ヒューマンインタフェース 53−13
（１９９４．３．１１）

ＨＣＩにおけるダイアログを階層的に捉えた形式的記述の試み

松林弘治　辻野嘉宏　都倉信樹
大阪大学基礎工学部

ＨＣＩにおいて，ダイアログ (*Human-Computer Dialogue*) 分析は重要な要素の１つであるが，インタフェース設計者の指針となるモデルがほとんど存在しないのが現状である．本稿ではダイアログを，ユーザとシステムという内部状態を持った２主体の間で記号列を交換するというモデルで捉え，表示的意味論の概念に基づいてその構造と意味を形式的に記述することを提案する．また簡単なテキストエディタについて，ユーザによる入力記号列の構造と意味の記述を行い，階層的な構造など，ダイアログの特徴的な性質を明らかにする．

A FORMAL APPROACH TO HIERARCHICAL SPECIFICATION OF HUMAN-COMPUTER DIALOGUE USING DENOTATIONAL SEMANTICS

Kohji MATSUBAYASHI　Yoshihiro TSUJINO　Nobuki TOKURA
Faculty of Engineering Science, Osaka University
1-1 Machikaneyama, Toyonaka, Osaka 560, Japan

Human-Computer Dialogue is one of the most important area in the Human-Computer Interaction. However, there exists few formal dialogue models which are very handy and helpful for interface designers. In this paper, we propose the formal dialogue model in which the dialogue is a sequence of the exchange of symbols between one user and one system, both of which have internal entities. We also propose the formal description of the structure and content of the dialogue, based upon the denotational semantics.

We have tried to specify the meaning of input symbols of the user, expressing the characteristics of the dialogue such as the hierarchy of the dialogue. A simple text editor is introduced as an example of specifying the structure and content of the dialogue.

1 まえがき

コンピュータシステムは，処理速度の向上，入出力デバイスの進化，視覚的要素を多く取り入れたソフトウェア技術など，ソフトウェア，ハードウェアの両面において様々な技術が研究開発されてきている．その一方で，機能が増加したことで操作性が問題になったり，必ずしも専門家でない一般ユーザの利用増加によりその使用性が問題となったりしている．また，特定の用途，特定のユーザに限られない利用形態も増加しており，多様性を持ったシステムが望まれている．

これらの問題に対して工学に限らず様々な分野からアプローチを行っているのがＨＣＩ(*Human-Computer Interaction*) であり，インタフェース設計，ダイアログ設計もその中の重要な研究分野である．しかしながら，インタフェース／ダイアログ設計の方法論については，ハードウェア技術の進歩に比べるとほとんど進んでいない．これは，インタフェース設計に際して指針となるべきモデルがほとんど存在しないことが主な原因である[2]．そのため，実際の設計においては，今までに経験的によいと思われてきたものを設計者の判断で適用していくという方法をとるのがほとんどであった．

人間とコンピュータの間で行われているダイアログを形式的に記述することができれば，インタフェース／ダイアログ設計に援用することができるようになり，また，ダイアログの分析やインタフェースの評価などに活かすことが出来るようになると考える．今回我々は，人間とコンピュータとの間で行われるダイアログを，内部状態を持った２主体が相互に記号列を交換し合うというモデルで捉え，表示的意味論の概念を援用してその構造と意味を形式的に記述することを試みる．

2 ダイアログについて

2.1 インタフェースとダイアログ

人間がコンピュータを利用する場合，物理的には，キーボード／マウス／マイクなどの入力デバイスを用いて命令を送り，コンピュータはその結果をディスプレイ／スピーカなどの出力デバイスを通してユーザに伝えていると捉えられる．もしくは，ディスプレイ内に仮想的に表現されたボタン／メニュー／ウィンドウなどのオブジェクトをマウスやキーボードで操作することでやり取りをしていると見ることもできる．これらのように，人間とコンピュータとの間で物理的ないし仮想的に入出力の橋渡しをしている部分，両者に共有されている部分をインタフェース (*Human-Computer Interface*) と呼ぶ．

このインタフェースを通じて両者の間で行われるやり取りを人間同士の対話になぞらえて，ダイアログ (*Human-Computer Dialogue*) と呼ぶ．従来ＨＣＩの分野では，インタフェースとダイアログという２つの用語が混同して用いられることがあり，その指すところも人によって異なることが多かった[1]が，本稿ではダイアログを「ユーザとシステムとの間でインタフェースを通じて行われる情報伝達のプロセス」と捉え，両者を区別した上で議論を進めることにする．

2.2 形式化へのアプローチ

Norman [8] のユーザ活動の段階 (*stages of user activities*) は，ユーザとシステムとの間でのダイアログの流れをモデル化したものである（図１）．

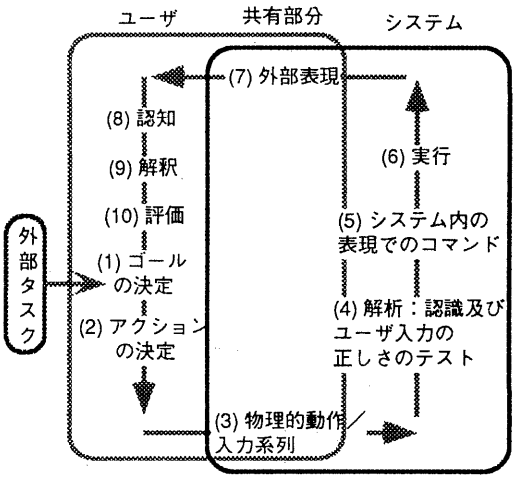


図１：Norman の “ユーザ活動のモデル”

図１を参考にダイアログに関連した形式化を考える場合，大きく分けて３つの形式化を考えることができる．

（a）ユーザのモデル化／特性分析
（b）システムの設計／分析，開発手法の研究
（c）人間とシステムの対話分析

GOMS モデル [5] や TAG [9] は（a）に相当し，(7) から (3) の間のユーザ側をモデル化したものであるが，(8) から (10) の間はブラックボックスとして扱われており，実際には (1) から (3) の箇所に対応しているといえる．

（b）に対応する，インタフェースの階層モデルやインタフェース生成用記述言語 [3] などは，(3) から (7) のシステム側をモデル化したものである．インタフェースを通じて受け取ったユーザの入力記号をシステムが分析する動作を記述したものであるということができる．

（c）は，(3) と (7) を中心にしてダイアログ全体を記述するという立場に立つもので，CLG [7] などは，(2) から (4) にあたる部分を形式的に記述したものであるということができる．

この3つのアプローチを比較すると，（c）が実際に両者の間で行き来する情報，すなわち対話言語 を軸にダイアログ全体を捉えるという包括的なアプローチであるといえる．しかしながら現時点では，入出力言語の記述という視点があるにも関わらず，システムの出力側の記述がまだ行われていないのが現状である [4]．

3 表示的意味論に基づいたダイアログの形式化の試み

3.1 ダイアログの形式的記述の目的

ＨＣＩにおいてダイアログは様々な方向から議論されているが，現時点においては次のような問題点がある．

(1) インタフェースやダイアログを議論する上で基盤となる包括的な理論的土台がないために，議論を有効にソフトウェア設計に活かすことができない
(2) ユーザの入力とシステムの出力は互いに関連し合っているものであるが，入力と出力の関係を形式的に記述されることは行われていない
(3) ダイアログを入出力を軸に捉えることの重要性は高まっているが，実際に出力まで含めた記述はほとんど行われていない

ダイアログの形式的記述に際しては，これらの問題を解決できるようなアプローチが望ましいと考える．そこで，ダイアログの入力と出力を組として捉え，かつユーザとシステムの内部状態も考慮した記述を，表示的意味論の概念に基づいて行うことを目標とする．

Scott & Strachey [10] らによる表示的意味論 (*Denotational Semantics*) は本来，プログラミング言語の意味記述のために生まれてきたものである．一般的には，(1) プログラムの構文の領域を抽象構文で定義する，(2) 意味を表す値の領域を定義する，(3) 構文の領域から意味の領域への関数を定義する，ことでプログラムの意味が記述される．これを用いて，ダイアログの構造と意味の記述を考える．

3.2 ダイアログの定義

ここで，ユーザとシステムとの間に行き交う情報を記号列であるとみなしてダイアログを形式的に捉えることにする．ダイアログＤを，ユーザの入力とシステムの出力が交互に行われるものであるとみると，次のように定義できる．

[定義] ユーザとシステムが与えられたとき，
両者の間のダイアログＤの集合 $\mathcal{D}$ は，

$$\mathcal{D} = \{\beta_0\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_n\beta_n \mid \alpha_i \in \Sigma^*,\ \beta_i \in \Gamma^*,\ \Sigma = \Sigma_s \cup \Sigma_c,\ \Gamma^* = \Gamma_s \cup \Gamma_c,\ \Sigma \cap \Gamma = \phi\}$$

である．ただし，

Σ_s ：ユーザがシステムに与える記号の集合
Σ_c ：ユーザからシステムに制御が移るきっかけとなるコントロール記号の集合
Γ_s ：システムがユーザに与える記号の集合
Γ_c ：システムからユーザに制御が移るきっかけとなるコントロール記号の集合

とする． □

この定義に見られる，両者の発話の列であるという構造は，コマンド入力に対する結果の出力，一文字入力に対するエコーバック，マウス移動に対するマウスカーソル移動など，至るところで見ることが出来る．もちろん，この$D \in \mathcal{D}$として記

述できるものの中には，相手の出す記号に対して脈略のない記号を返すといったような，一般的な意味での「対話」としてのダイアログでないものも含まれるが，ここでは「意味を持たない」ダイアログも含めて広義に定義している．

$\Sigma \cap \Gamma = \phi$ であるのは，ユーザとシステムのいづれが出した記号であるか，すなわち発話者を特定するためである．また，ユーザがａというキー入力を行い，それに対してシステムがａというエコーバックを返す場合，この両者は文字記号としては同じであっても，物理的にはキーの打鍵と画面への表示は全く別のものである．

リアルタイムで表示が進行するようなアプリケーションでは，ユーザ入力の有無に関わらずシステムの出力が進行している．この場合は，無入力という入力がユーザから行われた，すなわち α を ε とみなすこととする．

4　テキストエディタのダイアログの構造と意味の記述

4.1　テキストエディタの機能の概要

ここでは，ごく簡単な機能をもったテキストエディタを例にとり，そのダイアログの構造と意味について考えることにする．ここで念頭におくエディタはおおまかに以下の機能を持つことにする．

- 等幅の１バイト文字のみを編集対象とするテキストエディタとする
- ファイル名入力の際などを除き常にテキスト入力可能な状態にある
- 画面全体のうち最下行以外を編集対象表示領域とし，最下行はエラー表示／ファイル読込／保存のファイル名入力などに用いられる
- 画面の横幅よりも長い文字数をもつ行は画面上で折り返して表示される
- 全て挿入によって文字入力が行われる
- 一度に開くことのできるファイル数は１つだけである

このエディタの用意するコマンドとしては，１文字挿入，削除，カーソル移動，ファイル読み込み／書き出し，検索／置換，カット／ペーストなどを考える．

4.2　ダイアログの構造記述

本節では，このエディタにおけるダイアログの構造記述を試みる．構造は，抽象構文を用いて表現する．

ユーザの入力デバイスは，ここではキーボードのみを考え，キーの１ストロークが入力に関する終端記号と捉えることとする．

システムからの出力の終端記号が何であると捉えるかは難しいが，ここでは，１文字づつの出力，１ピクセル単位の出力と考えるのではなく，画面全体を１つの出力記号として捉えることにする．しかしながら，構文記述においては，画面全体の出力の特徴的な部分で代表させた方が分かりやすくなると考える．例えば，下に記した Iprompt という出力記号は，画面最下行に“ Input File Name: ”と表示される画面全体を表す，という風に，出力記号を定めることにする．

記述を見やすくするために，出力を表す記号には下線を引いて入力記号と区別する．入力記号においてシングルクォーテーションで括られた文字列は，１つのキー入力を表すものとする．四角で囲まれた文字列は，対応する１つのキーボードの入力を表す．また，[ctrl]-[R] というようにキー入力を横棒でつないだものは，２つのキーを同時に打鍵することを表す．出力記号におけるダブルクォーテーションは，その文字列の出力を表すものとする．

ユーザの入力記号：

char ：特殊記号を除いた全文字 を表す
Return ::= '[return]', Bs ::= '[backspace]',
Del ::= '[del]', Up ::= '[↑]', Down ::= '[↓]',
Left ::= '[←]', Right ::= '[→]',
InputFile ::= '[ctrl]-[I]',
OutputFile ::= '[ctrl]-[O]',
Search ::= '[ctrl]-[S]', Replace ::= '[ctrl]-[R]',
Redo ::= '[ctrl]-[D]', Mark ::= '[ctrl]-[M]',
Cut ::= '[ctrl]-[X]', Copy ::= '[ctrl]-[C]',
Paste ::= '[ctrl]-[V]', Quit ::= '[ctrl]-[Q]'

システムの出力記号：

Cmove ::= ('カーソル移動')*, Beep ::= 'Beep音',
Response ::= (char | Cmove | Beep)*,
Init ::= '画面消去', Iprompt ::= " Input File Name : ",
Oprompt ::= "Output File Name : ",
Sprompt ::= " String to Search : ",
Rprompt1 ::= " Replace from : ",

Rprompt2 ::= "Replace to :",
Qprompt ::= "Save before quitting? (y/n) :",

これら入出力記号を終端記号とすると，例えば1文字挿入コマンドに相当する箇所は

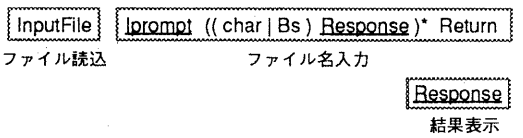
char Response

と表現されるし，ファイル読み込みコマンドは

InputFile Iprompt ((char | Bs) Response)* Return Response

のように記述される．この Dialogue は，先に定義したとおり，$\beta\ \alpha\ \beta \ldots \alpha\ \beta$ という並びになっている．

しかしながら，ユーザはそれらがよりまとまったレベルで入力を捉えていると考えることもできる．例えば，このファイル読み込みコマンドを，ファイル読み込みコマンド入力とその結果表示というやりとりとしてみると，



というまとまりとして捉えることができる．ここで，入力記号が2つ以上（ここでは2つ）並んでいるのは，ユーザがシステムに何らかの要求をする際，「何を」（「どのように」）「どうする」という情報を渡す必要があるからである．これは DOS のコマンドラインなどで「コマンド」「引数」と呼んでいるもの，もしくは「オペレーション」と「オブジェクト」と呼ばれるもの [1] である．ファイル読み込みコマンドの場合も「何という名前のファイルを」「カーソル位置に読み込む」という2つの情報をシステムに渡していることになる．

この際，ファイル名入力の部分は，先ほどのキー入力／エコーバックというレベルのダイアログから成り立っている．あるレベルでのダイアログの結果が，1つ上のレベルのダイアログもしくはべつのダイアログの入力記号として機能している，すなわち，ダイアログは階層的な構造をとっている，ということを表す一例である．

このファイル読み込みコマンドのダイアログの構造を2つのレベルに分けると，

InputFile <IFileName> Response
<IFileName> ::= Iprompt **Name** Return Response
Name ::= char Response | Bs Response

のように記述できる．ここで，<IFileName>のようにカギ括弧で囲まれた記号は，一つ下のレベルのダイアログの結果がこのレベルのダイアログの入力記号として機能していることを表すものとする．また，**Name**のように太字で書かれた記号は非終端記号を表す．これに基づいて，前節で記述したエディタのダイアログ全体の構造を階層的に捉えると，次のように書くことができる．

Dialogue ::= Init (char Response | Return Response
| Bs Response | Del Response | Up Response
| Down Response | Left Response
| Right Response
| InputFile <IFileName> Response
| OutputFile <OFileName> Response
| Search <SStr> Response
| Replace <RStr1> <RStr2> Response
| Redo Response | Mark Response
| Cut Response | Copy Response
| Paste Response)*
Quit <Yn> Response

このコマンドを単位としたダイアログにおいても，char や Copy など，もしくは InputFile <IFileName> や Replace <RStr1> <RStr2> などを1つの入力記号として考えると，$\beta\ \alpha\ \beta \ldots \alpha\ \beta$ という並びとして表現されている．そして，<IFileName>という一つ下のレベルのダイアログでも，上で書いたとおり，char, Bs, Return をそれぞれ入力記号とし，Iprompt, Response を出力記号として，$\beta\ \alpha\ \beta \ldots \alpha\ \beta$ という並びになっている．<OFileName>, <SStr> などの部分に関しても同様に階層的にダイアログを捉えることができる．

このように，ダイアログを階層的に捉えた場合でも，その各レベルにおいて，先に定義したとおり入力記号と出力記号が順に並んでいる様子が表現できる．

4.3 ダイアログの意味の表現

前節で，ダイアログの階層的な構造を表現した．本節ではそれに基づいて，表示的意味論に基づいてシステムの内部状態の定義や意味関数の記述などにより，ダイアログの意味の記述を試みる．

ここでは，ダイアログの意味を「ダイアログを

行う前と行ったあとで何がどのように変化したか」という面から捉えることにする．次の２つの対称的な変化が考えられる．

・ユーザから受け取った入力記号により，システムが内部状態を変化し，ユーザに出力記号を渡す（入力記号の意味）
・システムの出力記号を受け取ったユーザが内部状態を変化させ，次の入力記号をシステムに渡す（出力記号の意味）

この２つの変化が交互に起こっているのがダイアログであると考える．本稿では，前者の入力記号の意味について記述することを試みる．

4.3.1 内部実体の値の領域の定義

ここでは，システムの内部的な実体として，以下のものを考える．

・**Text**：編集可能なテキスト
・**Buf**：コピー／ペースト／検索／置換に必要なバッファ
・**File**：テキストが保管されたファイル（システム）
・**Disp**：ディスプレイ

内部実体は，ある値の領域，もしくは領域から領域への写像の組として記述される．例えば，**Text** は文字列であると捉えると，次のように定義することができる．

Text : (Natural → Byte)×Natural×Natural

この場合，最初の (Natural →Byte) は，文字列の何文字目に何の文字が格納されているか，を示す写像である．次の Natural はそれぞれ，カーソル位置，マーク位置を表す．

同様に，Buf, File, Disp などの内部実体に関しても，次のように定義できる．

File : Identifier → FileStorable
FileStorable = (Byte* + unused)

通常は Identifier は Byte* の部分領域である．File にファイル名を与えて，ファイルが存在すればその内容 Byte* を，存在しなければ unused が返ってくる．

Buf : (BufStorable + BufStorable + BufStorable)
BufStorable =(Byte* + unused)

Buf の中身は，順にコピー／検索／置換用のバッファである．Buf は，その内容が格納されていれば Byte* が，何も入っていなければ unused が入っている．

Disp : ((Natural × Natural)→Byte)×
(Natural×Natural)×Natural

Disp はシステムの外部表現である．最初の ((Natural ×Natural)→Byte) は，画面のどの位置に何という文字が表示されているかを表す写像で，次の(Natural×Natural) が画面上でのカーソル位置，最後の Natural が画面上での左上隅が Text での何文字目に相当するかを表す．

4.3.2 意味関数の領域定義と記述

このように定めたシステムの内部実体に基づいて，ダイアログの意味関数を記述する．まず，4.3節で説明したコマンド単位のダイアログについて，その入力記号の意味を決定する意味関数を定義する．入力記号の領域は Com とする．

execute : Com→(Buf→Text→File→
(Buf ×Text ×File))
display : Com→(Buf→Text→File→Disp→Disp)

execute は，入力記号の意味を内部実体の変化から捉えた意味関数である．ユーザの入力記号である Com の意味は，内部実体である Buf, Text, File から，次の内部実体への写像というかたちで表現される．

同様に ***display*** は，入力記号の意味を外部出力の変化から捉えた意味関数である．しかし，次の外部出力は，現在の外部出力だけでなく，現在の内部実体にも依存するので，Com の意味は上のように Buf, Text, File, Disp の４つから次の Disp への写像というかたちで表現される．

ここで，改行入力コマンドを例にとって考える．コマンド単位でのダイアログの構文記述のうち，改行コマンドに相当する部分は，

Return Response

である．この際の入力記号は Return であり，出力記号（外部表現）は Response である．このコマンドについて ***execute*** と ***display*** を以下のように記述することができる．

execute [Return] $b\ t\ f$ = (b, insert-char (t, Return), f)
（入力記号が Return の場合，Buf と File はそのままで

Text を insert-char を施したものに変更する)
display [Return] $b\ tf$ =
let (b', t', f') = ***execute*** [Return] **in**
let d' = clear-message d **in**
let $(table, (x, y), z) = d'$ **in**
if y=maxheight-1 **then** scroll-up-disp (t', d')
else redraw-disp (t', d')
(入力記号が Return の場合，まずメッセージ行をクリアして，さらにカーソルが編集対象の最下行にあれば1行スクロールし，改行記号が挿入された結果を表示する)

insert-char や clear-message などは，意味関数の記述を読みやすくするために別に定義した補助関数である．例えば，カーソル位置に与えられた文字を挿入する ***insert-char*** という補助関数は次のように定義されている．

insert-char : Text × Byte → Text
insert-char $(text, byte)$ =
let $(list, n, m)$ = text **in**
let $list' = \lambda n'.$ (**if** $n'<n$ **then** $list\ n'$ **else**
if $n'=n$ **then** $byte$ **else** $list\ (n'\text{-}1)$
in $(list', n+1,$ **if** $m<n$ **then** m **else** $m+1)$

このように各コマンドに対応して***execute***と***display***の記述を行えばよい．

4.3.3 階層的な構造に対応した意味記述

階層的な構造を持つダイアログに関しては，新たに別の意味関数を定義する必要がある．ここでは，4.3節で扱ったファイル読込コマンドを再び例として考える．

コマンドを単位としたダイアログの構文記述のうち，ファイル読込コマンドは

InputFile <IFileName> Response

という箇所に相当する．この記述においては，ファイル名入力の際のキー入力の1つ1つは明示されていないし，キー入力列の結果，つまり<IFileName>という中のダイアログの結果としてファイル名を受け取るだけである．しかしながら，編集を許すとすれば，ファイル名入力のためのバッファが必要である．そこで，ファイル名入力用に必要なバッファは，<IFileName>を更に細かく見たレベルでのキー入力／エコーバックというダイアログのみに関与する内部実体であると考えれば良い．表示的意味論においては，異なる構文領域に異なる意味関数を定義するのが一般的であり，ここでも，1文字のキー入力の意味と，1つのコマンド入力の意味は異なるので，先の ***execute*** / ***display*** とは別の意味関数を定めることにする．

ここで，ユーザの入力記号の領域を LCom と定める．この場合，

LCom ::= char | Bs | Return

となる．また，CmdBuf = Byte* と考える．

keyin : LCom → (CmdBuf → CmdBuf)
echo : LCom → (Disp → Disp)

例えば，***keyin*** に関しては，***execute*** と同様に以下のように記述できる．

keyin [char] cb = input-cmdbuf $(cb,$ char)
一文字入力 (cb の最後に $char$ を加えたものを新しい cb とする)
keyin [Bs] cb = bs-cmdbuf cb
バックスペースキー入力 (cb の最後の1文字を削除したものを新しい cb とする)
keyin [Return] cb = cb
リターンキー入力 (cb は変化しない)

さらに，意味関数 ***keyin*** とは別に，評価関数 ***eval*** を定義する．

eval : Input → Byte*
Input ::= <IFileName> | **Name**

この2つを導入することで，以下のとおり <IFileName> というダイアログの結果が ***eval*** [<IFileName>] として表現できる．

eval [Iprompt **N** Return Response] =
keyin [Return] ***eval*** [**N**]
eval [**N** char] = ***keyin*** [char] ***eval*** [**N**]
eval [char] = ***keyin*** [char] clear-cmdbuf
eval [**N** Bs] = ***keyin*** [Bs] ***eval*** [**N**]
eval [Bs] = ***keyin*** [Bs] clear-cmdbuf

これで，コマンド単位のダイアログにおいて ***execute*** [InputFile <IFileName>] を

execute [InputFile <IFilename>] $b\ tf$ =
let ft = read-file (***eval*** [<IFileName>])
in $(b,$ insert-str $(t, ft), f)$
(<IFileName> のダイアログの結果をファイル名としてそのファイルを読み込み，カーソル位置に

挿入したものを新しい Text とする)

とすれば，コマンド単位のダイアログでは <IFileName>の中のダイアログを見ることなく，<IFileName>の中で行われている別のレベルのダイアログの結果のみを受けることができる．このようにして，階層的な構造を持つダイアログに関しても意味の記述を行うことができる．

5 おわりに

本稿では，ＨＣＩの視点からヒューマン・コンピュータ・ダイアログを形式的に記述することを試みた．ダイアログを，ユーザとシステムという内部状態を持った２主体が互いに記号をやり取りしているものと捉え，その意味を，プログラム言語の構文と意味記述に用いられる形式言語と表示的意味論の手法を用いて記述することを提案した．また，実際の記述例として，キャラクタベースのインタフェースを持つテキストエディタを例にとり，文字単位，コマンド単位での入出力記号列の構造と意味の記述を試み，入力記号列に関して，コンパクトに見やすく書くことができた[6]．

ダイアログにおいては，その記号列のやり取りの構造や意味よりも，文脈に即した意味や時間的要素といった，より実際的な側面による影響が大きいことが知られている．今回行った構造と意味の記述に加えて，実際的側面を加味した記述を考える必要がある．

また，今回は入力→出力の方向のダイアログのみの記述を行ったが，これと対称的に，システムの出力記号によりユーザの内部状態が変化し，次の入力記号が出される，という出力記号の意味からみたダイアログの記述も行う必要がある．そのためには，ユーザの内部状態をどのように設定するかが最も重要な問題となる．

参考文献

[1] Booth, P.A.: "An Introduction to Human-Computer Interaction ", Lawrence Erlbaum Associates, Hillsdale, NJ. (1989).

[2] チグネル，M.H.，ハンコック，P.A.，ローエンサル，A.：*"知的インタフェース - 人とマシンの知的相互作用 - "*，辻 敏夫 訳，海文堂，pp.1-26 (1991).

[3] Hoppe, H.U., Tauber, M,J, & Ziegler, J.E.: *"A Survey of Models and Formal Description Methods in HCI with Example Applications "*, ESPRIT Project HUFIT, Report B3.2a. (1986).

[4] Hoppe, H.U.: "A Grammer-Based Approach to Unifying Task-Oriented And System Oriented Interface Descriptions", *Mental Models and Human-Computer Interaction 1*, pp.353-373. (1990).

[5] Kieras, D.E. & Polson, P.: "An approach to the formal analysis of user complexity", *International Journal of Man-Machine Studies*, 22, pp. 365-394. (1985).

[6] 松林 弘治：“ヒューマン・インタフェース・ダイアログの形式的記述法について”，平成５年度大阪大学大学院基礎工学研究科修士学位論文 (1994-02).

[7] Moran, T.P.: "The Command Language Grammer: A representation for the user interface of interactive cimputer systems", *International Journal of Man-Machine Studies*, 15, pp.3-50. (1981).

[8] Norman, D.A.: "Some observations on mental models", *Mental Models* , Lawrence Erlbaum Associates, Hillsdale, NJ. (1983).

[9] Payne, S.J. & Green, T.R.G. Task-Action Grammers - A model of the mental representation of task languages. *Human-Computer Interaction*, Vol.2, pp.93-133. (1983).

[10] Scott, D. & Strachey, C. Towards a Mathematical Semantics for Computer Languages. Proceedings of Symposium on Computers and Automata. Polytechnic Institute of Brooklyn Press, New York, U.S.A. pp.19-46. (1971).