. The basic model still applies; only some small extensions need to be made to allow the model to fully describe time dynamic interfaces.

The precision of the model determines how completely the model can describe and accurately represent all the various details of an interface. The three levels of abstraction --
semantic, syntactic, and lexical -- upon which the model is based, allow the model to describe many levels of discourse in various types of interfaces. At the highest or semantic level, the description could be of a global behavioral issue, while at the lowest or lexical level, the description is of single end-user actions. The dimensions of the model also enhance its descriptive power, allowing it to represent even specific individual messages and end-user inputs. This tailoring of all parts of an interface to the needs of the end-user, based on specific dialogue states, gives the model added precision.

The uniqueness of the model can best be determined through a comparison of models with similar goals as described in the literature. As the related works section (Chapter 3) of this dissertation indicated, no other model describes only the dialogue component or interface of an interactive system in such detail. The three levels of abstraction, coupled with the dimensions, allow a quite complete description of a human-computer interface. Many models have addressed various parts of interfaces, but few have addressed all aspects of the interface. A compiler-compiler, for example, treats only the language input part of an interface; it provides no mechanisms for integrating the other parts (the prompts and confirmations). Many models also include all or part of the
computational component, in addition to the dialogue component.

9.4 FUTURE EVALUATIONS.

This was a very limited preliminary evaluation of the dialogue transaction model through its instantiation in AIDE. Additional testing for different purposes should be done. One proposal would be to give a group of dialogue authors a requirements specification for an interface and then let them design the interface and break it down into transactions and interactions, rather than having the interface, transactions, and interactions completely defined in advance. A new version of AIDE, which will include the dimensions and other aspects of the model, must also be tested. Integration of the dialogue engineering methodology and AIDE must be assessed. These are only a few of the many exciting possibilities for future evaluations of the model, the methodology, and AIDE.
Chapter X

SUMMARY AND CONCLUDING REMARKS


A primary goal of this dissertation research has been to give insight into the essence of human-computer interaction. Without this insight, the interface through which a human and a computer communicate is a random collection of displays, devices, inputs, error messages, and help information. The underlying, hidden rules that help organize and explain the parts of an interface reveal the deep structure of interfaces. It is this deep structure, and not the surface events, which is useful in guiding the interface design process.

This research has presented many of the issues associated with modeling and specification of human-computer interaction; few other models exist. While formalisms for specification of static programming languages abound, these notations are simply not adequate for specification of human-computer interfaces. In addition, these notations are generally not easily understood by dialogue authors.
Much of the reason for the inadequacy of models, specification techniques, and tools for implementing human-computer interfaces of interactive systems is due to the special characteristics of such systems. Human-computer interfaces must be considered not merely with respect to manipulation of data, but also in terms of the human element. The very nature of this highly variable component increases the complexity of human-computer systems and therefore their interfaces. The wide range of input/output devices also contributes to the complexity of interfaces.

However, research in the formal modeling and specification of human-computer interfaces is progressing. Models are beginning to emerge to explain the structure of human-computer interaction. To this end, this dissertation research, set in the context of the Dialogue Management System (DMS), has had five major contributions. It has postulated a theory of human-computer interaction, which is a description of the inherent properties of human-computer interaction -- its phenomena, its elements, and their relationships. The theory was formulated by observing people interacting with a variety of interface types. This theory proposes that the elements involved in human-computer communication are essentially human input, computation, and computer output, often (but not always) in an Input-Process-Output (I-P-O) or "turn-taking" configuration between human and computer.
Formalization of the theory led to a structural, descriptive, language-oriented multi-dimensional dialogue transaction model. Such a language orientation, based on three traditional levels of language (semantic, syntactic, and lexical), is useful for seeing beyond the surface differences in form and content and dealing with the various kinds of communication in a uniform manner. By viewing human-computer dialogue as an interaction language, the problem of modeling, specifying, and implementing dialogue becomes, to a great extent, a generalized problem of modeling, specifying, and implementing languages. Dimensions of the model are orthogonal, and combine to allow tailoring of an interface to specific states of the dialogue. Both the language orientation and the dimensions are transparent to the end-user at run-time; they are mechanisms for guiding the dialogue author at interface design- and implementation-time.

Basic elements of the model are a dialogue input transaction, at the semantic level, which is a sequence of one or more interactions to extract a valid user input. This input is a set of linguistically related tokens (i.e., it can be described with a grammar). Intuitively, a transaction is one command language, and a transaction instance is one occurrence of a complete valid command. The purpose of an interaction, at the syntactic level, is to extract a single
valid token value from the end-user. An interaction is comprised of a system syntactic prompt, a human token input, and a system syntactic confirmation. A token input is made up of a sequence of one or more actions. The purpose of an action is to extract a single valid lexeme value from the end-user. An action is comprised of a system lexical prompt, a human lexeme input, and a system lexical confirmation.

This model is a design and implementation model and, as such, has two major manifestations: a dialogue engineering methodology and an integrated set of interactive dialogue implementation tools. A dialogue engineering methodology deals with the decomposition of dialogue transactions. It is a set of procedures and a specification notation for designing elements of the model. The methodology has a two-level approach to interface design. At the top level, it is graphical for the design of interactions within a transaction. For the prompt, input, and confirmation parts of an interaction, the approach is forms-based; the dialogue author "fills-in-the-blanks" to describe each interaction part. An Author's Interactive Dialogue Environment (AIDE) is an interactive dialogue implementation tool for constructing dialogue transactions. It is based on the concept of direct manipulation or "what you see is what you get", so that a dialogue author can implement an interface without writing programs.
Finally, an evaluation of the research has been done to determine the efficacy of the work. Specifically, a group of subject dialogue authors used AIDE to create and modify a prespecified task interface, and a group of subject application programmers used a programming language to create and modify the identical interface. The dialogue author subjects performed the task in a mean time of just over one hour, while the application programmer subjects averaged nearly four hours. The results support the hypothesis that implementation of an interface is faster and easier using AIDE than using a programming language.

This research has considerable breadth in its treatment of human-computer interface concepts. Several of these concepts have been developed in depth. The others provide context, and connect this research to other efforts in the field of human-computer interaction. Thus, a rich spectrum of issues has been raised as possibilities for future research in this exciting area. Observations of numerous types of interfaces will continue, in order to keep the theory of human-computer interaction current. The basic multi-dimensional dialogue transaction model has, for some time, remained stable in its ability to describe a variety of interface types. It will continue to evolve as necessary to accommodate an ever-increasing variety of interfaces.
The dialogue engineering methodology has great potential for future research; it needs extending and evaluating to determine its usefulness in specifying human-computer interfaces. A new version of AIDE is already in the planning stages; it will incorporate many of the features of the model and complement the methodology. Evaluation of all phases of the work will also continue. Such theories, models, methodologies, and automated tools promise to contribute greatly to the ease of production and to the quality of human-computer interfaces.
REFERENCES


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


Williams, M.H. "A Flexible Notation of Syntactic Definitions," ACM Trans. on Programming Languages and Systems. 4,1 (January 1982).


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


The vita has been removed from the scanned document