Dialogue Management as an Integral Part of Software Engineering

H. Rex Hartson
Deborah Hix
Thomas M. Kraly

TR 87-22

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 20(5), 425–457 (MAY 1990)

Developing Human–Computer Interface Models and Representation Techniques

H. REX HARTSON AND DEBORAH HIX
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, U.S.A.

AND

THOMAS M. KRALY
IBM Corporation, 6600 Rockledge Dr., Bethesda, MD 20817, U.S.A.

SUMMARY

The Dialogue Management Project at Virginia Tech is studying the poorly understood problem of human–computer dialogue development. This problem often leads to low usability in human–computer dialogues. The Dialogue Management Project approaches solutions to low usability in interfaces by addressing human–computer dialogue development as an integral and equal part of the total system development process. This project consists of two rather distinct, but dependent, parts. One is development of *concepts* for dialogue management, and the other is *implementation* of a dialogue management system (DMS) to evaluate these concepts. The goal of this paper is to describe our approach to the development of two of these *conceptual* aspects and how we oriented those toward the needs of practical *implementation*.

The two conceptual aspects are

(a) a structural, descriptive *model of human–computer interaction*, and
(b) *Techniques for representing* both the behavioural (end-user's) view and the constructional (developer's) view of dialogue.

The approach to their development was a technology transfer process that was part of a two-year university/industry research liaison between the Dialogue Management Project and IBM Federal Systems Division (FSD), now called Systems Integration Division. Part of this liaison was aimed at moving our research ideas and results into a real-world dialogue development environment. Following presentation of the technical problems and solutions, the paper concludes with a discussion of results of our liaison and by raising and addressing some questions of mutual interest that arose during our co-operative interaction.

KEY WORDS Human–computer interface development Human–computer interface management Dialogue models Dialogue representation techniques User interface management systems (UIMS) Technology transfer

THE DIALOGUE MANAGEMENT PROJECT

Background and overview

The Dialogue Management Project at Virginia Tech is studying the poorly understood problem of human–computer dialogue development. This problem often leads

0038–0644/90/050425–33$16.50
© 1990 by John Wiley & Sons, Ltd.

Received 3 January 1989
Revised 19 October 1989

to low usability in human–computer dialogues. The Dialogue Management Project approaches solutions to low usability in interfaces by addressing human–computer dialogue development as an integral and equal part of the total system development process. This project consists of two rather distinct, but dependent, parts. One is development of *concepts* for dialogue management, and the other is *implementation* of a dialogue management system (DMS) to evaluate these concepts.

Research objectives of the Dialogue Management Project are to develop a theoretical understanding of the process of human–computer interaction, to produce and evaluate an experimental system (DMS) as a test-bed for our conceptual work and to bridge the gap between computer scientists and behavioural scientists in the dialogue development process. The goal of this paper is to describe our approach to the development of two of the *conceptual* aspects and how we oriented those toward the needs of practical *implementation*.

The two conceptual aspects are

(a) a structural, descriptive *model of human–computer interaction* (see section entitled 'Modeling human–computer interaction'), and
(b) *techniques for representing* both the behavioural (end-user's) view and the constructional (developer's) view of dialogue (see section entitled 'The dialogue representation process').

The approach to their development was a technology transfer process that was part of a two-year university/industry research liaison between the Dialogue Management Project and IBM Federal Systems Division (FSD), now called Systems Integration Division. Part of this liaison was aimed at moving our research ideas and results into a real-world dialogue development environment. Following presentation of the technical problems and solutions, the paper concludes with a discussion of results of our liaison and by raising and addressing some questions of mutual interest that arose during our co-operative interaction.

Perspective from IBM Federal Systems Division

IBM recognizes the need for an improved methodology for human–computer interaction. End-user interfaces are critical components of systems, providing the window for the end-user into the system's functional capabilities and strongly influencing the end-user's perception of the system. There has been much interest in principles of good human factors of end-user interfaces, but there has been less work done on the dialogue development process specifically.

Under the direction of Harlan Mills, a major software engineering program was developed within IBM, beginning in 1977. An extensive practices program documented how the key technologies were to be applied to projects in IBM, and a supporting education program was developed and implemented. Over 2300 programmers and managers were taught the concepts. The program was aimed at training IBM software personnel (both programmers and managers) to use software engineering practices. Features of the program included

(a) using mathematical concepts, models, and analytical techniques
(b) viewing programs (i.e. pure procedures) as having the properties of mathematical functions

(c) viewing collections of programs (e.g. data abstractions) as having the properties of mathematical state machines
(d) using mathematical logic to study the correctness of software
(e) using stepwise refinement to improve the development and representation process of software designs by capturing levels of abstractions
(f) using a textual design language to improve the representation of designs
(g) separating specification from implementation as an integral part of stepwise refinement
(h) Emphasizing encapsulation of data in software modules, with limited access to the data through well-defined functional operations, and
(i) using modern software management techniques that improve control of the process and products.

A more complete description of the IBM Software Engineering Program is given in Reference 1.

The Software Engineering Program did not, however, include a methodology for the development and representation of the human–computer dialogue for a system. The courses assumed a static representation of an abstract interface, which resulted in a software design represented in a textual program design language. The dynamics of end-user interaction were not part of the formal specification process. This required the software developer to address the problem in the software development phase in an *ad hoc* manner.

During the same time period, the importance of systems engineering increased, owing to expansion in the scope of many IBM projects, stressing activities at the front end of a project, such as requirements analysis, systems architecture, design and formal specification writing (often according to military standards). The systems engineering organization is separate from the software development organization. A course titled 'Systems Engineering Principles and Practices' has been developed and is being presented to a large number of systems engineers and a smaller number of software engineers in IBM. In addition, human factors specialists are being used to educate both systems and software engineering organizations in the principles of human factors.

With the emergence of these two rather independent technical thrusts—systems engineering and software development—in different organizations, the responsibility for the end-user interface has become blurred. In principle, the systems engineering organization establishes the specification for the end-user interface as well as other aspects such as software, hardware and safety, with the support of the respective development organizations. Then each development organization creates the required implementation. Thus, the evolution of a formal methodology for end-user dialogue and where it fits into the IBM technology process has become an obvious and critical question that needs answering.

The co-operative work between the Dialogue Management Project and IBM raised such issues as

(a) differences in needs and constraints of a dialogue developer as compared to those of a software developer, and bringing those two roles together, and
(b) application of existing techniques and tools from each role (dialogue developer and software engineer) to the other.

Questions and answers addressing these issues more specifically are given in the section entitled 'Issues addressed in the co-operative work'.

TERMINOLOGY

Target system architecture

Some terminology is important for establishing a common understanding of several terms that will be used in this paper. Particularly important is an understanding of the architecture for a target system produced using the Dialogue Management Project approach. The logic of the target system is broken into three components, shown in Figure 1. The *computational component* contains the semantic functionality of the target system. The *dialogue component* contains dialogue logic and content, such as displays, error messages and input processing. The *global control component* governs logical sequencing among dialogue and computation. Each component of the system logic is produced in software and mapped into hardware.

The dialogue component is the collection of units of dialogue functionality, and it contains dialogue logic. It also contains some computation, but *only* computation that directly supports dialogue, such as validation of end-user input. Dialogue does *not* include any semantic computation for target system functionality. The computational component is the collection of units of semantic computation of target system functionality, and does not contain any dialogue at all. The global control component contains high-level logic for sequencing among dialogue and computational units. It invokes separate dialogue and computational units. (The term 'unit' is used in these definitions to represent, for example, a function, a module, or a procedure; it can be either elemental or decomposable). Emphasis in this paper (and in the Dialogue Management Project research in general) is on developing the dialogue component of an interactive system.

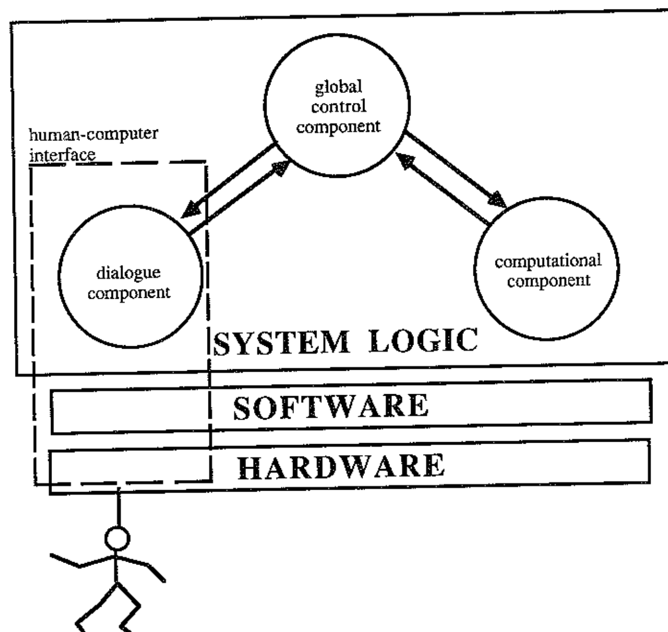


Figure 1. Target system architecture

Kinds of dialogue

Ideally, the terms *human–computer dialogue* and *human–computer interface* are defined separately to denote, respectively, the communication between a human user and a computer system, and the medium for that communication. Thus, a dialogue is the observable two-way exchange of symbols and actions between human and computer, while an interface is the supporting software and the hardware through which this exchange occurs. However, the two terms are tied closely together in the development process and we shall use them synonymously here just as they are in most of the literature.

Two basic types of human–computer dialogue have emerged as the most prevalent in interactive systems: sequential dialogue and asynchronous dialogue. In *sequential dialogue*, the end-user describes to the system what to do, often by using a command language. Sequential dialogue moves in a predictable manner from one part of the dialogue to the next, and allows both end-users and developers to visualize specific logical sequencing behaviour. Examples of sequential dialogue include request/response interactions, typed command strings, navigation through networks of menus and data entry. In non-sequential or *asynchronous dialogue*, the end-user shows the system what to do by 'grabbing' and manipulating, for example with a mouse, visual representations of dialogue objects. The well-known direct manipulation, or WYSIWYG ('what you see is what you get'), interaction style is often used with asynchronous dialogue. In sequential dialogue, the system presents the end-user's work one task at a time. In asynchronous dialogue, many tasks are available to the end-user at one time. Both kinds of dialogue are found in most current human–computer interfaces. However, the dialogue transaction model (see section entitled 'Modeling human–computer interaction') and representation techniques (see section entitled 'The dialogue representation process') presented in this paper are primarily applicable to sequential dialogue.

Dialogue independence

Dialogue independence[2] is a term coined in the Dialogue Management Project to refer to an approach in which design decisions affecting only the dialogue/interface component of the target system are isolated from those affecting application system structure and computational software. In practice this means, for example, that appearance of the interface to the end-user and choices of interaction styles (e.g. a menu) used to extract inputs from the end-user are not known either to the computational software or its developers. After working together with the application programmer to formally define the names, values and specifications comprising the communication between the dialogue and computational components, a dialogue developer can work on the dialogue design relatively independently of application programmers. Dialogue independence also means that multiple interfaces can be developed for the same target system computational functionality.

Dialogue independence goes well beyond just separating the target system into components. The key is in knowing how to structure the system for achieving a useful separation, choosing where and how to separate. Three major aspects of system development—methodologies, models and interactive tools—contribute to the support of dialogue independence in an interactive system development environment; each of these three topics is a major thrust of the Dialogue Management Project research. The dialogue management representation techniques (see section entitled 'The dialogue

representation process') integrate human–computer interface management and software/ systems engineering at a high level, but address their design activities separately and in a manner tailored to the needs of different developer roles (e.g. dialogue developer and application programmer). Structural modelling of the dialogue component (see section entitled 'Modeling human–computer interaction') into several levels of abstraction compartmentalizes the development process, and provides mappings between device- and interaction-style-dependent dialogue and a normalized (device- and style-independent) view of dialogue for the rest of the system. From the global system view down to details of a specific device, there are many decisions about how to separate one level from another—semantic from syntactic, syntactic from lexical, and lexical from physical end-user actions and gestures. Each new layer introduces another level of internal interface and the mappings to adjacent levels. Interpretation of the levels influences how the basic—and sometimes arbitrary—separation is made between what is called dialogue and what is called computation in a given system architecture.

The structural model and the representation techniques of the Dialogue Management Project reasearch are supported with an integrated set of interactive tools called the dialogue management system.[3] These tools have a direct manipulation interface for the dialogue developer and produce dialogue definitions that are interpreted, essentially eliminating the need for the dialogue developer to write lines of source code. The dialogue can be easily modified through the tools and parts of the dialogue can be exercised separately for rapid prototyping. Prior to the co-operative research efforts with IBM, researchers in the Dialogue Management Project created and empirically evaluated a set of dialogue development tools called AIDE (the dialogue author's interactive dialogue environment). Findings[4,5] showed a nearly four-to-one improvement with the use of AIDE versus traditional programming to create and modify a human–computer interface. In addition to this empirical evaluation, AIDE was used to produce several modest 'demonstration' interfaces (e.g. menu-driven and PF-key driven) that executed with the same computational component, demonstrating that the concept of dialogue independence is feasible for the development and implementation of at least some classes of interfaces.

Beyond these evaluations, the dialogue independence concept has been continually tested within the Dialogue Management Project for about seven years, applying it to the development of numerous types of interactive systems. General findings are that it is quite easy to apply in a sequential system in which a turn-taking paradigm of input–process–output is used. This pattern makes it fairly easy, during the development stages, to delineate among dialogue, computation and global control. We have found that separation can be more difficult to determine and achieve in systems, such as those having direct manipulation interfaces, in which the dialogue and computation tend to be more closely interleaved in time and may share a common data representation of the interface and application objects. Nevertheless, design decisions regarding appearance and behaviour of the interface can be kept independent from those for the software that manipulates the corresponding data structures.

We have also found that dialogue independence is not entirely without difficulties or drawbacks. For example, some approaches to dialogue design tools require considerable *a priori* knowledge of the style of interaction (e.g. menu, use of a mouse) in the dialogue interpreter and tools themselves. New interaction styles, techniques, or devices require significant new programming in the interpreter and tools. Another difficulty stems from having the separate role of dialogue developer to produce the dialogue

component, potentially increasing the need for communication among more developers. Further, separation of dialogue software from computational software can potentially cause a decrease in performance (although to the authors' knowledge, no data have been collected to demonstrate this). This can be overcome, at least to some extent, by system architectures emphasizing, for example, concurrent execution of dialogue and computation components. In sum, the considerable advantages of dialogue independence appear to outweigh the disadvantages.

MODELLING HUMAN–COMPUTER INTERACTION

A variety of 'dialogue models' has sprung into existence over the last decade in response to the need for organizing the process of human–computer dialogue development. Structural models describe the generic process of human–computer interaction, and can be used to guide a dialogue developer in constructing the dialogue. Within the Dialogue Management Project, a structural, descriptive *dialogue transaction model* has been developed. It identifies observable linguistic objects in the behavioural (end-user's) domain, and defines processing of those objects in the constructional (developer's) domain of the dialogue. This model is useful primarily for developing sequential, turn-taking dialogue. This section presents the basic parts of the model and how it is used to describe human–computer dialogues.

Motivation for structural modelling of human–computer interaction

A structural, descriptive model of human–computer interaction is needed to guide a dialogue developer during both the design and the representation process. Models of dialogue also aid readability of the design. Without such models, dialogues can be unstructured collections of prompts, error messages, cursor movements, end-user inputs, confirmations, graphics and text. Without the guidance offered by such models, dialogue development can be an *ad hoc*, often disorganized process. At least part of the reason for the existence of so many poor quality human–computer interfaces revolves around the lack of structural models to guide dialogue development. With such models, the dialogue developer can follow an organized process, methodically developing the dialogue as defined by the model.

Only a few such structural, descriptive models exist, and they are primarily for describing sequential dialogue. Rarely have these models been applied to development of a human–computer interface in a real-world development environment; most appear to be theoretical and untested in practice. One structural model for describing sequential dialogue is a 'dialogue cell',[6] which consists of four basic elements that define dialogue structure: a prompt that indicates types of input to be entered, a symbol that is the end-user's input, an echo that is the system's interpretation of the symbol, and a value that is the mapping of the symbol to data usable by the computational component. Basic dialogue cells, comprised of these four elements, can be combined to represent sequences of human–computer exchanges. Another structural model is an 'interaction event',[7] developed as the main component of a dialogue; a dialogue is a sequence of these events. An interaction event contains a system prompt which indicates to the end-user that an input is desired, the system responds with an action based on the input, and flow control determines the next interaction event. Other dialogue features

in interaction events include input checks, a help feature, an escape mechanism, and default values for missing end-user input.

Structure of the dialogue is important in at least two domains during the dialogue development process: the *behavioural domain* and the *constructional domain*. The *behaviour* of the dialogue is what the end-user sees, hears and does while interacting with the computer system. The *construction* of the dialogue determines how the computer system is designed in order to make the behaviour happen. Thus, the behavioural domain is of interest to the end-user and dialogue developer during the dialogue development process, whereas the constructional domain is of additional interest to the dialogue developer and implementer. Construction of dialogue involves both the underlying structure of the design, as well as the form and content of dialogue attached to that structure. Form and content of dialogue is the subject of human factors and behavioural science studies. The work presented in this article concentrates on the constructional domain, and, within that domain, on the structural design of dialogue. Our focus is to identify objects in the behavioural domain and then to determine what corresponding objects in the constructional domain must be developed in order to produce those behavioural objects for the end-user of the system.

Derivation of the dialogue transaction model: observing human–computer interaction

The dialogue transaction model was empirically derived by the Dialogue Management Project researchers over several years of observations of end-users interacting with computer systems. We observed numerous interface types and styles, interaction techniques and hardware devices within the context of industrial development environments, usability testing laboratories, student projects, videotapes of interactive system usage, and our own experiences with innovative interfaces. Creative, innovative interfaces that were observed included those of the iconic mouse-driven Macintosh, an airline reservation system, a document retrieval system, an electronic mail system, a carrier-based air traffic control system, windowing systems, form-filling systems, and so on. We were looking for recurrent types of objects, relationships and common patterns within this wide variety of interfaces. An object, in this context, is an entity within the dialogue. Some of the early analysis from these observations revealed underlying grammatical structures. In fact, we deduced a hierarchy of three linguistic *behavioural objects* (similar to those described in Reference 8)—sentences, tokens and lexemes—that are associated with the observable two-way flow of symbols and actions that constitutes dialogue between humans and computers. We then created a corresponding hierarchy of three *constructional objects*—transactions, interactions and actions—as mechanisms for processing the behavioural objects.

Behavioural objects

A behavioural object called a *token* is the smallest sequence of lexical elements that can have formally defined meaning in terms of the application. One word of a typed command or a single spoken word are examples of tokens. A behavioural object called a *lexeme* is the smallest unit of raw input from the end-user; a token is comprised of a sequence of one or more lexemes. A lexeme results, for example, from the pressing

of a single keyboard key or from a mouse pick of an iconic menu—any single input action by the end-user. The mouse pick of an icon is both a lexeme and a token (i.e. a token containing a single lexeme). A behavioural object called a *sentence* is a sequence of one or more related tokens (just as it is in natural language). A complete command with all its parameters is an example of a sentence. To illustrate, a typed command to request a directory listing, SHOW DIR, is a sentence; SHOW is a token; and S is a lexeme. This hierarchical concept applies to all other kinds of interaction styles and devices as well (e.g. voice recognition, track ball, joy stick, foot pedal, graphics). All input devices translate the end-user's physical actions and gestures into byte streams that are interpreted as lexemes.

Constructional objects

Having identified a hierarchy of behavioural objects, we wished to create a corresponding hierarchical set of objects in the constructional domain to process those behavioural objects. A constructional object called an *interaction* (e.g. processing of SHOW in the above example) is a function that maps lexemes to valid, normalized token values. A constructional object called an *action* (e.g. processing of a keystroke of S in the above example, or a mouse click) is a function that maps end-user actions to a valid lexeme. A *transaction* (e.g. processing SHOW DIR in the above example) is the constructional object that accepts from the end-user, processes linguistically, and returns a syntactically valid sentence. The structural part of dialogue design is now largely a problem of developing these constructional dialogue objects to satisfy behavioural needs.

The dialogue transaction model

The *dialogue transaction model* is the hierarchical set of behavioural objects plus the corresponding hierarchical set of constructional objects, as shown in Figure 2. Each of the constructional objects is composed of *constituent objects* of these types:

(a) display, of which there are three kinds:
  (i) informative
  (ii) prompt
  (iii) confirmation
(b) input
(c) dialogue computation, of which there are three kinds:
  (i) validation
  (ii) mapping
  (iii) general.

Constituent objects types

Within a transaction, a *display object* causes a presentation to the end-user. For example, a display may be text or graphics on a CRT terminal; it may be voice or music from a synthesizer; or it can be graphical animation or a moving picture from

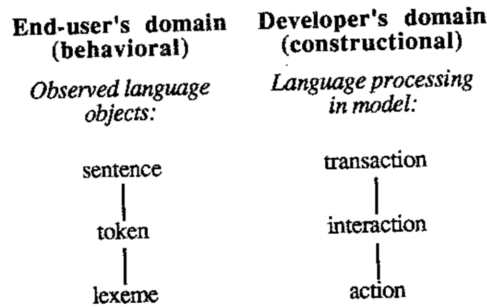


Figure 2. Dialogue transaction model hierarchy: behavioural and constructional objects

a video disc. Display objects that can be completely defined prior to run-time are *static displays* (e.g. a fixed 'welcome' message), whereas *dynamic displays* require form and/or content to be determined at run-time (e.g. a bar chart or results of a database search to find a default value). Depending upon whether its list of choices is invariant, a menu, for example, can be either a static or a dynamic display object.

Because some kinds of display objects recur frequently and have very specific purposes, the dialogue transaction model has three kinds of display objects: informative displays, prompts and confirmations. An *informative display object* is used for all displays to the end-user except those used for special purposes of prompting or confirming. In practice, informative display objects are used for presenting computed results to the end-user (e.g. data retrieved from a database) or for other information (e.g. on-line help) to the end-user. A *prompt object* (e.g. a message or prompt symbol) requests an input (generally a token or a lexeme) from the end-user. A *confirmation object* is for presenting positive confirmation (e.g. 'atta girl') or negative confirmation (e.g. an error message) of validity in response to end-user inputs (either tokens or lexemes).

An *input object* is a constituent object for accepting raw (unprocessed and uninterpreted) end-user inputs. The purpose of an input object within an interaction is to accept (take in but not process) a raw token value as a result of a sequence of actions. Action input objects accept raw lexemes from the end-user and are where all direct physical end-user input happens.

Definitionally, pure dialogue could be limited to include only the output and input symbols that flow to and from the physical terminal device. This limitation, however, does not lead to an interesting view of dialogue from a structural viewpoint. Processing of, and sequencing among, end-user inputs is also part of dialogue. Therefore, any 'computation' that directly supports dialogue and is performed in the dialogue component is called a *dialogue computation object*.

Because some kinds of dialogue computation objects recur frequently and have very specific purposes, the dialogue transaction model has three kinds: validation, mapping and general dialogue computation. A *validation object* checks end-user inputs against pre-established criteria for lexical and syntactic validity. These objects are often rather complex, since they typically involve parsing of behavioural objects. Validation objects are used within interactions to validate a token after all its lexemes are entered, and within an action, validation objects do most of the same kind of checking found in interactions, but on a dynamic lexeme-at-a-time basis. In this model, end-user inputs

are lexically and syntactically validated within the dialogue, rather than in the functional software of the application. Within a transaction, validation objects are responsible for validation of the grammar (inter-token relationships and constraints) within a sentence. As an example, in an airline reservation system it may be desired to prevent, or at least issue a warning for, the booking of a flight for which the departure city is the same as the arrival city. In this case, *departure city* and *arrival city* are tokens returned by separate interactions and the constraint must be enforced at the next higher level within a transaction. Rules and predicates that define sentence validation criteria can involve any variables or constants known to the transaction, including any values passed into the transaction at its invocation.

A *mapping object* translates raw end-user input tokens into *normalized token values* that are understood by the rest of the application system. Whereas all other constituent object types can appear in each of the kinds of constructional objects, mapping objects occur only in interactions. This mapping allows all parts of the design above the interaction in the constructional hierarchy (i.e. transactions and all non-dialogue parts of the target system) to be independent of input devices, interaction styles and input techniques. For example, suppose that a given token value can be entered by the end-user in a number of different ways via concurrently active input techniques such as those represented by a mouse, a keyboard and a voice recognizer. All different raw token values from these various devices can be mapped to the same normalized token value. The design of the rest of the system is independent of whether that token value was entered via mouse pick of a graphical icon, typing on the keyboard, or voice recognition. This mapping is the key to *dialogue independence*, discussed previously in the section entitled 'Dialogue independence'.

All other kinds of computation that directly support dialogue are called *general dialogue computation*. An example is the computation of a default token value for an interaction display, something which is clearly part of dialogue. There is also a 'grey' area in which examples of computation can be classified as either dialogue or functional semantics. An example is semantic validation of an end-user input, such as a file name requiring an invocation of the file directory function to determine if the named file exists. Classification of this kind of function requires judgement of the dialogue developer and/or systems engineer and can be based on such factors as performance or convenience of implementation. Of course, there is also a clear non-dialogue computational component of each application system, which includes all processing that is explicitly part of the semantic functionality or task of the application system.

Levels of abstraction in the model

The hierarchical relationship among constructional objects of the model aids in organizing dialogue into levels of abstraction, each level helping to control complexity by hiding details of levels below it. Figure 3 shows a typical configuration of constructional and constituent model objects, indicating levels of abstraction and relationships among objects. At the highest level of abstraction shown, there is global sequencing among dialogue transactions and functional computational processing. Each transaction associated with end-user input (i.e. all transactions except those that do nothing but produce an informative display) contains one or more interactions, plus transaction-level displays, prompt, validation, confirmation, and dialogue computation, any of

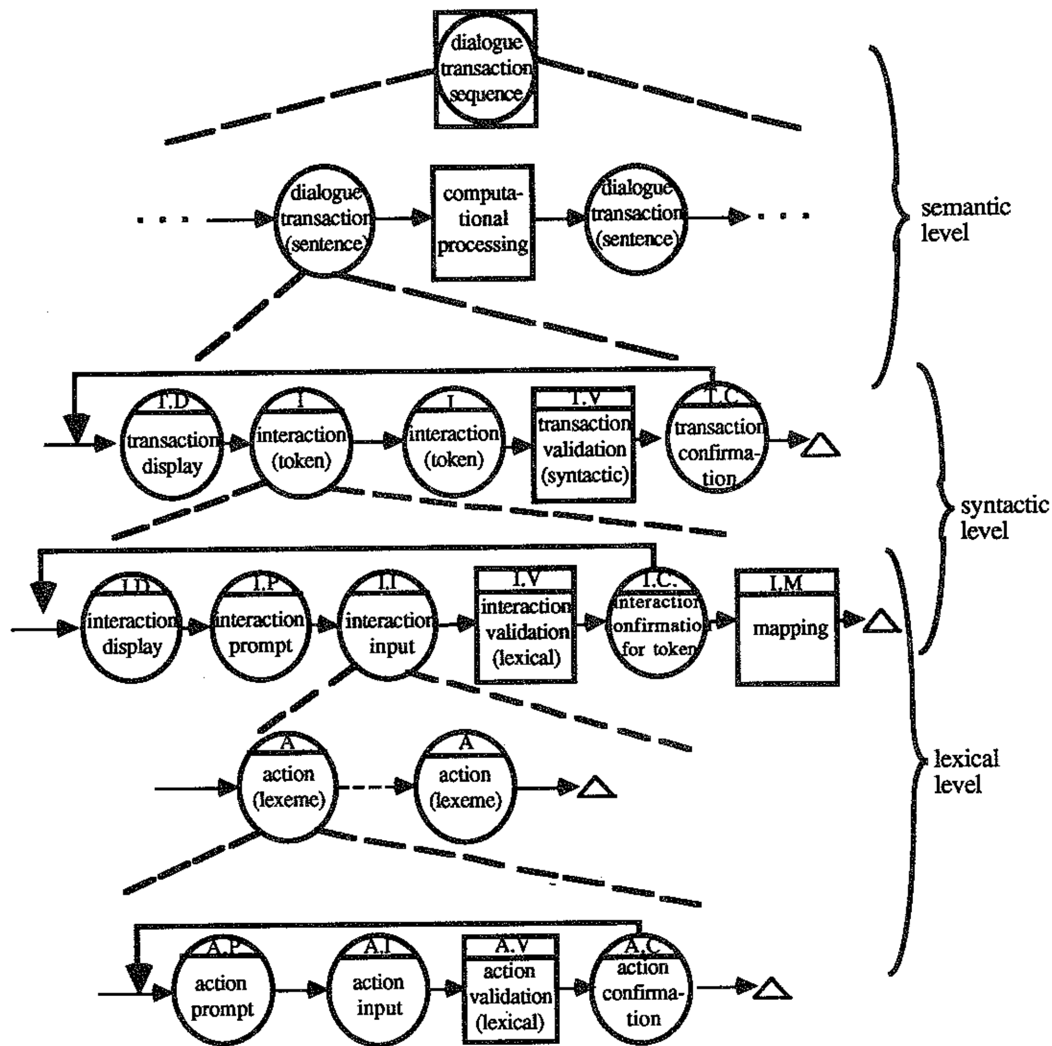


Figure 3. Levels of abstraction in the dialogue transaction model

which can be null. The relationship among interactions represents the grammar of the end-user sentences, independent of interaction styles and techniques.

An interaction contains displays, prompt, token input, validation, confirmation, mapping and dialogue computation, of which display, prompt, confirmation and dialogue computation can be null. An action contains displays, prompt, lexeme input, validation, confirmation and dialogue computation, any of which can be null except lexeme input and validation.

The dialogue transaction model is well-suited for sequential dialogue with significant linguistic structure, such as parts of a dialogue relating to commands, parameters, selection of choices, data entry, and values requiring parsing and/or validation. To define highly interactive kinds of non-sequential (asynchronous) dialogue, such as those found in direct manipulation interfaces, we are developing a user action notation

(UAN)[9] that is a metalanguage for describing end-user actions, system feedback and system state. Discussion of this notation is beyond the scope of this paper.

Applying the dialogue transaction model: an example

Figure 4 shows a transcript that is a fragment of sequential, turn-taking dialogue from a very simple hypothetical employee leave system to receive and process interactively, by mailing the request to the employee's manager, employee requests for leave time. There are two possible types of leave: vacation leave and sick leave. The menu shown in the

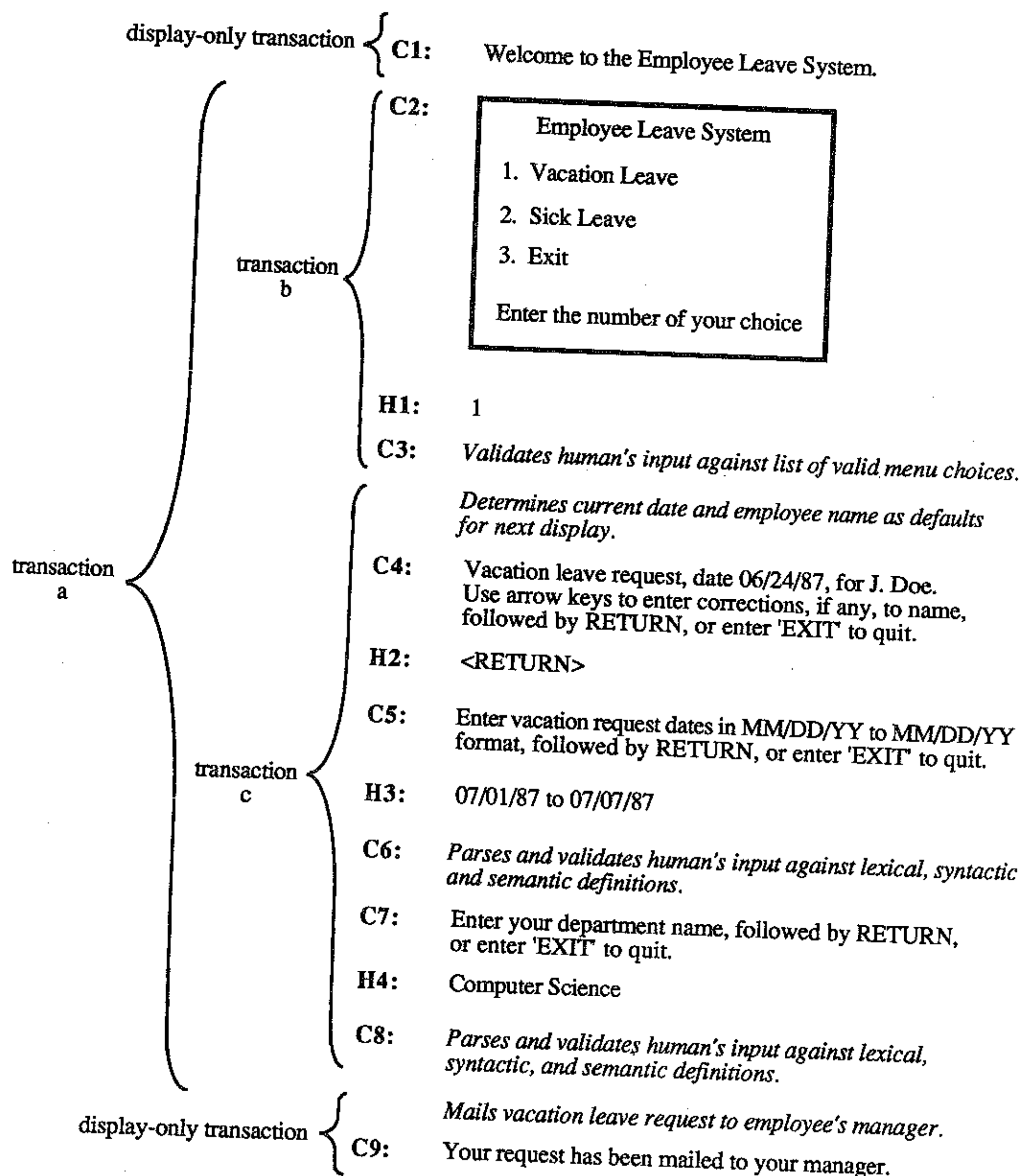


Figure 4. Dialogue transcript from the employee leave system, with some alternative groupings of transactions

box in Figure 4 is used for choosing between these; this transcript fragment is the result of choosing vacation leave. (An important point here: no claim is made that this is a 'human factored' dialogue; it is simply presented here to be used as a running example throughout this paper.) 'C' indicates computer dialogue and 'H' indicates human dialogue; regular fount represents what is seen on the screen and italics represent computer internal processing. Following is a discussion to illustrate identification and structuring of behavioural and constructional objects in this transcript.

C1 is an example of a computer output resulting from a *transaction* used only as an informative display. C2 is a *prompt* of menu choices contained in the *interaction* associated with H1, an end-user *input* token. The first part of C3 indicates *validation* for this same interaction. The second part of C3 is *general dialogue computation* to compute the employee name and current date that are used in this interaction's *positive confirmation*, found in the first part of C4 (up to the word 'Doe.'). As an alternative, it is also possible, depending on the preference and judgement of the dialogue developer, to make the first part of C4 an *informative display* for the subsequent interaction. In either case, the second part of C4 (instructions to enter corrections) is the *prompt* for the second *interaction* of the transcript. Also in either case, 'J. Doe' is the default value for the employee name token extracted by the second interaction (to get the name of the employee requesting vacation leave). H2 returns the default token, 'J. Doe'. *Positive confirmation* for the second interaction is implicit.

C5 is the *prompt* for the *interaction(s)* associated with H3. The number of tokens in H3 depends upon how the dialogue developer wishes to define linguistic constraints for the two date values that must be entered. The most probable definition would separate month, day and year each into individual tokens, making a total of six interactions in H3. The delimiters ('/' and 'to') can be automatically filled in. C6 is *validation* for the tokens of H3.

C7 is the *prompt* for the final *interaction*, to extract the employee's department name *input* token of H4. The first part of C8 is *validation* for this final token. The second part of C8 is non-dialogue computation (functional semantics of the employee leave system). C9 indicates a separate *transaction* with only an informative display used as a semantic confirmation for feedback to the end-user.

Since this simple example is keyboard-oriented, all single keystrokes of the end-user are lexemes. It is interesting to note that the '1' of H1 is both a lexeme and a token, processed first by an action and then by an interaction, whereas the '1' in the date value of H3 is only a lexeme.

Identification of transactions and even interactions can vary, depending upon judgement of the dialogue developer. C1 and C9 are definitely separate transactions containing only informative display objects. There are several alternatives for grouping into transactions all interactions beginning at C2 and going through C8. Unless there are reasons for doing otherwise due to linguistic structuring of the application, the most probable grouping is to combine all these interactions into one command-language-oriented transaction, as shown by the large brace labelled 'transaction a' in Figure 4. In this case, the token returned from the menu is the command name, and the other token values are command parameters (operands). An alternative grouping is to separate the interaction containing the menu into one transaction, labelled 'transaction b' in Figure 4, with the other interactions grouped together into a separate transaction, labelled 'transaction c'. A third alternative is to have each interaction be a separate

transaction. This alternative offers flexibility, but because the sentence/transaction grouping is not used in modelling the end-user's language, the ability to represent grammatical relationships among tokens directly (e.g. which parameters follow choice of 'vacation' for the first token) is lost. Yet another possible grouping is to put the tokens making up the date values of H3 into a single transaction to elicit dates.

Evaluation of the dialogue transaction model

The dialogue transaction model presented in this paper is based on linguistic and formal language theory. It has been evaluated through long-term practical application to numerous dialogues. As a result of this continuing application, it has been iteratively refined and enhanced. Although this model can be used manually for structuring human–computer dialogues, its power is more completely realized when the model is built into human–computer dialogue development tools (often called user interface management systems or UIMS; see the section entitled 'Interactive dialogue development tools' and also see Reference 10) where it can be used directly to organize and guide the dialogue developer. This dialogue transaction model addresses dialogue development in a very broad sense, giving structure to the process of constructing human–computer interfaces.

THE DIALOGUE REPRESENTATION PROCESS

An unsolved problem in dialogue development is a physical representation process that completely captures the products of the mental process of conceptual dialogue design. Much is known about *what to represent*, but to date there does not appear to be a notation (other than implementation code) that completely tells *how to represent* the total dialogue design. Such a representation technique is needed to organize the enormous number of *things* that have to be represented during dialogue development.

There are some requirements that we have determined for representation techniques for human–computer dialogue. They should have a sound formal basis, for validation and completeness. They should be supported by interactive tools, but also be manually producible, for those times when tools are unavailable. They should be complete for both the end-user's behavioural view and for the dialogue developer's constructional view. They should serve all phases of the dialogue development life cycle as a common, consistent technique.

Most well-known methods of language syntax representation are useful primarily for static programming languages. They are not powerful enough for expressing all concepts of programming languages (e.g. context sensitivity and semantics), not to mention representation of the visual and dynamic aspects of human–computer dialogue. Even for ordinary sequential dialogue, such representation methods must be augmented with other techniques. Two of the most popular representation techniques for programming languages have also been used to represent human–computer dialogue designs. These two techniques are Backus–Naur form (BNF) and state-transition diagrams. Both these techniques, however, are primarily means for representing grammatical relationships (e.g. logical sequencing of a command and its parameters) among end-user inputs. But neither BNF nor state diagrams show the process of how, for example, the command is solicited by the system (e.g. the apeparance of a menu or set of graphical icons on

a screen) and entered by the end-user (e.g. by typing a choice code or picking an icon) or how the system responds with semantic feedback (e.g. changes in the cursor during dragging. A comparison of these two techniques[11] has shown that state transition approaches provide more comprehensible language representations, because they show sequencing and surface structure of the human–computer interface more directly than BNF does. They are therefore a better cognitive match to the dialogue developer's mental model. Their main application, however, has been for representing designs for sequential, rather than asynchronous, human–computer dialogue.

What to represent in dialogue includes requirements and specifications, as well as behavioural and constructional details of both visible and non-visible aspects of dialogue. Some of these details include grammar of tokens in a sentence, intertoken syntactic constraints, appearance of all displays (e.g. position, cursor movement, clearing of screen, echoing, visual attributes), lexical rules for inputs, raw-to-normalized token mappings, data typing, declaration of dialogue variables and rules for defining conditional behaviour.

How to represent in dialogue was one of the major topics of the collaborative work with IBM. We have developed two techniques, each with state machines as the formal basis, for representing dialogues:

(a) *augmented scenarios* (see section entitled 'Behavioural domain: augmented scenarios'), to represent the behavioural view
(b) *augmented supervised flow diagrams* (see section entitled 'Constructional domain: augmented supervised flow diagrams'), to represent the constructional view.

Work on these representation techniques was a main thrust, and a major contribution, of the co-operative interaction between the Dialogue Management Project and IBM. A key idea of this research is to remove a dialogue developer from traditional programming approaches. Another key idea that has evolved is the dichotomy between the behavioural and the constructional domains of dialogue development.[12] The representation techniques and the separate development domains represent part of the Dialogue Management Project approach to producing non-programming 'languages' and techniques for use by a dialogue developer in designing and implementing dialogues.

When this co-operative interaction began, supervised flow diagrams (SFDs) had already been developed as part of the Dialogue Management Project methodological research. At that point, however, they were used mainly to represent the design of the global control and computational components of a system. During the co-operative work, SFDs were expanded and augmented so they can be used to represent the dialogue component of the system as well. Augmented SFDs represent a relatively complete representation of the dialogue. Scenarios were used rather loosely in the Dialogue Management Project methodological approach until they became a focus of this joint research. Augmented scenarios as a formal product of the representation process are a relatively new idea within this research. Both techniques, with examples, are detailed in the following sections.

A summary of representation techniques and roles that use them is shown in Figure 5. Augmented scenarios of the behavioural domain are used primarily by the end-user and the developer of the dialogue during the dialogue requirements, specification and design process. The dialogue developer, along with the applications programmer, uses augmented supervised flow diagrams of the constructional domain as detailed dialogue

	Behavioral Domain	Constructional Domain	Implementational Domain
User \ Technique	Augmented Scenarios	Augmented SFDs	Software
End-user	■		
Dialogue Developer	■	■	
Applications Programmer		■	■

Figure 5. Recording techniques and roles that use them

design and implementation takes place. Implementational software is primarily in the domain of the application programmer.

Behavioural domain: augmented scenarios

A *scenario* is a labelled visual transcript of exchanges between end-user and computer system. It includes an image of each screen, a label to identify each screen, and rules to define end-user inputs and screen transitions based on these inputs. It is a spatial representation that starts as a rather rough sketch of the screens and evolves, through iterative refinement, into precise final screen images. In fact, a set of scenarios is a prototype of the evolving dialogue.

Representation of dialogue using the *augmented scenario* technique includes a complete set of labelled screens, state diagrams to show screen transitions graphically, end-user requirements and specifications, a data dictionary, and revision history. Augmented scenarios are difficult to use to produce a complete representation of an interface design. A scenario gives a transcript of one specific path of the end-user through the system. To describe all possible paths can require very large numbers of screen images.

Scenarios: an example

Figure 6 is an example scenario of a fragment of sequential dialogue from the employee leave system transcript of Figure 4. Note the use of circled labels to identify each screen (upper right hand corner of each screen), and the use of circled labels to indicate to which screen a transition occurs upon human input. The circled '3' beside the 'Vacation' choice in screen 2 indicates that when the end-user types a '1' (the choice of vacation leave), screen 3 will appear. The circled '3+' in screen 3 indicates that end-user input at that point will keep the dialogue in screen 3.

The dialogue sequencing behaviour shown in this scenario can be represented graphically by a state-transition diagram (see explanation of Figure 8 below), as a technique to augment scenarios and as a bridge to constructional representation using supervised flow diagrams. The notation for state-transition diagrams is shown in Figure 7. A dialogue state is represented by a named circle, and two special state symbols exist, for start state and final (exit) state. A state transition, conditional on either end-

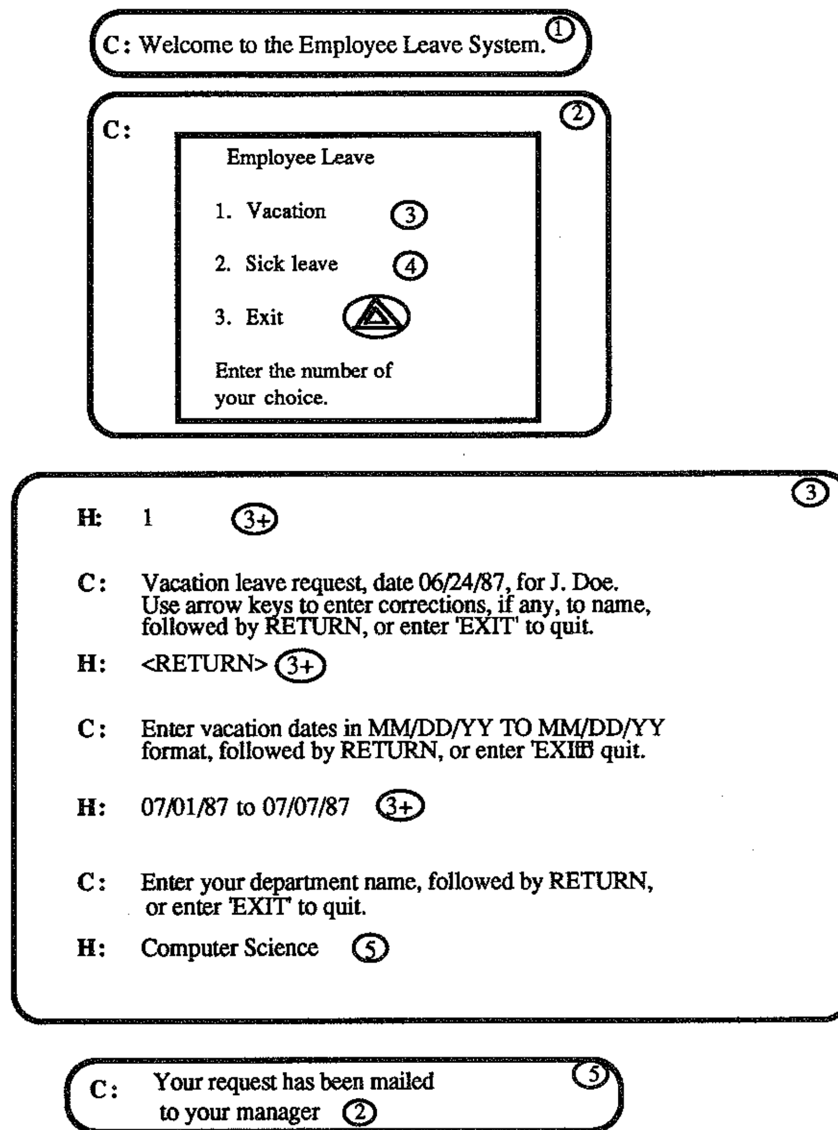


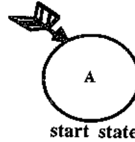
Figure 6. Example dialogue scenario for employee leave system

user input or an internal system event such as a timer, is indicated by predicates in angle brackets on arcs between states.

An example state diagram for the employee leave system is shown in Figure 8. Note that each circle (state) is a dialogue transaction, and that arcs between states correspond to the screen transitions due to end-user input in the scenario of Figure 6. State diagrams are thus a visual representation of sequencing behaviour indicated by circled labels in the scenario. There is not really any new information here; just information from the scenario given in a graphical, more general form. In fact, state diagrams can be generated automatically from labelled scenarios, with states being added as scenarios

- Dialogue state
  - - *Notation:*

  - - *Special state symbols:*



- State transition
  - - Conditional on end-user input
  - - *Notation:*

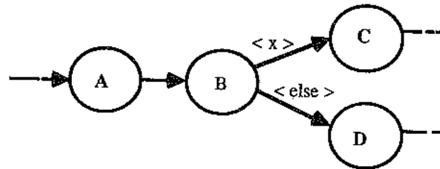


Figure 7. Notation for state diagram

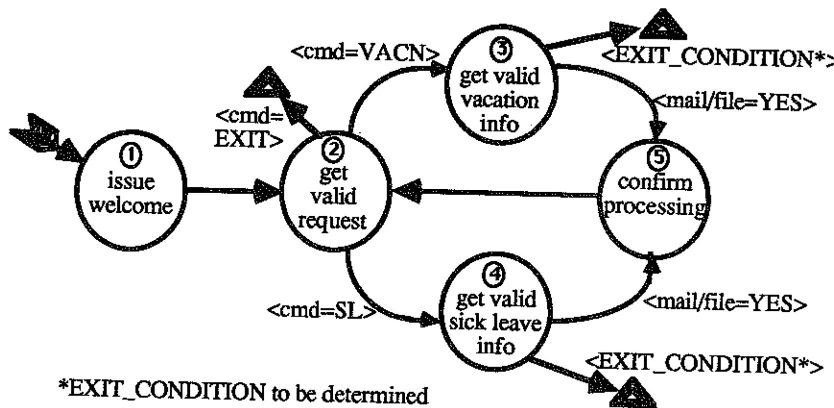


Figure 8. Example state diagram for employee leave system

become more complete during the development process. Note also that this representation is not lexically specific; that is, predicates on arcs, contained within angle brackets, indicate normalized, rather than raw end-user input, token values.

State diagrams can be considered as a graph, in the mathematical sense, and, as such, can have various abstraction processes applied to them. Many levels of abstraction are needed in the constructional design representation, since it is not possible or desirable to show all dialogue sequences in a single state diagram. Levels of abstraction,

used to control the amount of detail shown at different levels in the design representation, are based on objects of the dialogue transaction model described in the section entitled 'Modelling human–computer interaction'. An example abstraction for the state diagram of the employee leave system diagram of Figure 8 is shown in Figure 9. Considering each state as a dialogue transaction, the three transactions 'get valid request', 'get valid vacation info', and 'get valid sick leave info' from Figure 8 have been combined into a single 'get valid leave request' transaction in Figure 9. This combination of three transactions into one transaction is based on the developer's knowledge of the dialogue; it is left to the judgement of the developer to choose transactions that are logically related to form such abstractions.

Evaluation and state of development of scenarios

There is supporting evidence to show that scenarios are a viable technique for representation of dialogue, especially in the behavioural domain. They have a state machine basis, and are augmented with state diagrams to show control flow. The literature contains evidence of their long-term use, and our own observations[13] substantiate their usefulness. In fact, IBM practitioners have indicated the use of scenarios for dialogue development. The need for the kind of detail provided by scenarios in representing dialogue is evidenced by this statement from Gould and Lewis:[14] 'Another method is to construct detailed scenarios showing exactly how key tasks would be performed with the new system. It is extremely difficult for anybody, even its own designers, to understand an interface proposal, without this level of description.'

Scenarios have several advantages as a representation technique. Scenarios represent behavioural sequencing from an end-user's viewpoint. They are visual and concrete, as opposed to more cryptic notations (e.g. Backus–Naur form), and can be easily read by both customer and end-user. They provide a very early prototype of the dialogue, even in the absence of supporting prototyping tools. However, scenarios and their associated state diagrams do not have a disciplined, constrained technique for controlling abstraction. They also lack details, such as data flow, that are needed for the constructional view of the dialogue developer, in order to make the end-user's behavioural view happen. We have developed a technique called supervised flow diagrams (SFDs) to support the developer's constructional needs.

Constructional domain: augmented supervised flow diagrams

A *supervised flow diagram (SFD)* is a state-transition diagram that shows in a single representation: control flow, and data flow, states categorized by function and

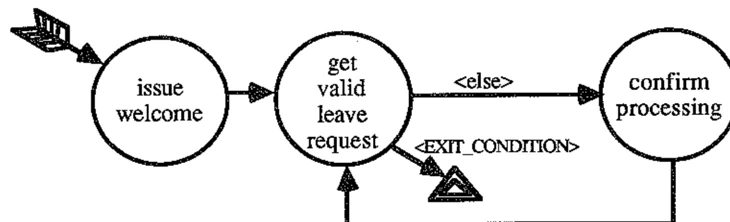


Figure 9. Example abstraction for state diagram of Figure 8

organization of levels of abstraction. We have developed a process by which SFDs are directly derivable by adding details to state diagrams of the augmented scenarios (see subsection entitled 'Supervised flow diagrams: an example' immediately below). Representation of dialogue using the *augmented supervised flow diagram* technique includes a complete set of SFDs, end-user requirements and specifications, a data dictionary, history for design decisions and revisions, test suites and a trace of which modules satisfy which requirements.

Supervised flow diagrams: an example

SFDs are produced using variations of the basic state symbol to indicate function (for different kinds of states) and level of abstraction. As such, SFDs are composed of symbols, shown in Figure 10, in which a circle represents a dialogue unit, a square represents a computation unit, and a circle inscribed inside a square represents a combination dialogue-computation unit that decomposes at lower levels of abstraction into circles and squares. Control flow in SFDs is indicated by the notation shown in Figure 11; these are essentially the same symbols used in the state diagram notation. In addition, a return symbol is required to return to higher levels of abstraction from 'calls' to lower levels.

Figure 12 shows an SFD for the employee leave system. Note that it is almost identical to the state diagram from Figure 9. In fact, it is derived directly from that diagram by adding 'black boxes' where appropriate for semantic (functional) computation. The black box serves as a place-holder for semantic computation of the system and is expanded by the application programmer during the development process.

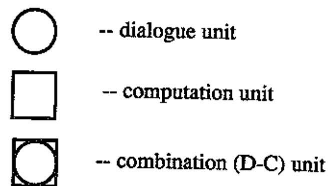


Figure 10. Symbols for functions/states in supervised flow diagram (SFD)

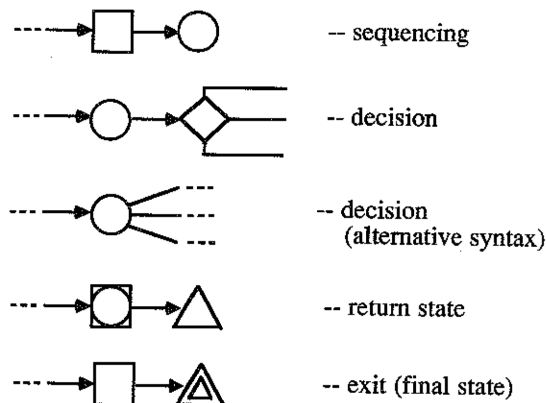


Figure 11. Symbols for control flow in supervised flow diagram (SFD)

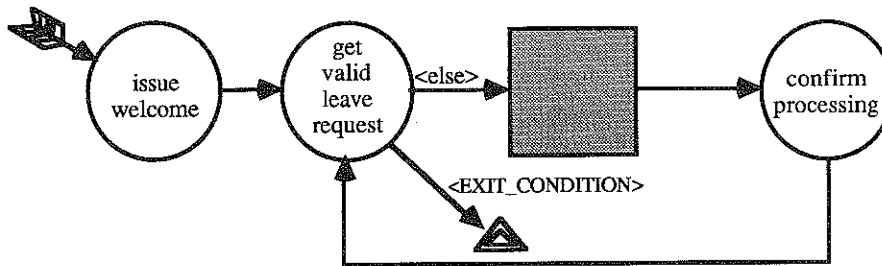


Figure 12. Supervised flow diagram for the employee leave system

The sequence of SFDs presented in the next several Figures represents a step-at-a-time development process, starting with the state transition diagram of Figure 9 and adding details and information (e.g. functionality of the black boxes, data flow that occurs).

Figure 13 is the SFD from Figure 12, with data flow added (parameter names shown going in and out of the circle and the box) and the computational black box 'cleared up' (its functional name, 'mail and file request', filled in). These are the kinds of details that are incorporated into the design as it evolves during the development process. Note that when data flow parameters are too long to fit easily, they can be abbreviated in the diagram and expanded below it (e.g. request_info).

Analysis of Figure 13 shows that the parameters date and emp_name flow into the 'get valid leave request' dialogue transaction, but there is no indication of where they came from. This can be a trigger that something is missing from the SFD; in this case, a computational unit to get values for the current date and employee name is missing. The SFD of Figure 13 is expanded to that shown in Figure 14, indicating a computation unit ('get date, emp_name from system') to obtain these values from the system.

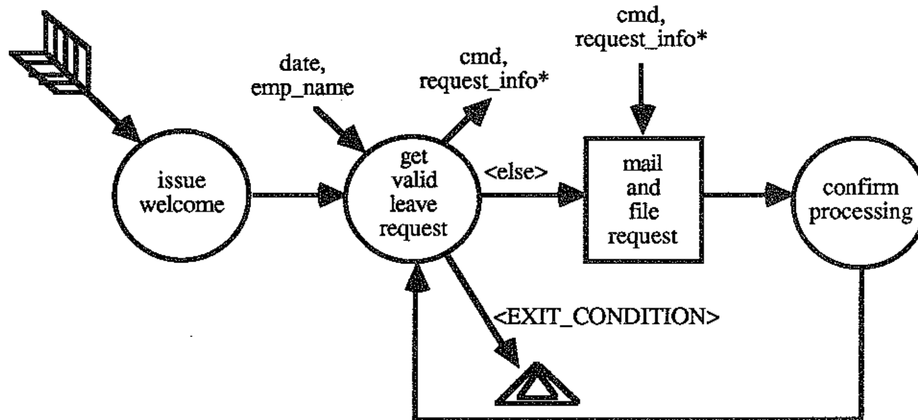


Figure 13. More complete supervised flow diagram for the employee leave system

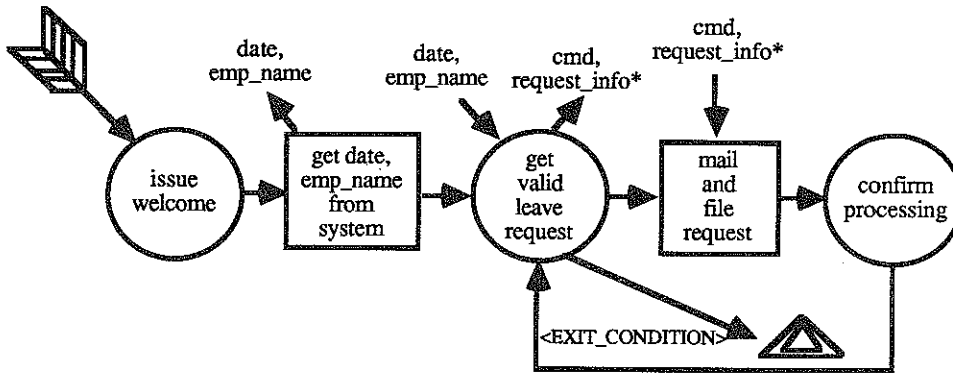


Figure 14. Expanding the supervised flow diagram for the employee leave system

From Figure 14, analysis shows that a cmd parameter is produced by the 'get valid leave request' transaction, but it does not appear to be used. This SFD can be further expanded, as shown in Figure 15, to make a distinction among end-user choices of the vacation command, the sick leave command, or the exit command, based on the value of cmd. This *expansion* of detail for clarity and completeness (from Figure 14 to 15) is the opposite of the *contraction* obtained by abstraction, as shown earlier (from Figure 8 to 9). Figure 15 also indicates how supervised flow diagrams got their name; the 'employee leave system' dialogue-computation unit at the top of the diagram is the

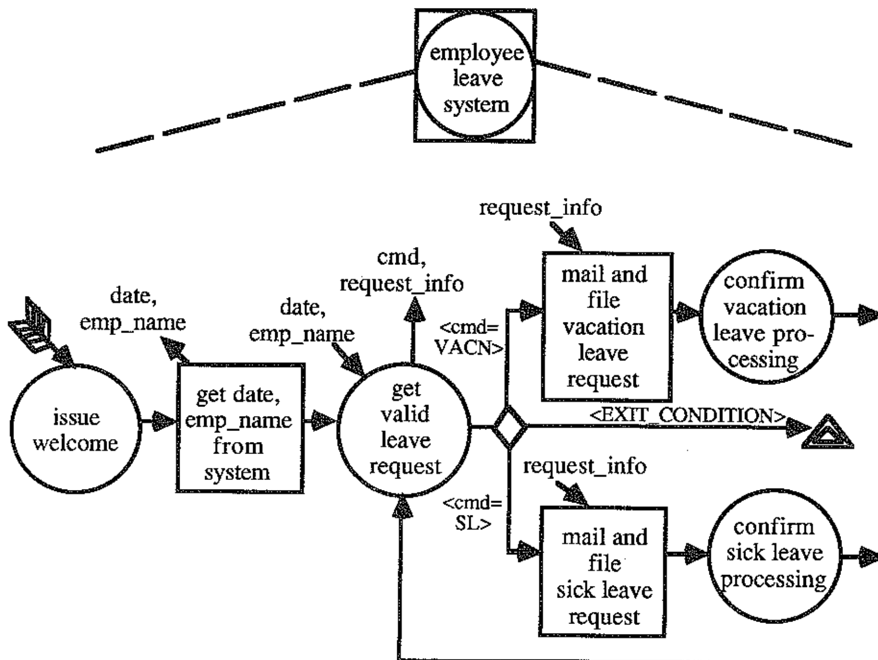


Figure 15. Further expanding the supervised flow diagram for the employee leave system

supervisor for the SFD shown under it, indicated by the dotted lines emanating from the dialogue-computation unit. All decision making, function (dialogue or computation) invocation, and parameter passing for data flow shown in an SFD are done within the supervisor of that SFD.

To relate symbols in Figure 15 to objects in the dialogue transaction model, note that each circle in this SFD is a dialogue transaction. The structure of the dialogue transaction model is used to develop and expand details of each of these dialogue transactions. For example, Figure 16 shows a detailed dialogue SFD that might evolve during the development process as a possible expansion for the 'get valid leave request' transaction (T) of Figure 15 into its interactions (I). This SFD belongs solely to the dialogue component of the system logic.

The 'get valid request' interaction (I) from Figure 16 represents an end-user input that happens to appear as a menu in the employee leave system scenario of Figure 6. Expansion of this interaction into its constituent objects during the development process might result in the detailed dialogue SFD shown in Figure 17. The interaction prompt (I.P) 'put prompt on screen' contains the format and appearance (a menu, in this design) of the prompt display screen. The interaction input (I.I) 'accept input' can be

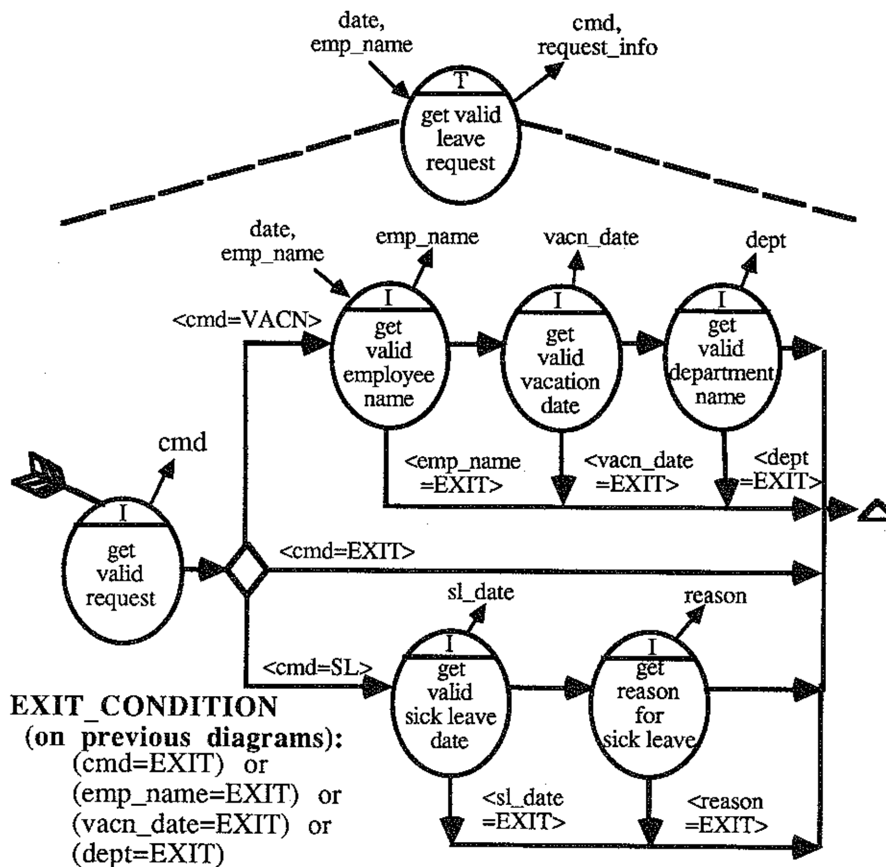


Figure 16. Expansion of 'get valid leave request' transaction into interactions

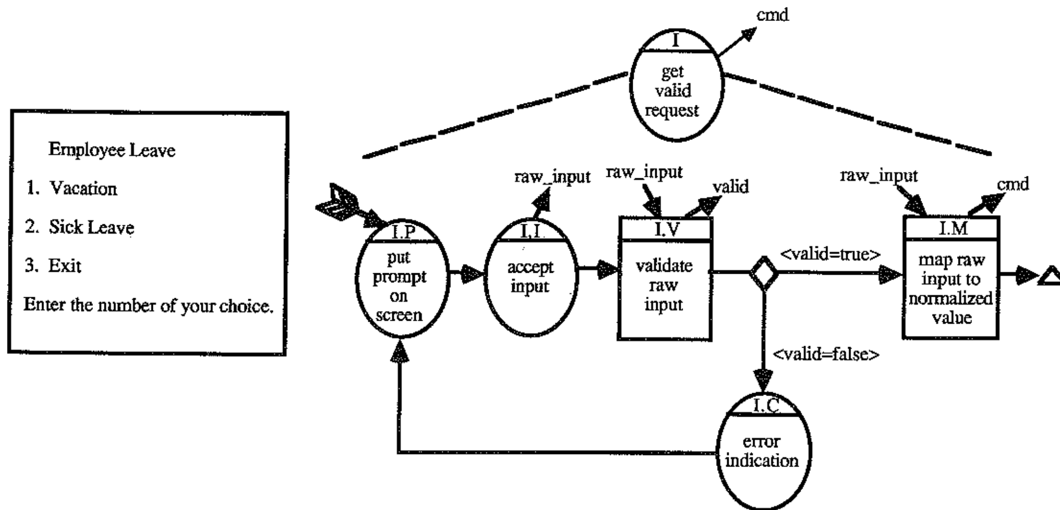


Figure 17. Menu from scenario of the employee leave system (Figure 6) and the corresponding expansion of the 'get valid request' interaction (that represents this menu) into its constituent objects

decomposed into an SFD to accept character at a time (lexeme) input from the end-user. The interaction validation (I.V) 'validate raw input' contains lexical validation rules for token input to interaction (I) 'get valid request'. The interaction mapping (I.M) 'map raw input to normalized value' shares with the interaction validation (I.V) the transformation table to map raw (device-dependent) lexical end-user input into normalized (device-independent) token values. Note that these are specialized dialogue computation objects (validation and mappings) in the dialogue transaction model, since they indicate non-semantic functionality that directly supports dialogue. The interaction confirmation (I.C) can contain an error message for each possible failure of end-user input against the validation criteria. Both the confirmation (I.C) and the prompt (I.P) can be conditionalized on end-user input errors.

Now let us suppose that human factors testing indicates the need for an alternative dialogue scenario, in which a form replaces the sequence of typed inputs in the employee leave system scenario of Figure 6. This form is shown in Figure 18. Navigation among fields in the form is accomplished using up and down arrow keys. Typing an 'X' in

Vacation June 24, 1987

Employee J. Doe

Vacation Dates

Department

EXIT w/o mail/file **MAIL/FILE**

Figure 18. Alternative fragment of dialogue scenario for the employee leave system

either the EXIT or the MAIL/FILE fields allows the end-user to leave the form. This scenario has the same requirements and same specifications as the previous one; however, its design is different.

One possible SFD expansion for this form-based version of the employee leave system is shown in Figure 19. The menu remains unchanged, and is the prompt display for the single transaction 'get valid request' to produce the cmd normalized token value. The screen image of the form is the prompt display for the single transaction 'get valid vacation info' that groups several interactions (see the explanation of Figure 20 below) together to produce normalized token values collectively called request_info. Each of these interactions is represented by a separate field in the form. Note that the dialogue developer made an arbitrary choice here to represent the menu and the form in these separate transactions, 'get valid request' and 'get valid vacation info', respectively.

One possible SFD expansion of the 'get valid vacation info' transaction (T) from Figure 19 into its five interactions (I) is shown in Figure 20. Note that this transaction has a transaction prompt display (T.P), 'put prompt on screen', that displays the visual image of the form. Each interaction ('employee', 'vacation dates', 'department', 'exit', and 'mail/file') represents a separate field in the form. 'IFN' represents 'interfield navigation' among fields of the form using the up and down arrow keys as explained above.

To continue downward expansion of the dialogue as it might develop, a possible SFD expansion of the single interaction (I) 'employee' from Figure 20 is shown in Figure 21. This Figure illustrates use of *dialogue primitives*, such as 'highlight this field' and 'lexically validate input'. These are basic dialogue building blocks, programmed and stored in a library for use by dialogue developers. Dialogue primitives are not decomposable, but are composable.

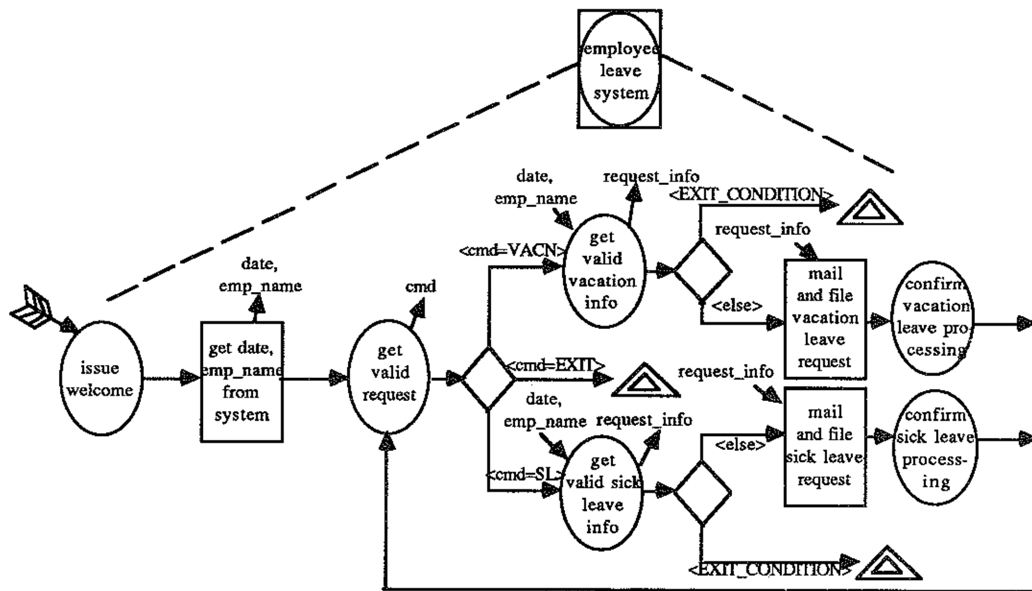


Figure 19. Alternative supervised flow diagram for the employee leave system, based on the form-filling dialogue scenario

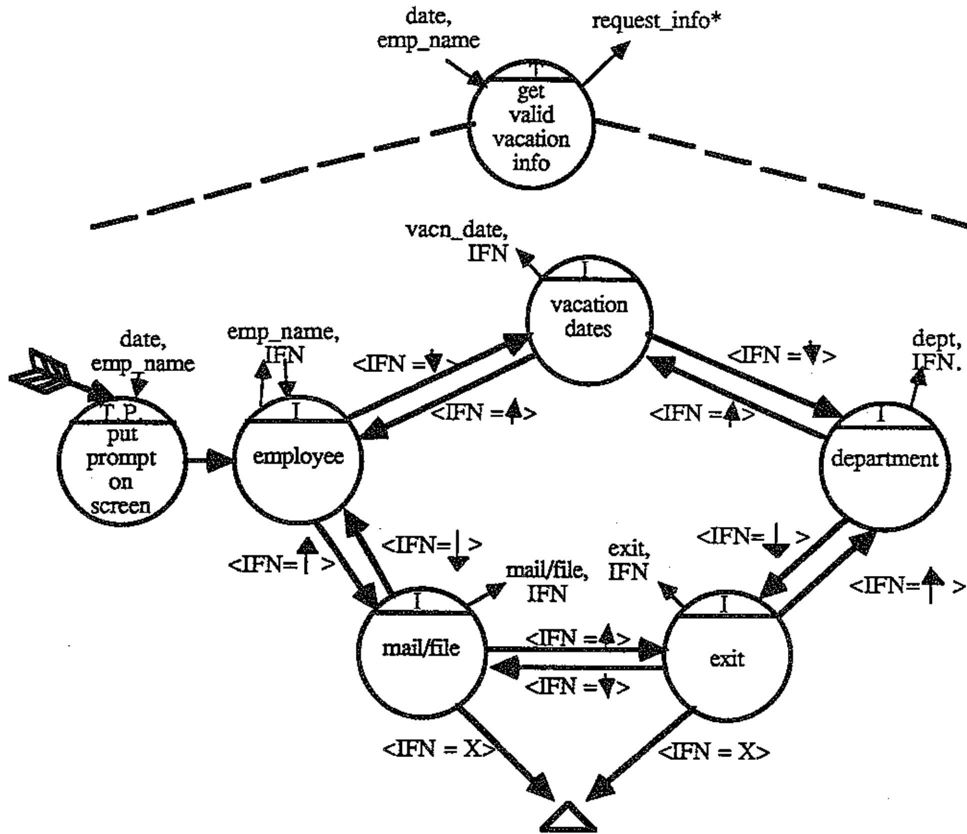


Figure 20. Detailed supervised flow diagram for 'get valid vacation info' transaction

It is important for overall context to realize that SFDs are used to represent both the global control and computational components, as well as the dialogue component. In the global control component, SFDs are the major determiner of target system sequencing, showing highest level control flow and data flow. An example of an SFD in the global control component was shown in Figure 19. In the computational component, SFDs show expansions of all functional computational units for the system. They decompose only into computational units (boxes) and map to source code at the lowest level. An example of an SFD in the computational component is shown in Figure 22, the expansion of the 'mail and file leave request' computational unit from the employee leave system.

Evaluation and state of development of supervised flow diagrams

As with scenarios in the behavioural domain, there is evidence to support usefulness of SFDs as a representation technique in the constructional domain. They are graphical and visual, readily producible by interactive tools. They map directly to executable source code that can be generated automatically, so that SFDs are themselves a high-

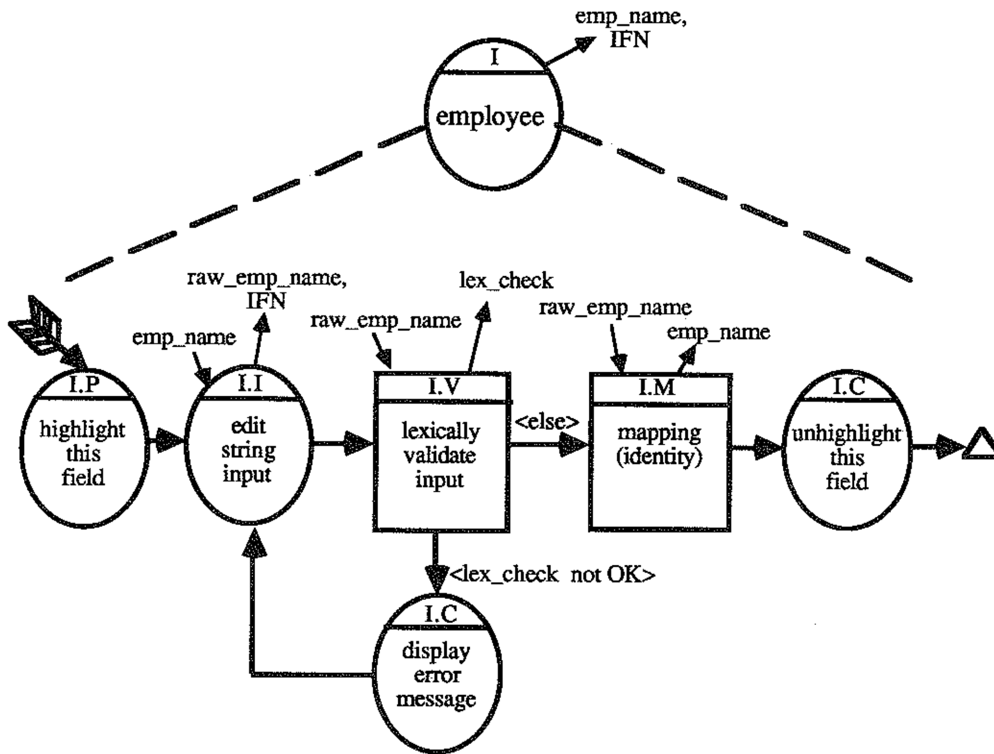


Figure 21. Supervised flow diagram expansion of the 'employee' interaction

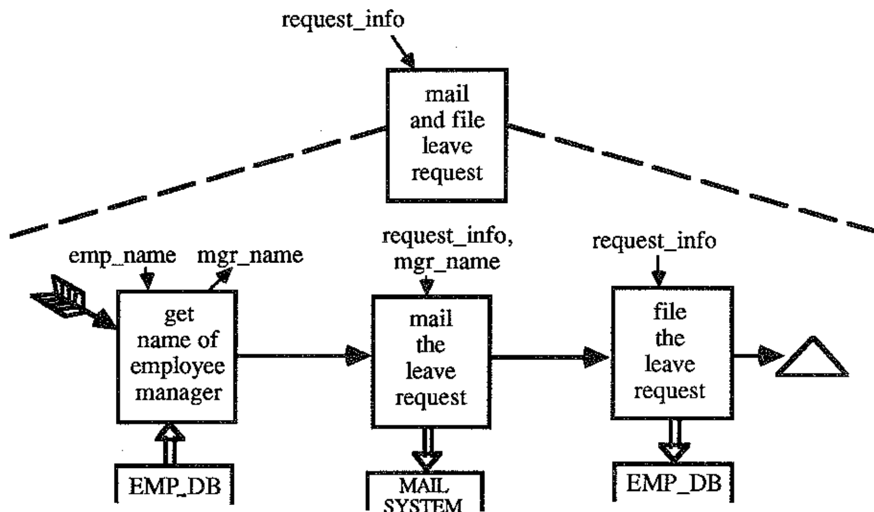


Figure 22. Example of a supervised flow diagram in the computational component

level graphical programming language. SFDs, as well as state-transition diagrams (STDs) and augmented transition networks (ATNs), are all formal, state-machine-based, graphical representations of control flow. Inputs cause transitions to new states and the current state is a representation of the history of previous inputs. The literature contains evidence that such graphical notations are favoured over textual notations as a representation technique. ATNs are a form of state diagram commonly used in the parsing of natural language. Subnets are used to match and recognize phrase patterns. The state diagrams of ATNs are usually supplemented with stack structures for dealing with non-determinism and look-ahead.

Although STDs have often been used to represent the control flow of user interface dialogue, SFDs have a number of advantages. First, SFDs embody both control flow and data flow in a single representation. Techniques using SFDs support the separation of the target system into dialogue, computation and global control components. The process of decomposition into levels of abstraction is structured and systematic. This decomposition can be done with STDs as well, but they offer no method for the process.

Our own results in long-term practical use also substantiate the usefulness of SFDs. They have been used over the past five years by more than 100 developers for creating about a dozen interactive systems. All developers have reported that SFDs are easy to produce, and particularly easy to read for an existing system, especially for making modifications.

Interactive dialogue development tools: automated support for representation techniques

Interactive tools for dialogue development are used by dialogue developers to produce human–computer dialogues. In particular, they interactively support representation of the dialogue as it is developed and they automate some or all of the implementation. The currently popular term used to refer to such tools is *user interface management system (UIMS)*.

Within the Dialogue Management Project, several experimental tools to support the dialogue representation and development process have been built. In an early version (AIDE—see the section entitled 'Dialogue independence'), built in 1983–1984, the approach was based on customizing tools for each specific interaction style. The result was a rather non-unified pot-pourri of graphics and text editors, menu and keypad formatters, and command parsers. Thus, only a few interaction styles and input techniques were possible in dialogue produced using this tool. However, AIDE did have a direct manipulation, non-programming interface for the dialogue developer, and was based on the dialogue transaction model to guide the developer. From these tool building experiences, we have learned that tool building is difficult and that tool builders also need meta-tools to help build dialogue development tools. We also learned that there are constraints imposed by the customized tools approach; there is limited flexibility and extensibility. Any new style necessitates a completely new tool, so it is difficult to be readily responsive to dialogue developers' changing needs.

In more recent tool building within the Dialogue Management Project, several goals have been established to help overcome the limitations experienced with the earlier tools. A unified approach will be achieved through use of the graphical programming language of the supervised flow diagrams. Escapes from the tools into the graphical

programming language and textual languages will also facilitate extensibility. Composability and reusability will be achieved by use of dialogue primitives, composed into SFDs to form both tools and target dialogues. Device and interaction style dependencies can also be encapsulated this way. A non-programming direct manipulation interface will be created for the dialogue developer, based on the dialogue transaction model as a guide for producing target dialogues.

ISSUES ADDRESSED IN THE CO-OPERATIVE WORK

During the two-year liaison, the DMS researchers developed and conducted a series of Research Workshops in which the Dialogue Management Project concepts—primarily the dialogue transaction model and the representation techniques for dialogue development—were presented to groups of IBM practitioners, both systems and software engineers. The aim was to extract and blend the strongest aspects of the Dialogue Management Project approach with those of IBM, to produce ideas for an integrated approach with emphasis on dialogue development while maintaining the functional strengths of the existing IBM methodology. The scope of the Workshops (and of the research in general) included research in the areas of modelling, methodology, representation techniques and interactive tools—essentially the material presented in this paper.

Main issues addressed

Key ideas from the Dialogue Management Project which led to its support by IBM were the following:

Dialogue development is not the same as human factors

Dialogue development is the process of creating, representing and testing an end-user dialogue. Human factors principles, when incorporated into the dialogue during the development process, make dialogue more acceptable and usable to the end-user of a system. Given these two definitions, once can conclude that they involve different goals across two strongly related disciplines.

A model of human–computer interaction

This includes a collection of well-organized and related definitions which allows a dialogue developer to capture the essence of an end-user dialogue in a standard vocabulary. This model is based on empirical observations and formal linguistic concepts. Use of this model could give a dialogue developer more confidence that a particular dialogue is 'complete', at least relative to the model.

Dialogue independence

Separation of the dialogue portion from the computational (functional) portion of a system appears to be a natural extension of the ideas of software engineering. This fits into the general model of software engineering which stresses data encapsulation and strongly associating functions with data into well-formed modules.

Role of a dialogue developer

Under the dialogue independence paradigm, a non-programmer using a non-conventional, non-programmer 'language' and tools could and should create the end-user dialogue of a system. This should specifically not be done by a traditional programmer using a traditional style of program development.

Questions raised and answers attempted

The following questions represent the basis of many discussions during our co-operative work. Answers to all these questions do not exist, but much has been learned about the problems and more specifically how to isolate the issues.

1. Can IBM, by incorporating the Dialogue Management Project ideas, establish a more complete system development model based on the strong paradigm which stresses separation of the dialogue portion of a system from the computational logic as a key architectural objective?

The general answer is yes. The Dialogue Management Project approach adds to the base of knowledge needed to create a more complete system development process but does not give a complete answer. The concept of dialogue independence appears to be a very powerful principle in system architecture; however, the implications of this idea at run-time (i.e. performance) are not yet clear, nor is its relationship to other technology developments, such as the Ada programming language. The advantage of separating the dialogue aspects of a system from its computational aspects is to reduce complexity of the system and potentially improve its maintainability. A consistent representation process would give IBM the benefit of a standard base to which human factors principles could be applied and evaluated. The challenge of blending this idea into real world projects, educating the systems and software engineering populations, and developing appropriate tools is real.

2. Can the mathematical ideas of functions, state machines, refinement, and correctness (i.e. verification) be applied to the dialogue development process?

Mathematics encourages organization and documentation of thinking so that others can understand. Developing dialogue software requires an understanding of the 'what' of the software and ensures that implementations meet specifications. Therefore, the same problem of correctness exists. Again, the separation idea improves the ability to study correctness by rigorously separating and isolating dialogue logic from computational logic.

Use of state machines to show screen transitions in a more rigorous manner appears very useful. In addition, use of state machines to model programs which encapsulate data appears needed since there seems to be dialogue state data owned by the dialogue component. Checking initial values from the end-user within the dialogue component is really performing an initial 'domain check' to see whether the value is within the domain of the function. Use of the domain idea in both the dialogue aspect and the computational aspect of a system reinforces and improves confidence in the system's ability to handle end-users' inputs as expected.

There is no reason to conclude that dialogue software has fundamentally different properties than other types of software. Therefore, dialogue software should exhibit the same mathematical properties as other types of software (e.g. computational software), and the same principles can and should apply.

3. Can a more complete and standard representation process be created to improve documentation of the developing dialogue?

A major problem in this area is the lack of standard representation mechanisms. Words such as 'scenario' are used loosely by both researchers and practitioners without formal definitions. It is a very difficult challenge to represent completely the end-user dialogue structure before the actual system is developed and coded in a final implementation language. However, representation techniques introduced early in the process reduce later problems significantly.

4. Can a 'language' be created which facilitates capturing of the end-user dialogue by non-programmers?

This is probably a question of definitions. Programmers are already getting away from classical programming languages. Similarly, the dialogue development process will evolve into easier-to-use languages and quality support tools. However, certain aspects of programming languages, such as sequencing and decision-making constructs, will always appear in some form since they are the basis of any process.

5. Can rapid prototyping of the end-user dialogue, based on the dialogue independence paradigm, be merged into the software engineering approach?

This can be done and is necessary. In large system development activities spanning many years, it is simply inadequate to have end-users wait until the system is delivered to 'see' and 'feel' the capabilities of the system. Statically written specification techniques are inadequate to express and understand the dynamics of an interactive end-user. Therefore, early rapid prototyping is critical.

6. Can a more formal dialogue development methodology be constructed and taught to the IBM systems and software engineering population?

This is possible, but it cannot be done in isolation of the other technologies.

Overall, our co-operative interaction demonstrated the value of university/industry relationships. The Dialogue Management Project contract provided a vehicle for opening discussions and sharing ideas. It is clear that different ideas, attitudes and vocabularies exist in university and industrial environments, and that there is much to be gained by the interchange. Although we found that the university researchers did much of the theoretical work, the industrial partner contributed through his knowledge of real-world software and systems engineering.

SUMMARY

The joint interaction of this research has added to both the Dialogue Management Project's and IBM's base of knowledge about systems, their characteristics, and their development process. The challenge is to more completely move these principles into actual day-to-day engineering of real systems. Our joint interaction is a step in this direction.

The main contributions of the Dialogue Management Project research include:

(a) a structural, descriptive model for human–computer interaction, and
(b) techniques for representing both the behavioural (end-user's) view and the constructional (developer's) view of the dialogue.

Emphasis on the co-operative research between the Dialogue Management Project and IBM has been on refining the dialogue. All agree that the co-operative effort was productive and synergistic, exploring ways to bring together academic researchers with industrial practitioners. Such liaisons will ultimately be a vehicle by which the technology transfer process is achieved.

ACKNOWLEDGEMENTS

The authors gladly recognize the contributions to this research by Dr. Roger W. Ehrich, Antonio Siochi and Eric Smith. We also wish to acknowlege funding support, under the supervision of Dr. H. E. Bamford, from the U.S. National Science Foundation; under the supervision of T. M. Kraly, from IBM Federal Systems Division; and the Virginia Center for Innovative Technology. Special thanks also to team members Pat Cooper, Sheila Casey, and especially Jo-Anne Lee Bogner for their cheerful and competent typing, graphics and layout support for this paper.

REFERENCES

1. H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer and R. E. Quinnan, 'The management of software engineering', *IBM Systems Journal,* **19**, (4), 414–477 (1980).
2. R. W. Ehrich and H. R. Hartson, 'DMS—an enviroment for dialogue management', *Proceedings of COMPCOM81*, Washington, D.C., 1981.
3. H. R. Hartson, D. (Johnson) Hix and R. W. Ehrich, 'A human–computer dialogue management system', *Proceedings of INTERACT '84, First IFIP Conference on Human–Computer Interaction*, London, 1984, pp. 57–61.
4. D. Hix and H. R. Hartson, 'An interactive environment for dialogue development: its design, use, and evaluation—or—is AIDE useful?' *Proceedings of CHI '86 Conference on Human Factors in Computing Systems*, Boston, MA, 1986, pp. 228–234.
5. D. Hix, 'Assessment of an interactive environment for developing human–computer interfaces', *Proceedings of Thirtieth Annual Human Factors Society Conference*, Dayton, OH, 1986, pp. 1349–1353.
6. H. G. Borufka, H. W. Kuhlmann and P. J. W. ten Hagen, 'Dialogue cells: a method for defining interactions', *IEEE Computer Graphics and Applications*, (July), 25–33 (1982).
7. I. Benbasat and I. Wand, 'A structured approach to designing human–computer dialogues', *International Journal of Man–Machine Studies,* **21**, 105–126 (1984).
8. J. D. Foley and V. L. Wallace, 'The art of natural graphic man–machine conversation', *Proc. IEEE,* **63**, (4), 462–471 (1974).
9. A. Siochi and H. R. Hartson, 'Task-oriented representation of asynchronous interfaces', *Proceedings of CHI'89 Conference on Human Factors in Computing Systems*, Austin, TX, May 1989, pp. 183–188.
10. H. R. Hartson and D. Hix, 'Human–computer interface development: concepts and systems for its management', *ACM Computing Surveys,* (March), 5–93 (1989).
11. R. J. K. Jacob, 'Using formal specifications in the design of the human–computer interface', *Communications of the ACM,* **26**, (4), 259–264 (1983).
12. H. R. Hartson, D. Hix and T. M. Kraly, 'An empirically-based methodology for human–computer interface development', *Proceedings of Second International Conference on Human–Computer Interaction*, Honolulu, HA, (August), 1987.
13. H. R. Hartson and D. Hix, 'Toward empirically-derived methodologies and tools for human-computer interface development', *International Journal of Man–Machine Studies,* **31**, 477–494 (1989).
14. J. D. Gould and C. Lewis, 'Designing for usability: key principles and what designers think', *Communications of the ACM,* **28**, (3), 300–311 (1985).