予測 / 例示インタフェースシステムの研究

増井 俊之

目次

図目次

表目次

第 1 章　序言

1.1　研究の目的と概要

ハードウェアやソフトウェアの進歩により、複雑な計算や作業を行なうことのできる計算機が急速に普及しつつあるが、自動車や電話のように長い歴史をもち完全に生活の一部となっている機械に比べると計算機は複雑で使いにくい面が多く、まだまだ誰もが簡単に使える状況からほど遠いのが現状となっている。計算機を簡単に使うための様々な手法について、ヒューマンコンピュータインタラクション (HCI) やコンピュータヒューマンインタフェース (CHI) などと呼ばれる分野で幅広い研究が行なわれている。

本来、計算機は単純な作業を人間のかわりに効率良く実行してくれるものと考えられているが、計算機が大きな処理能力を持ち複雑な処理を行なわせることが可能になったにもかかわらず、一般に普及している計算機アプリケーションにおいては単純な繰り返し作業を人間が行なわなければならないような状況が数多く見うけられている。例えば以下のような場合、機械的な仕事であるにもかかわらず人間が手作業で処理を行なっている状況が多いと思われる。

- ワープロの文章中の何行かを 2 文字ずつ字下げしたい

  10 行ぐらいの操作であれば「行頭へ移動」「空白をふたつ入力」「次の行へ移動」といった基本編集操作を手で繰り返しても大きな手間ではないが、 100 行、 1000 行ともなればなんらかの方法で自動実行させたくなる。そのためには、このような機能をもつ特別なエディタやプログラムを使用したり、プログラムを特別に作成して処理しなければならないかもしれない。

- 前回と同じ条件で再コンパイル / リンクを行ないたい

  条件の指定が面倒だったり沢山のファイルを指定する必要があったりする場合、以前と同じ条件でのコンパイル / リンクの再実行も容易ではない。プロジェクト管理ツールを使用すればよいのだが、一度きりの再実行にそのようなツールを使用するのはかえって非効率かもしれない。

- 値を変えて前の計算を再実行したい

  電卓などで各種の計算を実行した後で、値を変えて同じ計算を実行したいことがある。プログラミング電卓を使ってわざわざプログラムを作ることなく小さな手間で再計算を実行したい。

- ブロックダイアグラムを美しく描きたい

  複雑なブロックダイアグラムを図示しようとする場合、図形エディタを使って人間が全てを自分で配置しなければならないのが普通である。図形を美しく配置を行なうための各種の自動配置アルゴリズムが提案されているが、自動配置の結果は必ずしも満足いくものとは限らないため、最終的にはユーザの好みにもとづいた手直しが必要となる。手直しを繰り返しているうちに、自動配置プログラムがユーザの好みを反映するように学習してくれるとありがたい。

このような例は、現在の計算機アプリケーションの多くにおいて、非定型的な繰返し作業を楽に効率良く行なうことがむずかしいことを示している。計算機を頻繁に使用するユーザにとっても常に例外的な処理が発生するし、そうでないユーザにとってはあらゆる処理が例外的処理となるため、このような繰返し作業はあらゆるレベルのユーザにとってわずらわしい問題となっている。このような場合、繰返し作業にかかる時間も問題であるが、つまらない作業を強いられることが苦痛である点において問題が大きいと考えられる。

1.2 予測 / 例示インタフェースによる問題の解決

多くのユーザがいつも全く同じ作業を行なうのであれば、その作業を手順化して登録しておいたり、アプリケーションプログラムの基本機能として用意しておけばよいが、特殊な繰返し作業はごくたまにしか必要とならないため、それぞれのケースへの対応を最初からシステムで用意しておくことは良い選択ではないし、全ての特殊な場合に対応することは不可能である。しかし現実のパーソナルコンピュータなどのアプリケーションでは、幅広い要求に応えるために沢山の機能をあらかじめ用意しておくことがよく行なわれており、無用に肥大化したアプリケーションが数多くみられる。このような、一般的でない機械的な繰り返し作業を効率よく実行するには、あらゆる場合を想定してシステムに用意しておくよりも、場合に応じてシステムにユーザの次の操作を予測させたり、以前の実行結果を例として再利用することがより有効であると考えられる。

予測を行なうための情報としては、操作に関する一般的知識や、ユーザの以前の操作履歴などを使用することができる。例えば UNIX のコマンド行インタフェース (シェル) では、コマンド名辞書を用いることによって、ユーザが入力した最初の数文字からコマンド名全体を予測したり、操作履歴を覚えておくことによって、以前発行したコマンドと同じコマンドを簡単に再発行することができるようになっているものがある。またワードプロセッサの仮名漢字変換システムでは、日本語文章に関する知識を用いることにより変換される漢字を予測したり、以前のユーザの変換結果を利用して次回の変換を予測したりできるものが多い。以前の操作と似た操作を何度も繰り返すことは多いため、このような予測手法は単純で使用が簡単であるにもかかわらず効果が大きい。

このような単純な予測インタフェースシステムは使用が容易かつ有用でもあるが、ユーザの意図をくみとって動作しているわけではないし、複雑な操作を予測することはできないので、有用さには限界がある。ある程度複雑な予測を行なわせるためには、何らかの方法でユーザの意図をシステムに伝える必要がある。

ユーザが繰り返し操作の意図を充分認識している場合は、繰り返される操作をユーザが自分で定義したり呼び出したりすることにより繰り返し操作を効率化することができる。多くのワープロやエディタにはマクロ定義機能が装備されており、複雑な操作列をユーザが自分で定義して使用することができるようになっている。例えば、長い操作列を別の単純な操作で置き換える「キーボードマクロ」のような単純なマクロ定義機構が広く使われている。また、複雑なマクロ定義はプログラムを作成するのと同じであり、プログラマでない一般のユーザには難しいのが普通なので、システムに対して操作の例を指示することにより、プログラミングを行なうことなくシステムに自動的にマクロを生成させるような「例示インタフェース (Demonstrational Interface) システム」も多数提案されている。このようなシステムを使用すると、操作の具体的な実例をユーザがシステムに示すことによって自分の意図を伝えることができる。これはプログラムのような抽象的記述の使用に比べると誰にでもわかりやすいという特長がある。

簡略化したい操作手順についてユーザがあらかじめよく認識している場合は上記のような手法は有効であるが、問題点も多い。まず、定義すべき操作についてユーザがあらかじめ理解して

おく必要があるが、操作を実際に繰り返す前にその操作について理解することは難しいし、意図が明確でないことも多い。また、マクロ定義のために必要となる余分な操作がわずらわしいという問題点もある。特に、例示によりプログラムを作成するような場合、システムが間違った推論を行なう可能性があるため、正しい推論を行なわせようとすると必ずしも定義のための手間が減るとは限らない。

既存システムにおける以上のような問題点を考慮すると、繰り返しのような操作の意図を効率よく計算機に伝えるためには、以下のような特徴をもつ予測 / 例示インタフェースシステムが有効であると考えられる。

- 例からシステムがプログラムを自動的に作成する
- ユーザの暗黙的意図が自動的にくみとられる
- システムはユーザの指示によりシステム予測や推論を開始する
- システムの予測結果は単純な方法で実行される
- ヒューリスティクスを多用せず単純で信頼性の高い予測手法を使う
- 操作取り消しなどの機構により予測の実行によるリスクが回避される

これをまとめると、**信頼性の高い単純な手法によりユーザの操作履歴などからユーザの暗黙的な意図をプログラムとして自動的に抽出し、単純な操作で再利用できるシステム**ということになる。

数多くの予測 / 例示インタフェースシステムが提案されてきたにもかかわらず、このような条件をすべて満たしているものは稀であり、予測などを行なわない通常のインタフェースに対し競争力のあるものはほとんど無かった。本論文では、上記のような考えに基づいて既存のシステムにおける問題点を克服した新しく実用的な予測 / 例示インタフェースシステムを 3 種類提案し、それらの実装と評価について述べる。また、これらを統合して実現し、かつ広い範囲のユーザインタフェースソフトウェアを容易に構築するためのユーザインタフェースアーキテクチャを提案する。従来の予測 / 例示インタフェース手法では冒頭のような問題を簡単に解決することができなかったが、本論文の手法を用いることにより、あらゆる熟練レベルのユーザにとって問題となる 各種の繰り返し作業を大幅に簡単化することができる。実際の計算機操作においては、繰返し作業がほとんどの部分を占めているわけではないため、本論文で提案する手法による繰返し作業の効率化により作業時間全体が大幅に短縮されるわけではないが、ユーザが単調作業から解放されることによる心理的効果は大きい。

1.3 本論文の構成

本論文の構成は以下のとおりである。

まず第 2章において本研究の背景について述べ、現在までに提案されている各種の予測 / 例示インタフェースについて解説しその特徴について述べる。

第 3章において、既存の予測 / 例示インタフェースの問題点を整理し、それらを克服する新しい予測 / 例示インタフェースシステムに必要な条件について考察を行なう。

続く 3 つの章において、 3章の考察にもとづいて問題点を克服した予測 / 例示インタフェースシステムを 3 種類提案し、それぞれの評価を行なう。

第 4章では、操作履歴中から繰り返し操作を自動抽出して再利用することのできるシステム “Dynamic Macro” を提案し、その評価や応用などについて述べる。

第 5章では、操作履歴情報から操作間の依存関係を自動抽出し再利用をうながすシステム “Smart Make” を提案し、その評価や応用などについて述べる。

第 6章では、ユーザの好みを例からプログラムの形として抽出し後で利用する手法について述べ、その実例として、以前の配置例をもとにして図形の配置の評価関数を自動抽出するシステムについて述べる。

第 7章では、以前の章で提案した予測 / 例示インタフェースシステムを統合して実現するためのユーザインタフェースアーキテクチャについて述べ、各種の予測 / 例示手法を統合して実現した環境について述べる。

予測 / 例示インタフェースの手法は、単純な繰返し作業の効率化に有効であるだけでなく、既に存在しているデータを例として活用することにより、広い範囲の新しいインタフェース手法に適用することができる。第 8章では、予測 / 例示インタフェース手法を適用した新しいインタフェースの例を示し、今後の展望について述べる。

第 2 章　研究の背景

2.1　はじめに

本章では、誰もが機械を簡単に使用できるようにするための手法として、システムにユーザの操作を予測させたりシステムへの例示により操作手順を指示したりする予測 / 例示インタフェースの考え方が有効であることを示し、現在までに提案されている各種の予測 / 例示インタフェースシステムの概要について述べる。

2.2　使いやすい機械とは

2.2.1　使いやすさと操作の量

あらゆる社会生活において機械の使用が必須となっている現在、人間にとって使いやすい機械を作る各種の技術が非常に重要となっており、特に複雑な機械である計算機と人間の間のやりとりを円滑に行なうためのユーザインタフェースに関する各種の研究が行なわれている。

機械を使いやすくするためには多くの要素が関連している。計算機の場合、その大きさ / 重さ / 形 / 色や入出力装置の種類 / 構造 / 方式がもちろん問題になるし、同じ装置を使う場合でも、入力に対して計算機がどのように反応するかやどのように出力を提示するかなど、多くの要素が使い勝手を左右すると考えられる。

一方、機械を使いやすくするための重要な要素として、「必要な処理を少ない操作で実行できること」があるであろう。同じ装置を使って同じ結果が得られるのであれば、なるべく少ない操作で処理を指示できることが望ましい。

コマンドをキーボードから入力することにより計算機への指示を行なうシステムでは、操作量を減らすためにコマンドの名前をなるべく短くする工夫をしているものがある。UNIX オペレーティングシステムはそのような工夫をしたシステムの一例である。UNIX では例えばファイルのリストを得るためには “**ls**” という短い名前のコマンドを使用するが、これは例えば “**list-the-name-of-files**” などと入力するのに比べ操作の量が格段に少なくてすむ。

しかしこのような単純な手法で操作の量を減らそうとすると、実際の操作の内容と操作手順の対応関係がわかりにくいため、かえってシステムが使いにくくなる危険がある。上の例では “**ls**” という文字列は “**list**” の略であるためいくらか対応関係が覚えやすくなっているが、“**list**” の略が “**ls**” であることを覚えるための余分な手間が必要になるし、多数のコマンドを使用する場合は省略形を覚えきれなくなる危険もある。

2.2.2　予測 / 例示による操作量の低減

前節のように操作の量を単純に減らす方式に対し、機械に人間の次の操作を予測させることにより人間の操作の量を減らすという手法が考えられる。例えば前節の例の場合、“**list-the-**...” という文字列を全部人間が入力しなくても、“**lis**” まで入力した時点で残りの文字列を機械に

予測させ表示させることができれば、わかりにくい省略名を使うことなく少ない操作で処理を指示できるようになるはずである。

ユーザの操作をシステムに予測させる手法は数多く存在する。上の例のような場合はシステムにコマンド名の辞書を持たせるだけでよいが、ユーザの以前の操作履歴を例として記憶しておくことにより次の操作を予測させるといったことも可能であるし、例示によりデータをあらかじめ明示的に与えておくこともできる。例データから規則を学習したり他データに適用するための推論を行なうためには、機械学習や帰納的推論[1] に用いられる各種の手法を使用することができる。例データからの推論をうまく行なうことにより、次の操作の予測だけでなく、ユーザの意図を汲み取ったり、癖を抽出して適応的な動作をさせることさえできるようになる可能性がある。このように、次の操作を予測したり、与えた例示データから規則を推論したりする予測 / 例示インタフェース手法は、機械を使いやすくするための非常に重要な技術のひとつであるということができる。

2.3 予測 / 例示インタフェースの分類

本章では、現在までに提案されている各種の予測 / 例示インタフェースの特徴を概観する。アプリケーションの知識やユーザの操作履歴などからユーザの次の操作をシステムが予測するものを予測インタフェースシステムと呼び、ユーザが明示的にシステムに例を与えることにより次の操作のためのプログラムを生成する例示プログラミング [1]の手法をインタフェースに応用したシステムを例示インタフェースシステムと呼ぶ。 Myers は、予測 / 例示インタフェースを、プログラムの作成を支援するかどうか・知的処理を行なうかどうかの 2 つの基準で 4 種類に分類しているが [62]、実際には様々な種類の予測インタフェースや例示インタフェースが存在するため明確に 4 種類に分類できるわけではない。本章ではまずプログラミングを行なわない単純な予測インタフェースシステムについて解説し、その後で複雑な処理を含む例示インタフェースシステムの解説を行なう[2]。

2.4 単純な予測インタフェース

コマンド行インタプリタや文書エディタのように、キーボードを使用して文字列のみを操作するシステムにおいて、各種の単純で効果の高い予測手法が広く使用されている。例えば UNIX の**`csh`** や**`tcsh`** のようなコマンドインタプリタ (シェル) では、以前起動したコマンドに対しその省略形や番号を指定するだけで再実行することのできるいわゆるコマンドヒストリ機能や、ファイル名の先頭の何文字かを指定するとそれに続く文字列の候補リストをシステムが提示してくれるコンプリーション (補間) 機能 がよく使われているし、テキストエディタ GNU Emacsではコンプリーション機能に加え、テキスト中に既に現われているの文字列を参照して次の入力を予測する **`dabbrev`** (Dynamic ABBREViation) 機能 が広く使用されている。これらにおいては、ファイル名 / コマンド名 / テキスト中の文字列などが静的または動的辞書として予測に使用されている。

Greenberg は、 UNIX のシェルやインタプリタのフロントエンドにおいて操作履歴をどの程度再利用可能かどうかについて大規模な実験を行ない、その結果にもとづいて WORKBENCH システムを提案した [21]。近い過去に起動されたコマンドは近い将来に再度実行される可能性が高いため予測や再利用に有効であることを示し、最近使われたコマンドの引数などを一般化したもののリストを常に表示しておくことが有効であることを示した。

---

[1] PBE(Programming By Example) または PBD(Programming By Demonstration) システムと呼ばれることもある。

[2] 本章で述べるシステムの多くは [14][98][116][119][120]などでも解説されている。

仮名漢字変換システムも、大規模な辞書を用いて予測を行なわせる予測インタフェースの一種である。例えばGNU Emacs上で動作する仮名漢字変換システムSKK[93]では、図2.1のような単純な型式のテキストが辞書として使用されており、ユーザが例えば「あいいん」と入力した後変換キーを押すことにより「愛飲」「合印」が順に提示される。SKKシステムではコンプリーション機能も使用することができる。

```
あいあいがさ / 相合い傘 /
あいあん / アイアン /
あいいく / 愛育 /
あいいろ / 藍色 /
あいいん / 愛飲 / 合印 /
あいうち / 相打ち / 相内 /
あいうら / 相浦 /
あいえんか / 愛煙家 /
...
```

図 2.1: SKK の辞書の例

Darraghらの Reactive Keyboard システム[16]では、UNIXのシェルのようなコマンド言語インタプリタやテキストエディタのキーボード操作において、ユーザの以前のキーストロークの頻度情報から次の入力文字列を予測し、表にして提示してユーザに選択させたり、入力場所を示すカーソルの位置に候補を表示したりすることにより、障害者やタイピングが苦手な人の補助に成功している。提示された予測結果がユーザの要求と一致していた場合はそれを採用し、一致していない場合は予測結果を無視して作業を続けることができる。Reactive Keyboardでは、テキスト圧縮手法として性能の良いPPM (Prediction by Partial Match) 法[5]に似た頻度集計方式を使用してユーザが次に入力する文字列の予測を行なっている。

IBM-PC上の KeyWatchシステム [53] は、同じキーストローク列の繰り返しをシステムが発見すると音などでユーザにそれを知らせ、マクロとして再実行可能にする。

罫線引き機能[102]も非常に単純な予測インタフェースと考えることができる。罫線引き機能とは、テキスト編集プログラムにおいて、上下左右の矢印キーの操作により矢印の方向にカーソルを動かしながら罫線を引くことにより図を書く機能である。現在のコンテキスト (カーソルの下の罫線の方向) 及び直前の操作 (罫線描画方向) から次の動作を決定しているため一種の予測インタフェースとみることもできるが、予測がはずれることはほとんど無いためそのようには認識されていないと思われる。

上述のような単純なテキスト予測方式を表にしたものを図2.2に示す。

名前	システム	予測に使用する情報
dabbrev	Emacs	文書中の文字列 (動的辞書)
completion	Emacs	コマンド名 / ファイル名など (静的 / 動的辞書)
罫線引き	Emacs	直前のキー及びカーソル直下の文字
かな漢字変換	各種ワープロ	かな漢字辞書
“**.**”	vi	直前のコマンド
“**!!**”	csh	直前のコマンド
“**!**(string)”	csh	コマンド履歴
“ESC **l**”	tcsh	ファイル名
Reactive Keyboard	shell	キー入力の統計情報

図 2.2: 各種の予測手法

これらの単純な予測方式においては、履歴情報・辞書などのアプリケーション知識・現在のコンテキストなどが予測情報として使用されている。

2.5 例示インタフェース

前節で述べたような単純な予測インタフェースはユーザの作業量の軽減にある程度有効であるが、操作手順をプログラム化しているわけではないので適用範囲は限られており、長い操作手順の繰り返しの自動化やパラメタを含む操作には役にたたない。複雑な作業の自動化を行なうためには操作手順をプログラムとして定義 / 実行する仕組みが必要である。

多くの複雑なアプリケーションでは、独自のプログラミング言語の使用によりユーザの処理をプログラムとして定義することができるようになっている。例えばテキストエディタ GNU Emacs では Emacs Lisp を使用してあらゆる操作をプログラムすることが可能であるし、 Microsoft 社の表計算ソフトウェア Excelでは Visual Basicを使用して処理を記述することができる。また、任意のプログラムを定義することができないものでも、一連の処理手順をマクロとして定義し再利用することのできる「マクロ定義機能」を備えるアプリケーションは多い。このようなプログラミング言語やマクロ定義機能を使用することによりユーザは複雑な処理を自動化することができるが、プログラム作成の知識が要求されるので誰でもこのような機能を使用することはできないし、プログラム作成には一般に手間がかかるため知識のあるユーザでも気軽にその機能を使うことはできない。

このような問題を解決するため、予測インタフェースと同じような気軽さで操作を自動化するプログラムを作成することができるように、システムに示したユーザの操作例からシステムが自動的にプログラムを生成するような、例示によるプログラム作成手法をインタフェースに応用する例示インタフェースの手法が各種提案されている。例示によるプログラミングとは、ユーザがプログラムを自分で書くかわりに各種の例をシステムに与えることによりシステムが推論などにより自動的にプログラムを抽出するシステムである。前節の予測インタフェースシステムは、システムに与える例としてユーザの操作履歴や辞書を使用した単純な例示インタフェースシステムと考えることができる。

2.5.1 例示インタフェースの特徴

例示インタフェースには以下のような利点があると考えられている [62]。

- 複雑な操作を何度も行なう場合、同じ操作を何度も繰り返す必要がない
- プログラミングの知識がなくても、操作手順を示すプログラムを作成することができる
- 抽象化が要求されるプログラミングという作業を行なわず、具体的な操作を通じて、操作に対する結果をひとつずつ確認しながらプログラムを作成していくことができる

あらゆる例示インタフェースシステムはこのような特長を持っているが、例示インタフェースの実現手法は様々であり、例示操作を記録 / 再生するだけの単純なものから、複数の例示データをもとにして汎化学習を行なうものまで、レベルの異なる各種の例示インタフェースシステムが提案されている。

2.5.2 例示インタフェースシステム例

本節では、現在までに提案されている各種の例示インタフェースシステムを個々に紹介する。

キーボードマクロ

キーボードマクロとは、キーボードを使用したテキストベースのアプリケーションにおいて、ユーザが実際のキー操作を行ないながらその操作列をマクロとして登録することのできる機構である。例えば GNU Emacs のキーボードマクロでは、ユーザはまず特別のキー操作 (通常^X ( ) によりマクロ登録の開始を指示し、その後ユーザは一連のキー操作により実際の編集処理を行ない、最後に特別のキー操作 (通常^X)) によりマクロ登録の終了を指示することにより一連の操作をマクロとして登録する。登録されたマクロは特別のキー操作 (通常^X e) により実行することができる。

キーボードマクロは単純な機構ではあるが、ユーザがプログラミングを行なうことなく操作の例示により操作列を定義することができるため、最も有効な例示インタフェース手法のひとつとなっているが、以下のような問題点を持っている。

1. 操作の並びしか定義できない
2. 明示的に操作の開始と終了を指示する必要がある
3. 登録開始を指示せずに操作を実行してしまった後ではその操作をマクロとして定義できない

キーボードマクロは単純な機構であるため 1. は仕方がないものの、操作量を減らすという意味では 2. 及び 3. は問題になる。例えば n 個のキー操作からなる作業を m 回繰り返す場合を考えてみる。$n = 4, m = 5$ の場合、単純に操作を行なった場合は 20 回のキー操作で全作業が終了するが、2 回操作を行なった後で繰り返しに気付いてマクロを登録したとすると、$4 \times 2 + 1$(マクロ登録開始指示) $+4$(マクロ登録のためのキー操作) $+1$(マクロ登録終了指示) $+1$(マクロ実行のためのキー操作) $\times 2$(マクロ呼出し回数) $= 16$ 回のキー操作ですむことになる。しかしこの例の場合はあまりキー操作の節約に役にたっているとはいえないうえに、マクロ登録や呼出しに失敗する可能性があるし、マクロ定義機能の存在を思い出す必要があるなどユーザの負担は大きい。n が大きい場合は節約効果は大きくなるが、登録に失敗する可能性も大きくなるので、n が小さく m が大きい場合に最も効果的に働くと考えられる。 この例のように、作業の途中でプログラムを作成したりマクロを登録しようとする場合には共通の問題点がみられるが、これについては 3.2.3節で解説を行なう。

キーボードマクロは以下に述べるような他の高度な例示インタフェースシステムに比べると使用法が単純であるが、上述のような問題点があるため初心者にはあまり利用されていないようである。

Editing By Example[68]

Nix の Editing by Exampleシステム [68] は、テキストファイルの多くの部分に同じような修正を行なうとき、修正前のテキストと修正後のテキストの組の例をユーザが示すことによりシステムにその変換規則を推論させ、残りのテキストに対しその変換を適用させることができるようにするものである。ほとんどの例示インタフェースシステムはユーザの実際の操作を例データとして扱うが、 Editing by Example システムは操作前 / 操作後の変化のみを扱うという特徴がある。

編集操作としては、正規表現を使用した文字列置換のサブセットである “gap programming” のみが許されている。少ない数の変換例からそれを表現する正規表現を導くことは困難であるが、 gap programming は、編集操作に必要な機能を維持しつつ例からの置換形式の推論がしやすいようになっている。

TELS[86]

Mo らの TELSシステム [56] [86] は、 GUI による単純なテキストエディタ上でのユーザの繰り返し操作から汎化によりプログラムを生成して以降の操作に適用可能とするシステムである。簡単のため編集操作は挿入 / 削除 / 移動 / 選択の 4 種類しか許されていない。似たような操作が繰り返された場合、各操作の前後でのコンテキストを汎化することにより積極的にループを検出してプログラムを生成する。例えば電話番号を順次選択していくような場合、 “空白文字の後の数字**222-3456** の選択” と “空白文字の後の数字**234-5555** の選択” という操作が続けて実行された場合は “空白文字の後の数字**2**-**5*** の選択の繰り返し” というプログラムを生成する。このように、システムは同じ機能をもつプログラムのうちなるべく最小のものを見つけようと試みる。生成されたプログラムを実行するときに間違いが発見された場合は、ユーザが操作を訂正することによりインクリメンタルにプログラムを修正することができる。ヒューリスティクスにより一番もっともらしい操作が選択されるようになっていることに加え、修正により正しい条件判断ができるようになっているため、 Nix の Editing By Example システムでは扱うことができない複雑な処理も例示のみで指定可能となっている。

図 2.3のようなテキストを図 2.4のように修正する場合にプログラムが生成される様子を図 2.5, 2.6に示す。クリックなどの操作を行なったときは、図 2.5のように、その前後にどのような単語が存在するか / 文章のどの位置でクリックされたか / などのコンテキストが記憶される。同じような操作に対しては汎化を行なうことにより図 2.6のようなループを含むプログラムが得られる。

```
John Bix, 2416 22 St., N.W., Calgary, T2M 3Y7. 284-4983
Tom Bryce, Suite 1, 2741 Banff Blvd., N.W., Calgary, T2L 1J4. 229-4567
Brent Little, 2429 Cherokee Dr., N.W., Calgary, T2L 2J6. 289-5678
Mike Hermann, 3604 Centre Street, N.W., Calgary, T2M 3X7. 234-0001
```

図 2.3: 編集前のテキスト

```
John Bix,
2416 22 St., N.W.,
Calgary,
T2M 3Y7.

Tom Bryce,
Suite 1,
2741 Banff Blvd., N.W.,
Calgary,
T2L 1J4.

Brent Little,
2429 Cherokee Dr., N.W.,
Calgary,
T2L 2J6.

Mike Hermann,
3604 Centre Street, N.W.,
Calgary,
T2M 3X7.
```

図 2.4: 編集後のテキスト

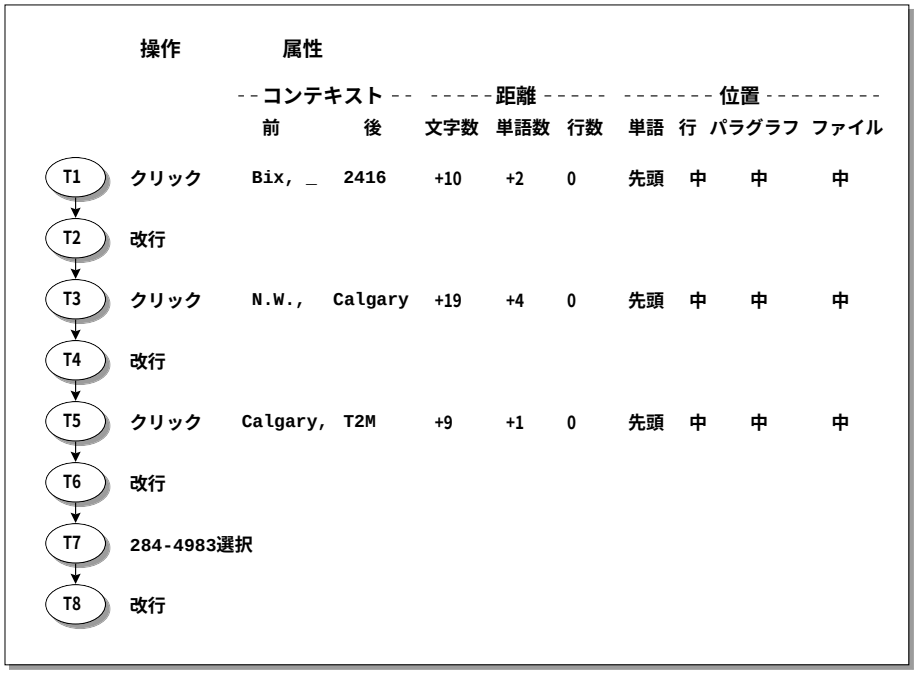


図 2.5: TELS によるプログラムの生成 (1)

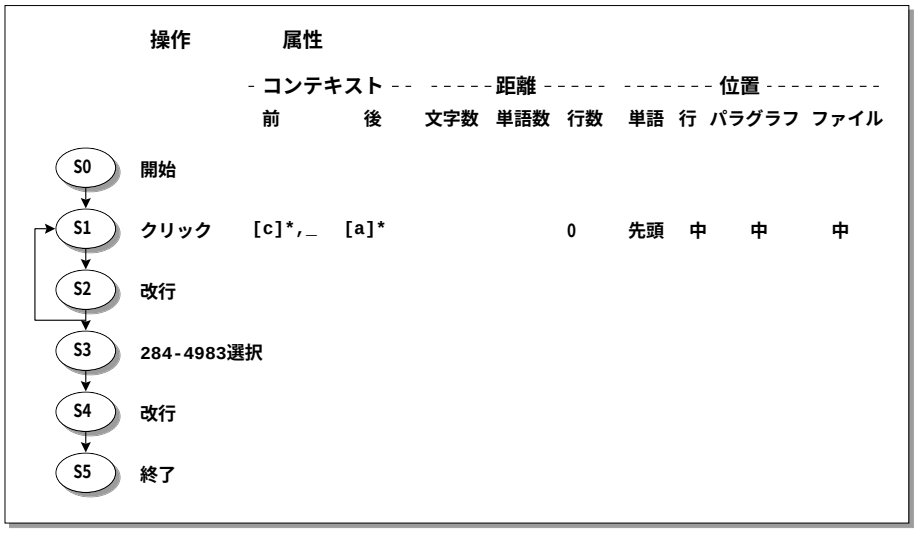


図 2.6: TELS によるプログラムの生成 (2)

Eager[13]

Eagerは、 Macintosh の HyperCard 上で似たような操作をユーザが繰り返したとき、システムがそのパタンを検知してユーザの次の操作を予測するシステムである。ユーザの次の操作が予測できたとき、システムはユーザが次に実行すると思われる操作を先取りしてユーザに示す。例えばユーザが次に特定のウィンドウを選択するであろうと予測した場合、システムはそのウィンドウをハイライトすることにより予測が行なわれていることをユーザに示す。ユーザはシステムの予測の有無にかかわらず操作を続けることができるが、ユーザがそのウィンドウを選択したときは予測が正しかったことをシステムに伝えていることになり、そうでない場合はシステムの予測を否定したことになる。そのような操作を何度か繰り返した後、システムが正しい予測を行なっているということをユーザが確信した場合は、残りの処理の自動実行をシステムに指示することができる。

システムはヒューリスティクスを用いてユーザの操作の繰り返しを検出する。例えばユーザが “月曜”、 “火曜” と順番に入力した場合はシステムは次は “水曜” であることを予測する。

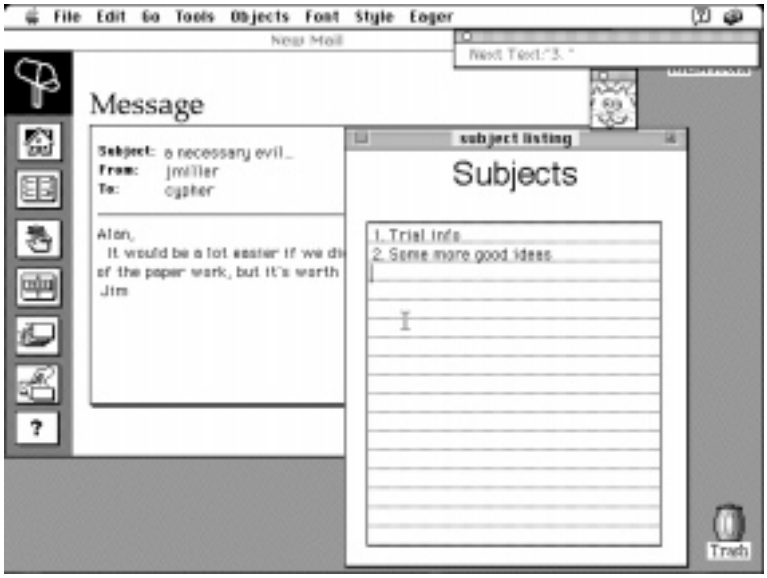


図 2.7: Eager

図 2.7に Eager の動作例を示す。ユーザは左上のウィンドウから**`Subject:`** の後の文字列を切り出して右下のウィンドウに貼り付けているところであるが、このような操作を 2 回繰り返すと右上のような「猫」が出現して繰返しの存在を知らせ、次のユーザの操作の予測を行なう。

SmallStar[25]

SmallStarは、 Xerox 社の Star ワークステーションのデスクトップの GUI 操作を例示によりプログラミング可能にするシステムである。 GUI 操作のプログラミングを簡単にするため、ユーザはキーボードマクロと同様に実際の操作を行なった後でその操作列を編集することによりプログラムを作成する。ユーザはプログラムしたい操作列の実行開始をシステムに指示し、実際の操作を行ない、最後に操作列の実行終了をシステムに指示することにより操作列を示すテキスト形式のプログラムを得て後でそれを修正することができる。例えば、「全ての “**`*.bak`**” という名前のファイルを “**`bak`**” というディレクトリに移動する」というプログラムを作成したい場合は、まず上記の手法により適当なファイルを “**`bak`**” ディレクトリに移動することにより「ひとつのファイルを “**`bak`**” というディレクトリに移動する」というプログラムを作成し、そのプログラムをエディタで編集して条件文を付加することにより必要なプログラムを作成することができる。

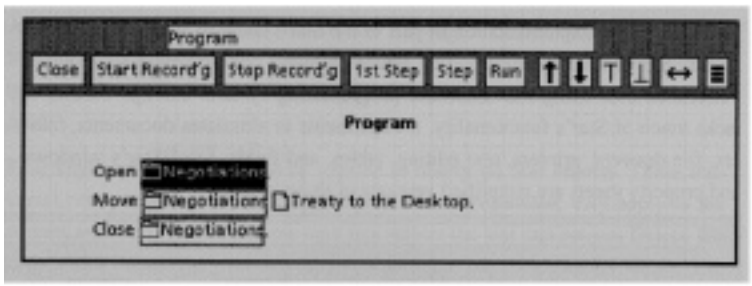


図 2.8: 例示により自動的に作成されるプログラム

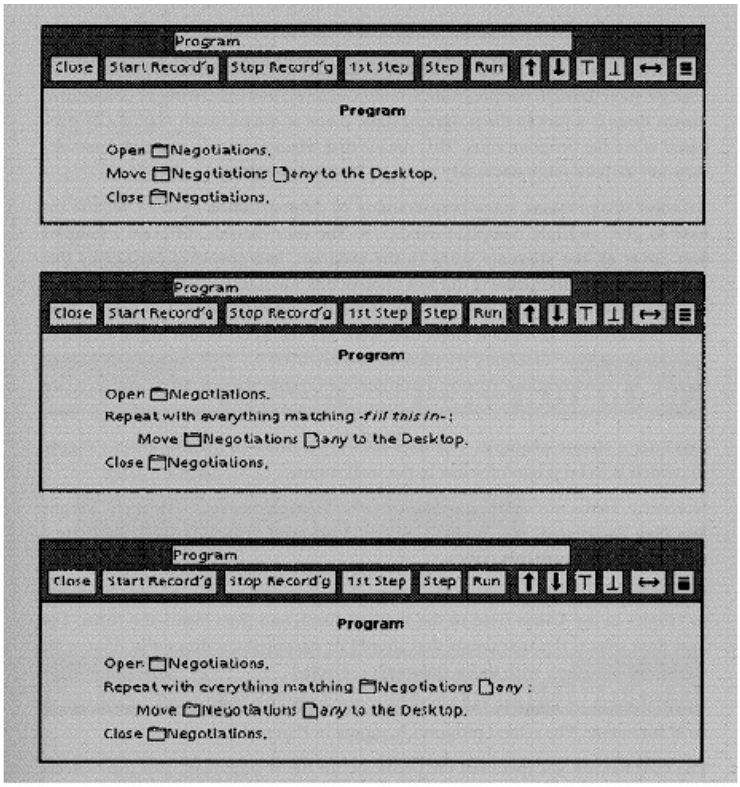


図 2.9: 編集により修正が加えられたプログラム

Chimera[40] / MacroByExample[41]

Chimera[40][39] は、図形エディタなどにおける編集操作の履歴をプログラムとして後で編集することにより、 GUI 操作をマクロとして定義可能にするシステムである。 SmallStar では GUI 操作列をプログラムテキストとして表現して後の編集を可能としていたが、 Chimera では GUI 操作をグラフィカルな形で表現したまま編集ができるような工夫を行なっている。 GUI 操作履歴はユーザにわかりやすいように図 2.10の編集操作履歴を図 2.11のように紙芝居風に表現し、細かい操作列をひとつのコマで表現したり、局所的な操作において重要な部分だけがコマ上に表現されるような工夫を行なっている。

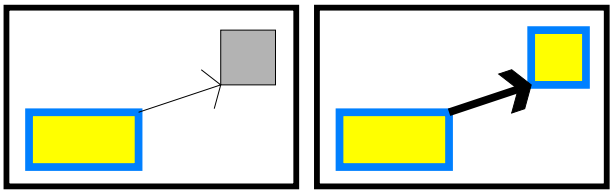


図 2.10: Chimera の編集操作例

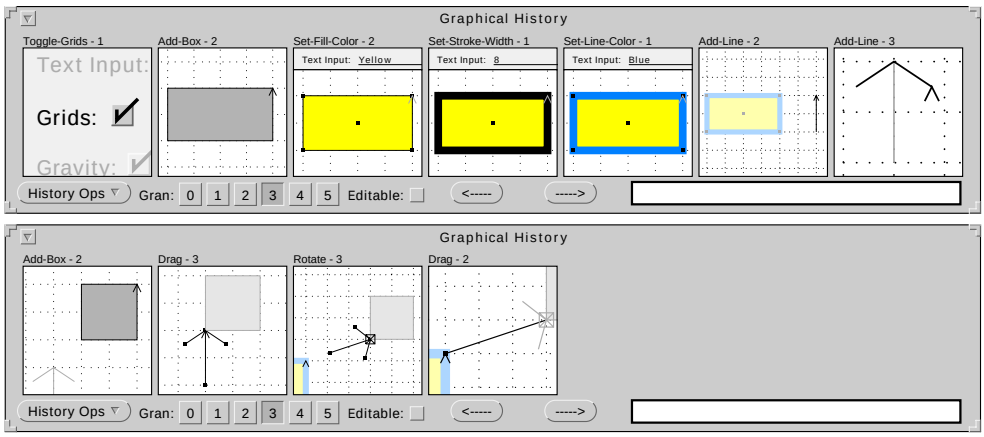


図 2.11: Chimera における図 2.10の操作履歴のグラフィカル表現

Mondrian[42]

Mondrian[42] も Chimera と同じように、図形の編集操作履歴をマクロとして登録することのできるシステムである。登録される各編集操作は編集前の状態と編集後の状態を組にしたドミノ状のアイコンで表現され、ビジュアルプログラミングの要素として再利用することができる。Chimera の場合と同様に、マクロ化にあたってシステムは各種の推論や汎化を行なう必要があるが、推論の様子を音声や自然言語テキストでユーザに知らせる工夫がされているため、間違った推論が行なわれた場合はユーザはすぐにそれに気付いて修正を行なうことができる。

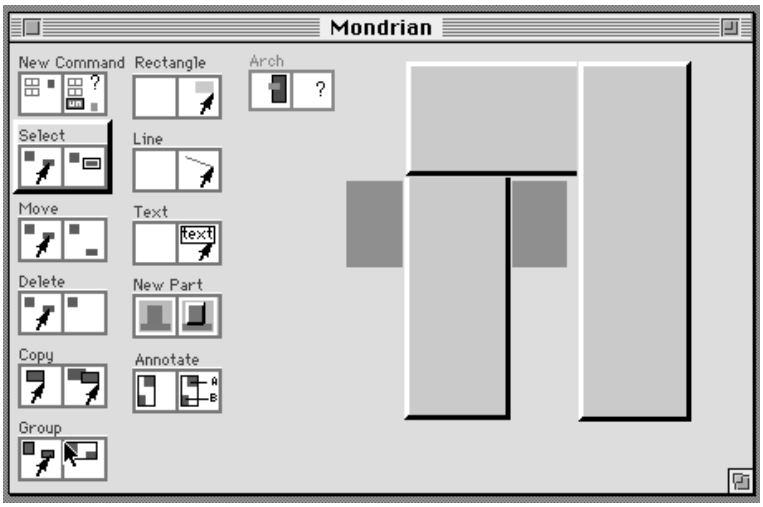


図 2.12: Mondrian

Metamouse[52]

Metamouse[52] もグラフィカルな図形の編集操作から編集プログラムを自動生成するシステムである。編集操作は、「上に移動させた直線が矩形に接したとき直線の長さだけ矩形を移動させる」といったようなプロダクションルールの集合として表現される。ループを構成する規則を使うことにより繰り返しを含むプログラムも作成できる。規則を発火させる条件としては、この例のように「接する」「交わる」といった幾何的条件が重視される。以前の操作と完全に同じ操作を始めたときは、システムはすぐにユーザの次の行動を予測して提示するため同じ操作を 2 度以上含む大きな繰り返しは検出できない。

規則はユーザの与える少ない例からの汎化により生成されるが、システムは推論の様子がユーザによくわかるように擬似マウスや制約条件を表示しながら徐々に推論を行なう。ユーザはシステムに自分の意図を教えるという意思を念頭に置きながら、間違った推論を修正しながら正しいプログラムを作成していく。

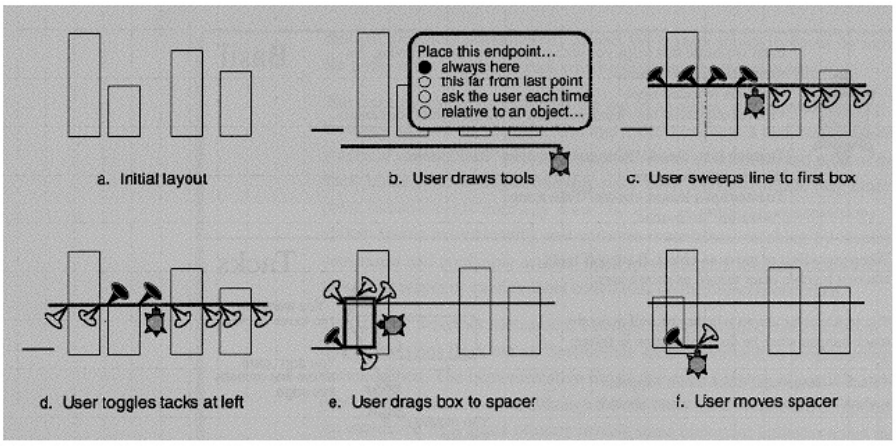


図 2.13: Metamouse(1)

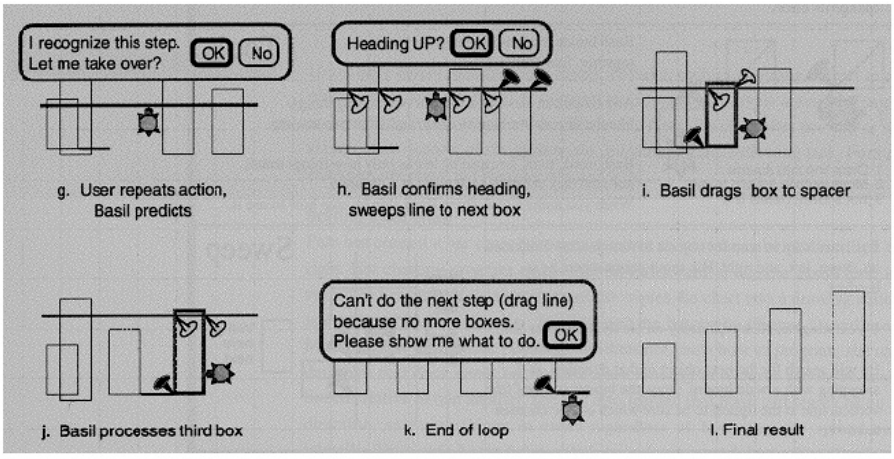


図 2.14: Metamouse(2)

Triggers[73]

Triggersは、アプリケーションが表示するビットマップ画面上のパタンマッチングにより操作の自動実行を行なわせるシステムである。画面上のビットマップパタン及びそれに関連する操作を例示により定義することにより、同じビットマップパタンが見つかったとき、定義された操作を自動実行させることができる。例えば、表計算ソフトにおいて全ての負の数値データに印をつけたい場合は、マイナス記号 “-” を示すビットマップパタンを指定してそのパタンの近辺に印をつける操作を例示により定義することにより、画面上の残り全ての “-” の近辺に同じ印をつけることができる。

ほとんどの予測 / 例示インタフェースシステムでは、予測や例示を行なうためにアプリケーション内のデータを利用するのが普通であるが、 Triggers はビットマップ画面のみを用いて操作の例示を行なう。 3.2.3節で述べるように、アプリケーション内のデータには必ずしもアクセス

可能とは限らないが、 Triggers は画面に表示されるビットマップデータのみを使用するため、画面へ表示を行なう任意のアプリケーションに対して適用することができる。

Layout by Example[32]

Layout By Exampleシステム [32] は、図形を配置するためのレイアウト規則を例示により推論するシステムである。図形が 2 個の場合の配置例、 3 個の場合の配置例... をユーザがシステムに与えることによりシステムは配置アルゴリズムを推論する。アルゴリズムは「内挿」と「繰り返し」をもとに構築する。

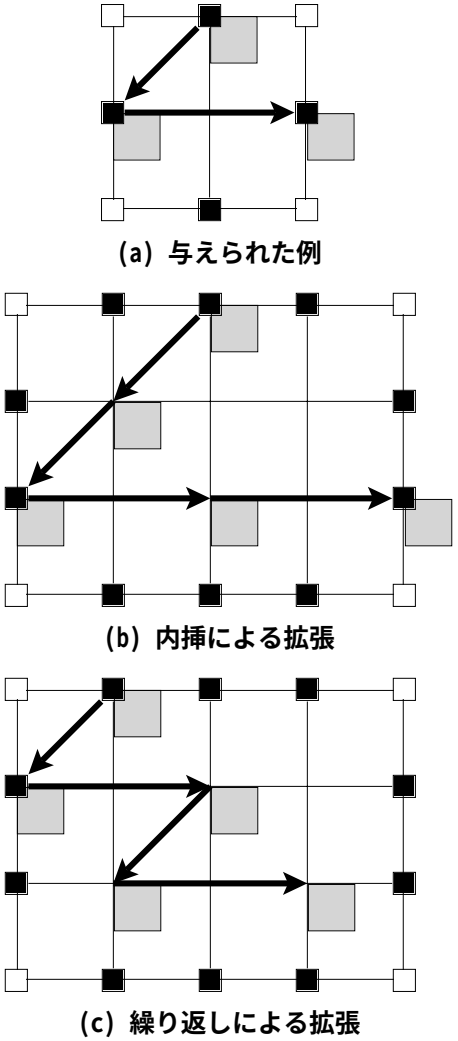


図 2.15: Layout by Example における推論

例の数が少ない場合は、その例を満足する配置アルゴリズムはいくつも存在するが、このような場合システムは別のデータに対しそのアルゴリズムを適用してその結果をユーザに示すことにより、ユーザはそのアルゴリズムが適当なものであるかどうかを判断する。このように、実際は推論された複数のプログラムの中からユーザの要求に一致するものを選択しているにもかかわらず、ユーザとシステムの間では例データのみがやりとりされるためプログラムを明示的に扱う必要が無いという利点がある。

TRIP3[55]

TRIP[33], TRIP2[80][95], TRIP3[55] は制約指向の宣言的図形配置システムである。最初に開発された TRIP は、図形間の制約関係を Prolog で宣言的に記述することにより図形の自動配置を行なうシステムであったが、 TRIP2 では図形の実際の配置の変更により制約の記述を修正するような双方向変換が可能となり、 TRIP3 ではさらに複数の配置例から宣言的配置制約を推論することができるようになっている。

IMAGE[54][121]

TRIP3 の後継である IMAGE[54][121] では、推論された配置制約を別の例に適用した結果をユーザに提示することによりその推論が正しいかどうかを判断できるようになっており、 Layout by Example システム [32] と同様の効果が得られる。

Peridot[60]/ Marquise[65]

Brad Myers は例示によりユーザインタフェースを作成する各種の試みを提案している。 Peridot[60] では、ユーザがシステムに示した GUI の操作例から推論を行なうことによりメニューやスクロールバーなどの動きをカスタマイズする。また、 Marquise[65] では、描画エディタのようなアプリケーションの GUI ほとんどの要素を例示でプログラムすることを試みている。インタフェースビルダは一般に静的な画面設計を行なうモード及びそれをテストするモードを持っているが、 Marquise ではこれに加え、マウスドラッグ時等のシステムの動的な反応を定義するための Train モードと Show モードを持っており、ユーザ操作に対してシステムがどのように反応するかも例示により指定することができる。システムは例示されたアクションから本当のアクションを推論する。例えば Train モードにおいてマウス押下とマウス移動を指定し、 Show モードでその間に点線を引くと、システムは「マウスドラッギングにより点線が移動する」ことを推論する。描画エディタにおいてオブジェクトの生成や属性変更によく使用されるパレットも簡単な操作で作ることができる。

KidSIM[15][77]

KidSIM[77] は、プログラミングの知識の無い人や子供でも簡単なアニメーションを作成することできることを狙った簡易シミュレーションショステムである。格子状の画面上に物体を配置し、単位時間毎に規則に従った動作をさせることによりアニメーションが実行されるが、規則の作成が例示により行なわれる。例えば単位時間毎に右に進む物体を表現したい場合は、ふたつの格子を指定してから左側の物体を右に動かすことにより時間経過の前後の様子を表現し、「左側に物体のある並んだ格子があるときは単位時間後にその物体は右に移動する」という規則を作成する。このような規則をいくつも例示により作成することにより複雑なアニメーションを作成することができるようになっている。

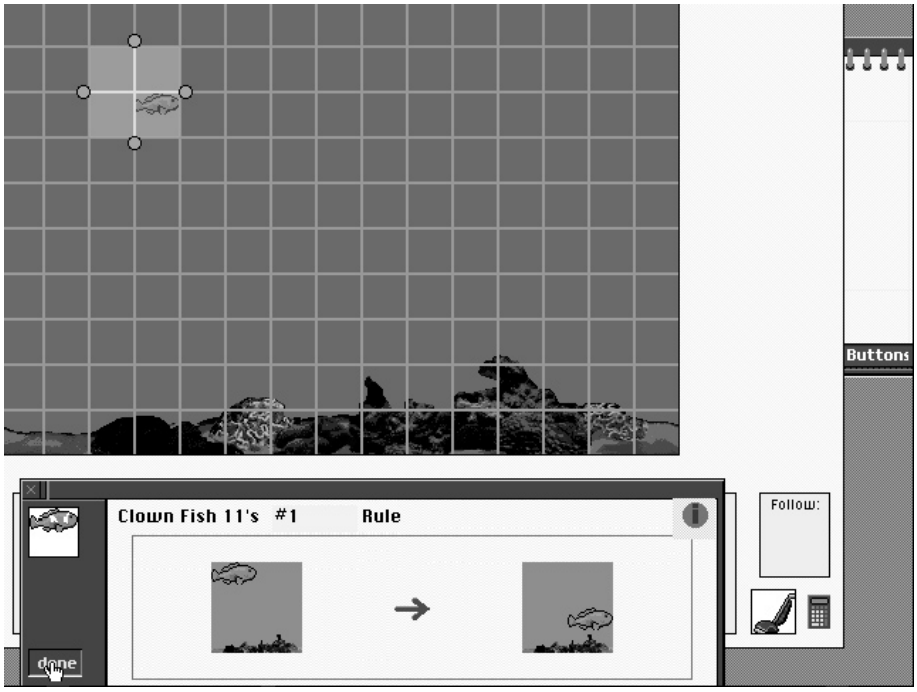


図 2.16: KidSIM

図 2.16は KidSIM の規則の定義の例である。画面上側のウィンドウの 2 × 2 の格子内で魚を移動させることにより、画面下側のような「魚の右 / 下 / 右下が空白であるとき、次のタイミングで魚は右下に移動する」という規則が作成される。この規則を単位時間毎に順次適用すると、画面上の魚はどんどん右下に移動していくことになる。

Pavlov[87]

Pavlov は、ユーザが例を示すことにより、時間経過やユーザの操作を刺激とするアニメーションやシミュレーションを作成することのできるシステムである。描画モード / 刺激定義モード / 反応定義モード / テストモードを使いわけることにより、ユーザ操作や時間などの刺激信号に呼応するオブジェクトの生成 / 変型 / 削除 / 前進 / などの反応を定義していく。時間経過を刺激として使用することができるため、 Macromedia Director のような、時間に基づくアニメーション作成システムと似た使い方をすることができる。

ひとつの刺激 / 反応を定義するためには一度だけ例示を行なう。システムが誤った推論を行なった場合はユーザがテキスト編集により修正する。

反応は、刺激の種類 / パラメータ / 反応の生ずる条件により定義される。例えば「ハンドルを右に回すと車が右を向く」という刺激 / 反応の組を定義する場合、システムがヒューリスティクスにより選択した各種の条件から選択を行なうことにより、「車が障害物に重なっていないとき」などといった条件を追加することができる。

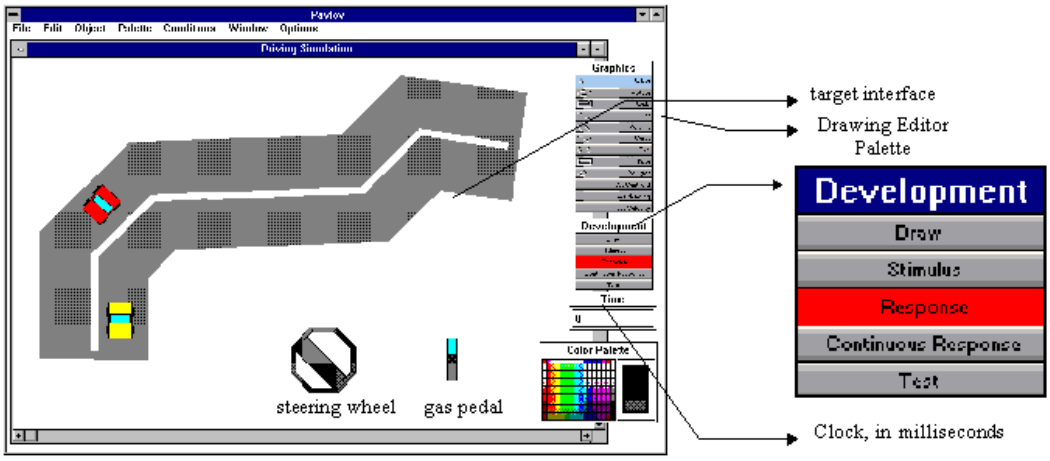


図 2.17: Pavlov

Pursuit[57] [58]

Pursuit [57] は、ユーザの例示操作からの推論 / 汎化により理解の容易な視覚言語プログラムを生成し、それをユーザが直接編集することによりシステムの間違った推論を修正できるようにしたシステムである。

プログラムの例を図 2.18に示す。視覚言語の表現を画面上での実際の操作で使われるアイコンに似せることにより、プログラムをより理解しやすいように工夫している。 Mondrian と同様に、操作前の状態と操作後の状態を組にして示すことにより操作内容を直感的に表現している。図 2.18のプログラムには条件分岐が含まれているが、条件分岐や繰返しもユーザの操作からある程度自動的に抽出できるようになっている。

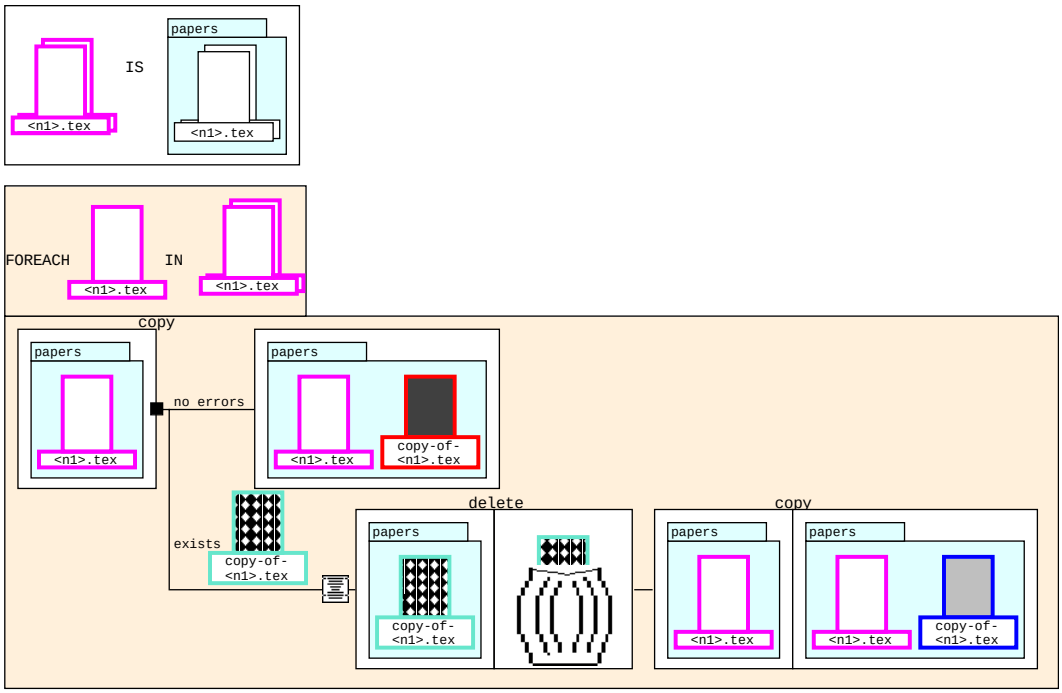


図 2.18: Pursuit

DemoOffice[94]

DemoOffice[94] は、アプリケーション上の GUI 操作から再利用可能な部分を自動的に抽出してマクロとして登録するシステムである。メールボックスのようなデータ集合の要素 (一通のメール) に対して選択やコピーなどの GUI 操作が行なわれた場合、システムは他の要素に対しても同様の操作が行なわれる可能性があると判断し、自動的にマクロ生成を開始する。またデータの依存関係を解析することにより、操作された要素に関連した操作のみを履歴から抽出 / 汎化してマクロを生成する。異なる操作が行なわれたときにはまた別のマクロを自動生成する。実際に他の要素に対して同じ操作が必要になった場合は、用意されたマクロに対して新しい要素を適用可能かどうかをユーザが実行前に試してみることができるようになっている。

Gold[64]

市販の表計算ソフトウェアでは、表データを棒グラフをはじめとする各種のグラフに変換する機能を備えているものが多いが、このような機能はシステム組み込みのものであるためグラフの種類は限られており、特殊なグラフを書かせることはできない。 Gold[64] は、表の中のデータの値とグラフ描画に使用される図形の属性との対応を例示により指定することにより特殊なグラフ化を可能にするシステムである。例えば棒グラフを作成する場合は、最初に軸と矩形をひとつ描き、矩形の位置 / 大きさ / 色などの属性が表中のひとつのデータとどのように対応しているかを指定し、残りのデータにも同じ対応関係を適用する。

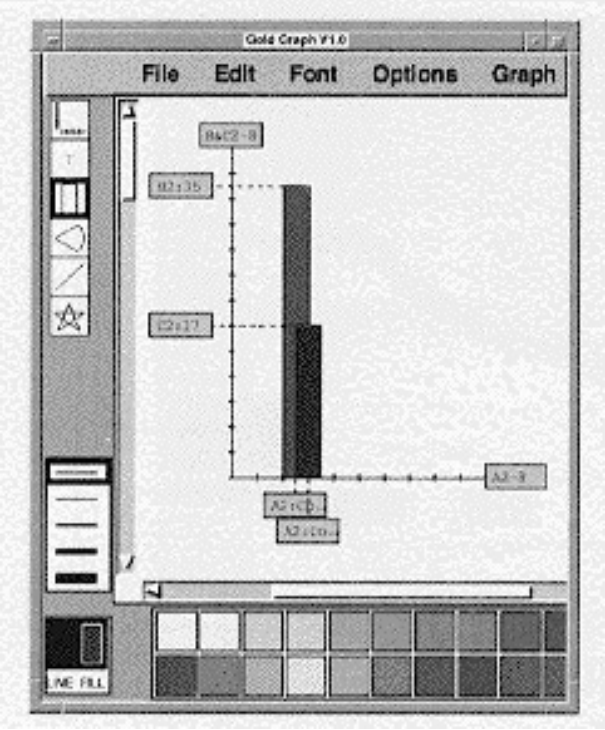


図 2.19: Gold における例示

図 2.19は、図 2.20のような表データから例示により棒グラフを作成しようとしているところである。棒の各種の属性が表中のデータとどのように対応しているかが示されている。例示により作成された視覚化手法をデータ全体に適用すると図 2.21のようなグラフが得られる。

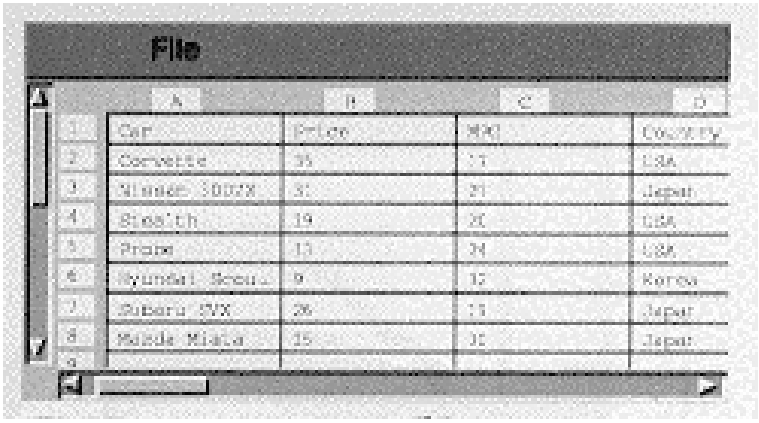


図 2.20: 使用される表データ

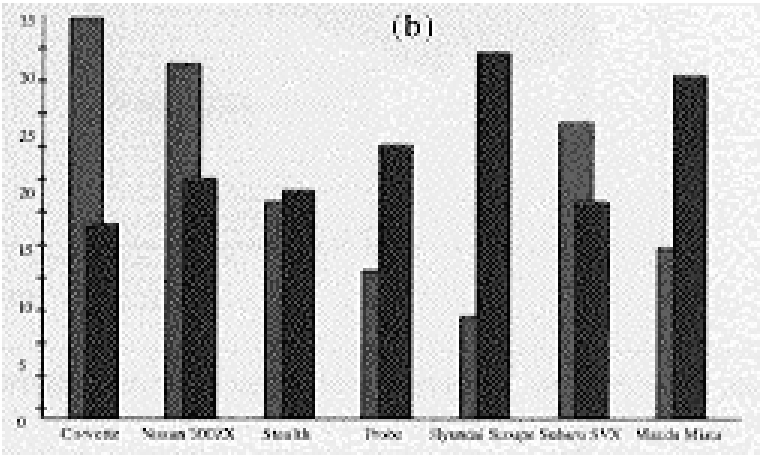


図 2.21: データ全体をグラフ化した結果

システムは「棒グラフに使われる矩形の片側は軸に接する」といった知識を持っているためユーザはあらゆる属性の視覚化手法を指定しなくてよい反面、推論方式がよくわからず期待と異なる動作をしてユーザが混乱する危険もあると考えられる。

Sage[74]

Sageは Gold と同じように例示により表データの図示を行なうシステムである。 Sage は視覚化のためのツール SageBrush と視覚化のための知識ベース SageBook から構成される。ユーザは SageBrush を使用して描いた図形の属性と表データを関連付けることにより視覚化規則を定義し、その知識を SageBook に格納する。 SageBook を使用することにより以前の視覚化規則を再利用することができる。

図 2.22はナポレオンのロシア行軍記録を視覚化したものであり、軍の位置、兵隊の数、気温など数多くの属性をもつデータがひとつのグラフとして表現されている。 SageBrush では例示によってデータと属性を容易に関連付けることができる。

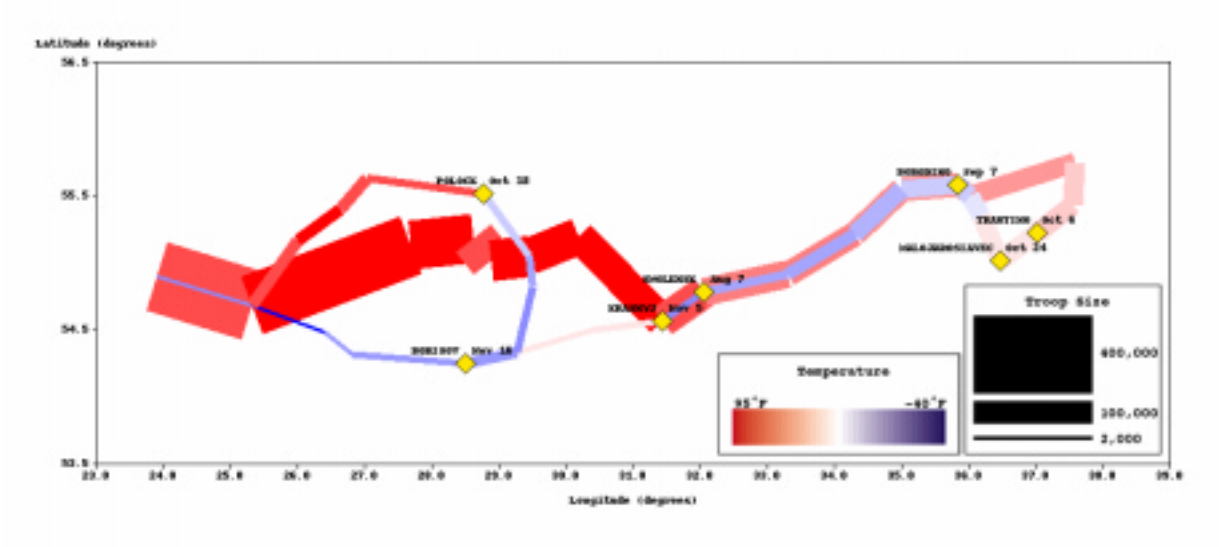


図 2.22: Sage による表データの視覚化例

2.6 まとめ

本章では、操作量の低減のために予測 / 例示インタフェースが有効であることを示し、現在までに提案されている各種の予測 / 例示インタフェースシステムについて紹介し、その特徴を述べた。次章ではこれらのシステムのもつ欠陥を整理し、有用な予測 / 例示インタフェースシステムの構築に必要な条件についての考察を行なう。

第 3 章　新しい予測 / 例示インタフェースの設計方針

3.1　はじめに

前章では現在までに提案されている各種の予測 / 例示インタフェースを概観した。商用システムにおいて実用的に使われている単純な予測インタフェースも存在するが、予測 / 例示インタフェース手法の多くはなんらかの大きな問題点を持っているため提案の段階にとどまっており実用となっていない。本章では、既存の予測 / 例示インタフェースのもつ問題点を解決するための新しい予測 / 例示インタフェースの要件についての考察を行なう。

3.2　予測 / 例示インタフェースに関する議論

予測 / 例示インタフェースでは, 予測の実行や例示によるプログラミングが単に可能であるだけでなく、ユーザが予測 / 例示インタフェースを便利だと感じるようでなければ意味が無い。本節では、予測インタフェースが有効に働くための条件や、例示によるプログラム作成における本質的な問題点についての議論を行なう。

3.2.1　予測 / 例示インタフェースに必要な情報の取得

正しい予測を行なわせるためには、予測に使用するための情報を充分に得る必要がある。複雑な予測を行なわせようとする場合は特に情報が沢山必要になるので、これが不充分であったり解析が困難である場合には高度な予測を行なうことはできない。例えば自動車の運転における運転者の次の操作を計算機に予測させることはほとんど不可能であろう。簡単に取得できる情報として、辞書 / ユーザの操作の頻度情報 / 操作履歴などが使用されることが多い。

3.2.2　予測 / 例示インタフェースが有効に働く要件

予測インタフェースが広く使われるようになるためには一般に以下の要件を満たす必要があると考えられる [112] 。

1. 予測を実行させる手間が予測を使用しない場合に比べ小さいこと
2. 予測がユーザの期待に一致する確率が高いこと
3. 予測機能を使わないユーザの邪魔にならないこと
4. 予測の実行に対しユーザが不安にならないこと

2.4章で紹介したような単純な予測インタフェース手法のいくつかはこれらの条件を満たしているため実際に広く使われているが、例示インタフェースについては各種の研究が行なわれているにもかかわらず、予測を実行させるための例示操作が面倒だったり予測結果が不適当だったりすることが多いために、あまり使われていないのが現状である。

これらの条件についてはユーザによって感じ方が異なるので注意が必要である。例えば高速に文字をタイプできる人間にとっては、予測を指示する操作の手間の方が自分でタイプする手間より常に大きいために予測が全く有効でないことになる[1]。2.4章で解説したReactive Keyboardにみられるような予測候補の提示は、タイピングの初心者や障害者などには親切と感じられるが、それ以外のユーザには邪魔に感じられるだけである[2]。また、非常に簡単な同じ操作を繰り返している場合、次の操作は予測可能でありユーザの期待に一致する確率も高いが、もともとの操作が単純であるためユーザはわざわざ予測機能を使おうとはしないであろう。

一般的に言えば、適度に複雑でかつ確率の高い予測が可能な場合に予測機能が有効に働きやすいということができる。

3.2.3 Just-in-time Programming

例示インタフェースはJust-in-time Programming[72] のための技法と考えることができる。Just-in-time Programmingとは、「必要になったときその場で即時にプログラムを作る」ことである。手作業でもできるがプログラムを作れば自動化もできるような仕事に遭遇したとき、その場で即時に簡単にプログラムを作ることができれば(Just-in-time Programmingが可能ならば)、処理の自動化により時間を節約することができる。例えばUNIXのシェルは、多くの小さなコマンド / パイプやリダイレクションなどの機能 / シェル言語などの組み合わせによりJust-in-time Programmingが容易であるため、それ以前のシステムに比べて有用であるとの評価を得たと考えられる。エディタなどのマクロ定義機能も簡単なJust-in-time Programmingであるが、例示インタフェースシステムは、ユーザが自分でプログラムを書くかわりに例示によりシステムにプログラムを作らせることによりJust-in-time Programmingを実現しようというものだと考えられる。

Just-in-time Programmingには以下のような5つの障害が考えられる [72]。

1. 必要なデータやオペレータにアクセスできない

   システムによっては内部データにアクセスしにくいため有用なプログラムを作成しづらい場合がある。特に商用のアプリケーションでは内部データにアクセスすることができないことが多いため、それらを有効に使用するプログラムを作ることができない。これに対し、EmacsやUNIXのように多くの仕様が公開されているシステムではデータにアクセスすることが容易であるためJust-in-time Programmingを実現しやすい。

2. プログラムの入力に精神的 / 物理的負荷がかかる

   ある程度複雑な処理を自動的に行なわせようとすると、それなりに複雑なプログラムを何らかの方法でシステムに入力しなければならないが、そのためには精神的負荷 (e.g. 手順の考案) と物理的負荷 (e.g. キーボードからのプログラム入力) の両方がかかる。2.5.2節で述べたように、キーボードマクロのような単純なシステムにおいてもそれを定義する行動には負荷がかかってしまう。

3. 汎用計算能力が必要

   ある程度複雑な処理を行なわせるためには、Just-in-time Programmingを行なおうとするシステムが汎用の計算能力をあらかじめ持っている必要がある。単なるマクロ定義 / 実行機能しか提供していないシステムでは条件分岐や繰り返しを使用して複雑な処理を行なわせることができない。

---

[1] タイプミスを修正するとき、後退キーを使って修正するよりも行全体を消して打ち直す方が速いようなタイピストも多いそうである。

[2] Reactive Keyboardの開発者のひとりは手に障害を持っていたため、その人物にとっては予測機能は非常に役に立ったということである。

4. 定義したプログラムを起動するのが面倒なことが多い

   何らかの方法により作成したプログラムは、適切な時点で容易に起動することができなければならないが、このための機能をシステムが提供している必要がある。

5. かえって時間がかかるというリスクや、プログラムがうまく動かないかもしれないといったリスクが大きい。

   プログラムの作成による作業の自動化により本当に処理の効率化が図れるかどうかが自明でないなど、 Just-in-time Programming を行なうにあたってはこれらののリスクがともなうことが多い。

例示プログラミングの手法は、特に 2, 3, 5 の障害を軽減しようとするものと考えられる。2.5.2節で解説した Triggers では、使用する例として画面上のビットマップを使用することにより障害 1 も取り除けるようになっている。

3.3 新しい予測 / 例示インタフェースの提案

2章で示したような数々の予測 / 例示インタフェースが提案されているにもかかわらず、シェルのヒストリやコンプリーション機能のような単純な予測インタフェースを除き、広く普及しているものはほとんど存在しない。最も単純な例示インタフェース機能であるキーボードマクロですら、定義や起動が面倒であるという理由のため一般に広く使用されているとはいえない。本節では、既存の予測 / 例示インタフェースの持つ問題点を明らかにし、有効な予測 / 例示インタフェースの満たすべき条件について考察する。

3.3.1 既存の予測 / 例示インタフェースの問題点

既存の予測 / 例示インタフェースシステムは、ある一面で便利であったとしても、別の面で大きな問題点を有しているために普及が阻害されていると考えられる。単純な予測インタフェースの満たすべき条件については 3.2.2節に述べたが、予測 / 例示インタフェース全般について使用を阻害する要因について列挙する。

1. プログラムを作成できない

   2.4節で紹介した各種の単純な予測インタフェースシステムは、プログラムを作成することができず、次のユーザ操作の単純な予測しかできないため、パラメタを含む操作のマクロ定義や条件判断 / 繰り返しを含むような複雑な操作の自動化に使うことはできない。

2. プログラム生成のための指示が面倒

   2.4節に示したような予測インタフェースシステムでは、システムに対してユーザが明示的にプログラムの作成を指示する必要が無いが、 2.5.2節で示した多くの例示インタフェースシステムでは、プログラミングを行なうことに対するユーザの自覚が必要であり、そのための特別な操作が必要である。キーボードマクロのような単純なシステムにおいても操作の開始と終了を陽にシステムに指示する必要があるが、繰り返し操作を何度か実行してしまった後で繰り返しに気付くことも多く、そのような場合にあらためて繰り返し操作を登録するのはわずらわしい。

3. プログラム実行のための指示が面倒

   例示により適切なプログラムを作ることができた場合でも、これを適切な状況で起動するのが難しい場合がある。例えばキーボードマクロの場合、起動する状況が定義した状況と

違っていた場合は全く異なる結果が得られる可能性があるので、起動するコンテキストにかなり注意する必要がある。

4. 正しいプログラムの作成が困難

予測 / 例示インタフェースシステムは帰納的推論によりプログラムを作成する自動プログラミングシステムの一種と考えることができるが、インタラクションにおける例示データは量が少ないことが多いし、誤りや無関係なデータが含まれていることも多いためプログラムを生成することはむずかしいし、ユーザの意図と一致したプログラムの生成はさらにむずかしい。2.5.2節で紹介した各種の例示インタフェースシステムのうち推論を行なうものでは、システムが間違った推論を行なわないようにするためユーザが頻繁に推論の間違いをチェックして修正する工夫がなされていたり、間違った推論結果を容易に修正できる

	プログラム不可能	定義時の指示が面倒	実行時の指示が面倒	正しい推論が困難	実行リスクが大	時間的リスクが大	勝手な実行が邪魔	大量データが必要	ヒューリスティクスが過度
シェルヒストリ	×	○	○	○	△	○	○	○	○
コンプリーション	×	○	○	○	○	△	○	○	○
dabbrev	×	○	○	○	○	△	○	○	○
Toolsmith[21]	×	○	△	○	△	△	○	○	△
仮名漢字変換	×	○	△	○	○	○	○	○	×
Reactive Keyboard[16]	×	○	○	○	○	○	×	○	○
キーボードマクロ	○	×	△	○	△	△	○	○	○
Editing By Example[68]	○	×	△	×	×	×	○	△	×
TELS[86]	○	×	△	×	×	×	○	△	×
Eager[13]	○	○	○	△	△	○	×	△	×
SmallStar[25]	○	×	△	○	△	△	○	○	△
Chimera[40]	○	×	△	×	△	×	○	△	×
Macro By Example[41]	○	×	△	×	△	×	○	△	×
Mondrian[42]	○	×	○	×	△	×	○	△	×
Metamouse[52]	○	×	△	×	△	×	○	△	△
Triggers[73]	○	×	△	○	△	×	○	○	○
Layout by Example[32]	○	×	○	×	○	×	○	△	△
TRIP3[55]	○	×	○	×	○	×	○	△	×
IMAGE[121]	○	×	○	×	○	×	○	○	×
Peridot[60] / Marquise[65]	○	×	○	×	○	×	○	△	×
KidSIM[77][15]	○	×	○	○	○	○	○	○	○
Pavlov[87]	○	×	○	△	○	△	○	○	△
Pursuit[57]	○	×	△	×	△	△	○	○	△
DemoOffice[94]	○	×	△	×	○	△	○	○	△
Gold[64]	○	×	△	△	○	×	○	○	×
Sage[74]	○	×	△	△	○	×	○	○	×

表 3.1: 予測 / 例示インタフェースの比較

インタフェースを備えているものが多い。

5. 実行に関するリスクが大きい

   プログラムの実行に対して**undo** のような機構を用意しておけば間違った処理を実行しても回復が可能であるが、間違った実行にユーザが気付かない場合もあるだろうし、完全にシステムに実行をまかせてしまうと重要なデータを見逃してしまう可能性がある。

6. 予測 / 例示手法を使わない方が楽である可能性がある

   推論を行なう例示インタフェースシステムでは、システムは常に間違った推論を行なう可能性があるため、それを訂正しながら正しいプログラムを生成するための手間を考えると例示インタフェースを使用しない方が得になる可能性が常につきまとうし、推論を行なわないシステムについても同様である。

7. 予測機能がユーザの邪魔になる

   Eager や Reactive Keyboard のようなシステムではシステムの予測結果が常にユーザに提示されるが、そのような機能はわずらわしく感じられることも多い。また、常に予測機構を動作させることによりシステムの速度が低下してしまっては困る。

8. 大量のデータが必要

   多くの例示インタフェースシステムではヒューリスティクスにより少ない例から汎化を行なう工夫がされているが、高度な推論を行なおうとするシステムは本質的に多量のデータを必要とするものも多く、応用が限られてしまっている。

9. ヒューリスティクスが多用されている

   多くの例示インタフェースシステムでは、少ない例からでも一般的プログラムが生成できるようにするために各種のヒューリスティクスが用いられている。しかしこれらのヒューリスティクスは特定のアプリケーション / 特定のインタフェース手法 / 特定のユーザを仮定していることが普通であるため、広い範囲のシステムやユーザに受け入れられない可能性がある。また推論方式やヒューリスティクスについてよく知らないユーザにとってひどく使いにくくなる可能性がある。

これらの要因のうちどれかひとつにでも問題があると例示インタフェースとして成功は望めない。 2.5.2節で紹介した既存の各種の予測 / 例示インタフェースシステムにおいてこれらの要因が問題になっているかどうかを表 3.1に示す。問題が無い場合を “○”、問題が若干有る場合を “△”、大きな問題が有る場合を “×” で示してある。 2章で紹介した全てのシステムが、ひとつ以上の要因において大きな問題を持っていることがわかる。

3.3.2 予測 / 例示インタフェースの条件

前節の考察より、実用的な予測 / 例示インタフェースにおいては以下のような方針の採用が重要であると考えられる。

A. プログラムを作成可能とする。

単純な予測インタフェースは有用ではあるものの適用範囲が狭い。複雑な処理を実行させるため、予測 / 例示インタフェースシステムにはマクロ定義 / 繰返し / 条件判断などのプログラムを生成する能力が必要である。

これは要因 1を克復するための条件である。

B. ユーザの暗黙的意図を自動的にくみとる。

例示データを明示的に与える操作はユーザにとって負担が大きいので、操作履歴情報のような暗黙的例示データを活用したり、システムに例を自動生成させたりすることにより、ユーザが特に意識しなくてもプログラムを自動的に生成できるようにする。

これにより要因 2が取り除かれる。

C. システムが自動的に予測を行なって結果をユーザに提示することはせず、ユーザの指示によりシステム予測や推論を開始する。

これは要因 7を防ぐための条件である。

D. 予測結果は、ユーザが予測開始の指示を出したときにすぐに実行するか、既存のコマンド実行と同じ方式で実現し、予測実行のために特別のコマンドを覚える必要がないようにする。

これにより要因 3が取り除かれる。

E. **undo** のように予測の実行を取り消す機構を容易することにより予測の実行によるリスクを回避する。

これにより要因 5が取り除かれる。

F. 汎用で単純な予測手法を使用し、アプリケーション個有のヒューリスティクスをなるべく使用しない。

多くのヒューリスティクスにもとづいた複雑な推論は結局時間が余計にかかりかねない。

これにより要因 4, 6, 8, 9が取り除かれる。

これらを全て満足することができれば前節の問題点は全て解決される。これらの条件をひとことで記述すると「少量の例示データから汎用の単純な手法によりユーザの暗黙的な意図をプログラムとして自動的に抽出し、簡単な操作でその後の操作に適用することを可能にするシステム」ということになる。

3.4 まとめ

本章では、既存の予測 / 例示インタフェースシステムの問題点を整理し、それらを克服するために必要な条件についての考察を行なった。次章及びそれに続く 2 つの章において、本章で提案した方針に従った新しい例示インタフェースシステムを 3 種類提案し、これらの条件を満足するシステムを構築することが可能であることを実証する。

第 4 章　操作の繰り返しを利用した予測 / 例示インタフェース手法

4.1　はじめに

本章では、テキスト編集処理において有効な、操作履歴情報から繰返しを自動抽出して再利用することのできるインタフェース **Dynamic Macro** を提案し、その利点及び他の予測インタフェース手法との融合について解説する。

第 2章で紹介したように、テキスト処理を行なうための各種の予測 / 例示インタフェース手法が提案されているが、 3.3.1節で述べたように、単純な予測を行なうものはあまり効果がなく、また複雑な予測を行なうものは推論の誤りが多いために有用でないものが多かった。 Dynamic Macro は、 3.3.2節で示したような予測 / 例示インタフェースの満たすべき条件を満たしており、単純な操作で効果的な予測を行なわせることが可能である。

4.2　予測インタフェース手法 Dynamic Macro

本節では、単純な操作で効果の大きい予測インタフェースである Dynamic Macro の解説を行ない、その利点 / 適用例 / 限界 / 拡張などについて述べる。

4.2.1　キーボードマクロ

文書やプログラムを作成 / 編集するとき、文字列の検索 / 修正を繰り返す場合や、連続する行の先頭に記号を挿入する場合などのように、ユーザが同じ操作を何度も繰り返さなければならないことがよくある。前者のように使用頻度の高い操作に対してはエディタがそのような機能をあらかじめ用意しているのが普通であるが、後者のようなあまり一般的でない操作を繰り返す場合のためには**キーボードマクロ**機能が使われる場合が多い。キーボードマクロとは一連のキー操作を別の簡単なキー操作で置き換える機能である。例えば GNU Emacsのキーボードマクロではまずユーザが特別のキー操作 (通常^X ( ) によりマクロ定義開始を指示し、その後のキー操作でマクロとして定義したい編集作業を行ない、最後に別のキー操作 (通常^X)) によりマクロ定義終了を指示する。これにより定義されたマクロはマクロ呼び出しを指令するキー操作 (通常^X e) により実行される。

キーボードマクロは汎用で便利な機能であるが問題点も存在する。まず、マクロを登録 / 使用するためには上の例のように最低 3 種類の操作が必要であり、定義も呼び出しも面倒である。また、キーボードマクロを使用する場合は同じ操作を繰り返し実行することを最初から知ったうえでマクロ定義を行なわなければならないが、実際の編集作業では同じ操作をユーザが何度か行なって初めて操作の繰り返しに気づくことも多く、このとき改めてキーボードマクロを登録するのはわずらわしい。 Macro by Exampleシステム [41] や塚本らのシステム [96]では操作履歴の一部分を後で編集してマクロとして定義できるようになっているが、面倒な操作が必要になることは同じである。

4.2.2 動的マクロ生成

繰り返し作業の効率化のために、キーボードマクロに代わるものとして、ユーザの指示により操作の繰り返しをシステムが検出して繰り返しパタンをマクロとして登録し実行する**動的マクロ生成実行機能** (Dynamic Macro) を提案する [51][108][112][113]。 Dynamic Macro ではシステムは常にユーザのキー操作の履歴を記憶しており、ユーザが「繰返し実行キー」 (以降REPEATと表記) を入力すると、システムが操作履歴から繰り返しパタンを検出してそれをマクロとして登録し実行する。例えばユーザが “**abcabc**” と入力した後でREPEATを入力すると Dynamic Macro は “**abc**” の繰り返しを検出して実行し、その結果新たな “**abc**” が入力される。ここでもう一度REPEATを入力すると “**abc**” がもうひとつ入力される。また、ユーザがふたつの行の先頭に “**%**” を挿入した後REPEATを入力すると、その次の行の先頭に “**%**” が入力される。

Dynamic Macro ではユーザが覚えなければならないキーはREPEATだけであり、また繰り返し操作を実行した後でも操作をマクロとして登録できるため、前節で述べたキーボードマクロの問題点が解決されている。また、 Dynamic Macro の仕様は単純であるにもかかわらず、後の例で示すように、広い範囲の繰り返し処理に適用が可能である。

Dynamic Macro は以下のふたつの規則で構成されている。

規則 1: REPEATが入力される直前に 2 回続けて全く同じ操作列が実行されていたとき、その 1 回分をマクロとして登録し実行する。そのような繰返しパタンが複数存在するときは最長の繰返しパタンを選択する。例えば直前のキー操作が “**abccabcc**” であったときREPEATが入力されると、システムは “**abcc**” 及び “**c**” の繰返しを検出するが、 “**abcc**” の方が長いためこちらをマクロとして定義し実行する。

規則 2: 規則 1 を満たすパタンが存在しない場合、REPEATが入力される直前の操作履歴から XYX というパタンを捜し、存在すればパタン XY をマクロとして登録する。ただし一回目のREPEATでは Y のみ実行する。このようなパタンが複数存在する場合は最長の X を選択し、その中で最短の Y を選択する。例えば “**abracadabra**” を入力した後REPEATを入力した場合 “**abra**” を X, “**cad**” を Y とみなし、 “**a**” を X, “**br**” を Y とはみなさない。

ユーザが “**abcdabcd**” を入力した後でREPEATを入力した場合は規則 1 が適用され “**abcd**” が実行される。また “**abcdab**” の後でREPEATを入力した場合は規則 2 が適用されて一回目は “**cd**” が実行され、もう一度REPEATを入力すると今度は “**abcd**” が実行される。このような仕様のため、繰返し操作を行なっているどの時点においてREPEATが入力されても適切な予測実行が行なわれる。

4.2.3 Dynamic Macro 使用例

ここでは GNU Emacs 上に実装した Dynamic Macro の使用例を示す。

注釈の追加

連続する行の先頭に注釈記号を追加する例を図 4.1に示す。

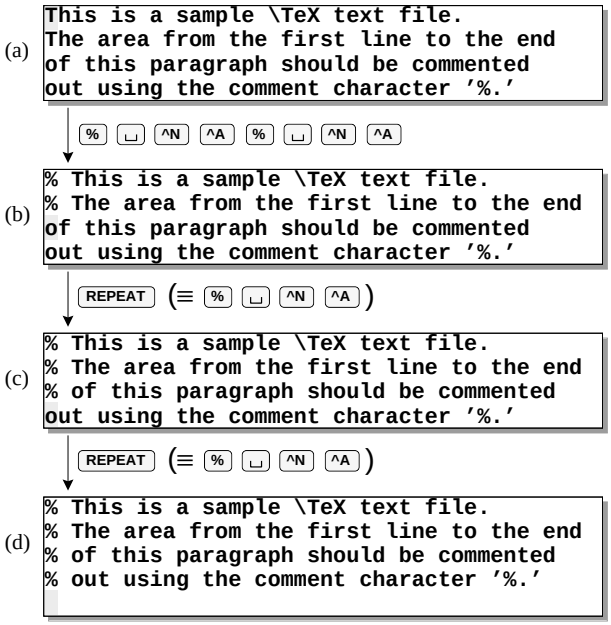


図 4.1: 各行の先頭に注釈記号を追加 (1)

図 4.1(a) が最初の状態である。ユーザが % ␣ ^N ^A % ␣ ^N ^A を入力して最初の二行に注釈記号をつけると (b) の状態になるが、ここで REPEAT を入力するとシステムは規則 1 により % ␣ ^N ^A の繰り返しを検出し、マクロとして登録し実行して (c) の状態となる。もう一度 REPEAT を入力すると (d) の状態になる。

REPEAT によりこのような結果を得るためにはユーザは必ずしも同じ操作を完全に 2 度実行する必要はない。図 4.2(a)–(d) はユーザが % ␣ ^N ^A % と入力した後 REPEAT を入力した場合の例である。

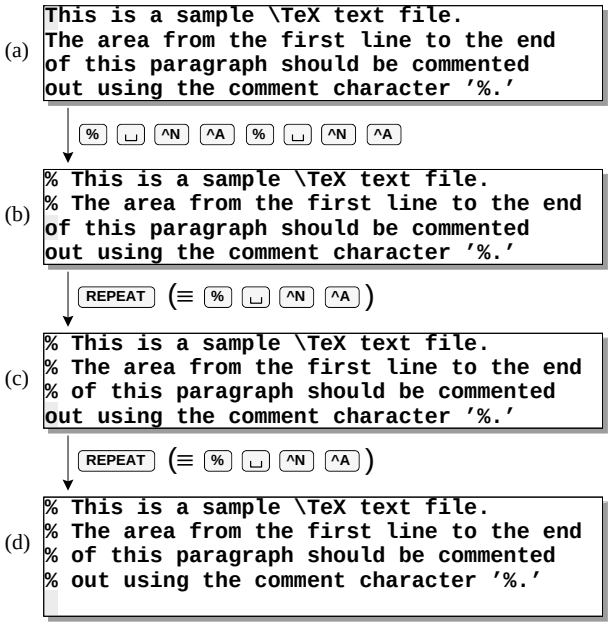


図 4.2: 各行の先頭に注釈記号を追加 (2)

この例では REPEAT の直前に同じ操作が 2 度繰り返されていないため、システムは規則 2 に従って XYX というパタンを捜し、X として % を、また Y として ␣ ^N ^A を得る。この結果システムは Y を実行し (c) の状態となる。もう一度 REPEAT を入力すると XY の両方が実行され (d) の状

態となる。このように、ユーザが操作を繰り返しているどの時点で[REPEAT]を入力しても同様の結

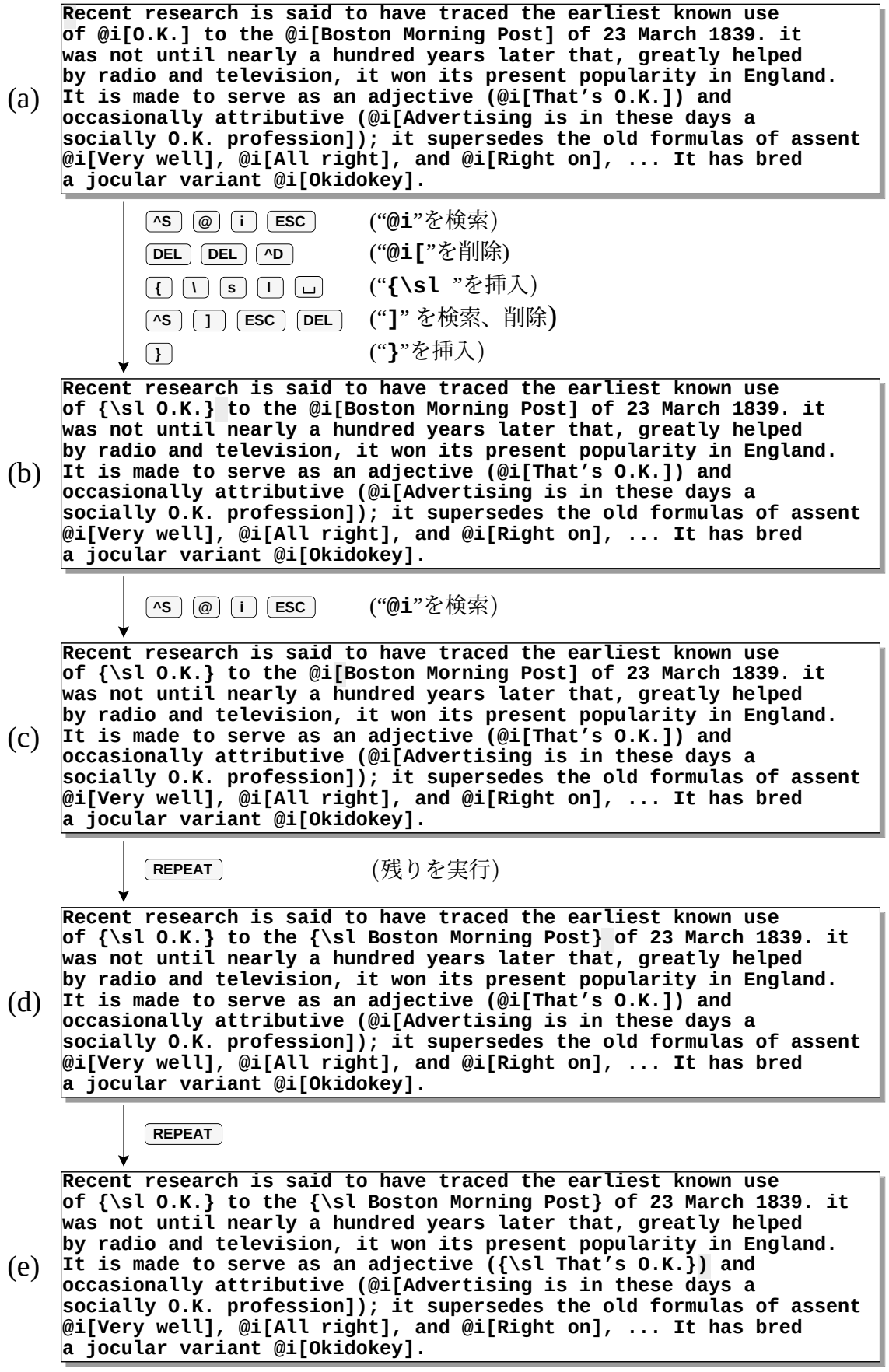


図 4.3: 文書整形指令の変更 ([68]の例)

果が得られるため、どのタイミングで[REPEAT]を入力するべきかを気にする必要がない。

パタン検索 / 置換

2.5.2節において、例示を用いたテキスト編集システムである Nix の Editing by Exampleシステムの解説を行なったが、同様のことが Dynamic Macro を用いて可能になることを示す。まず、Nix による例 [68]に Dynamic Macro を適用した例を図 4.3に示す。図 4.3(a) の全ての "**@i[***text***]**" (Scribe のイタリック指令) を "**{\sl␣***text***}**" (TEX のイタリック指令) に変換することが目標である。検索 / 修正操作を 1 回実行してから 2 回目の操作の最初の部分を実行した後[REPEAT]を入力するだけで残りの操作が自動的に行なわれる。操作前と操作後の文字列の組をシステムに提示しなければならない Nix のシステムと異なり、通常の編集作業以外のユーザ操作は[REPEAT]のみである。

関数へのコメント追加

Nix による別の例 [68]に Dynamic Macro を適用した例を図 4.4に示す。ここでは Lisp プログラム (a) のすべての関数の先頭にコメントを追加し (b) のようにすることが目標である。長い変換操作が必要となっているが、繰り返しの 2 回目の最初の操作を行なった後で[REPEAT]を入力すると (b) のような結果が得られ、以後[REPEAT]を入力する度に次の関数の先頭にコメントが挿入される。

(a)

```
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
(define (halts f)
  ( .... ))
```

("**(define**"を検索)
(行末までの文字列をテンポラリバッファに格納)
(1行上に"**;***␣**"を挿入)
(テンポラリバッファの内容を挿入)
("**;*** ---** …**;***" を挿入)
("**(define**"を検索)

[REPEAT]

(b)

```
;*** (factorial n)
;*** ----------------------------------
;*** <perspicuous description here>
;***
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
;*** (halts f)
;*** ----------------------------------
;*** <perspicuous description here>
;***
(define (halts f)
  ( .... ))
```

図 4.4: Lisp 関数へのコメント追加 ([68]の例)

4.2.4 Dynamic Macro の利点

Dynamic Macro には以下のような利点がある。

使用法が単純 使用するキーは[REPEAT]のみである。また繰り返し操作においてどの時点で[REPEAT]を入力しても同様の結果が得られる。

広い範囲に適用可能 単純な機構にもかかわらず、前節の例のような複雑な繰り返し操作にも適用できる。

実装が簡単 システムは[REPEAT]が入力されない限り操作の履歴を保持するだけでよいし、規則 1、規則 2 は高速に実行することが可能である。履歴保持のオーバーヘッド及び計算コストについては 4.5.2節で詳しく述べる。

ユーザの邪魔をしない 普段システムは操作の履歴をとっているだけなので、そのために動作が遅くなったりすることはない。また[REPEAT]を入力しない限り予測は実行されないので予測機能を使わないユーザの邪魔になることがない。

汎用 Dynamic Macro はテキストエディタ以外にも適用可能である。

4.2.5 Dynamic Macro の問題点

Dynamic Macro の最大の問題点はユーザの期待と異なる予測を実行する可能性があることである。例えば、[TAB][TAB] a b c [RET] [TAB][TAB]の後で[REPEAT]を入力すると規則 1 が適用されて[TAB]が実行されるが、ユーザは a b c [RET]を期待しているかもしれない。規則 2 が規則 1 より優先するようにすればこの場合ユーザの期待どおりになるかもしれないが、“**bab** *long-forgotten-sequence* **abab**” のような操作の後で[REPEAT]を入力すると、“**ab**” のかわりに忘れられた昔の操作列が実行されてしまうことになり望ましくない。また、“**abracadabra␣ab**” の後で[REPEAT]を入力した場合、ユーザは “**racadabra␣**” を期待しているかもしれないが、規則 2 により “**ra␣**” が実行される。 Dynamic Macro は予測手法のひとつであるため、どんな場合でもユーザの期待通りの動作をするような仕様を設定することは不可能であるが、この問題は新たに「予測キー」を導入することにより解決する。

4.3 Dynamic Macro の拡張

4.3.1 既存の予測手法との融合

2.4章で述べたように、テキストベースのプログラムにおいては各種の予測手法が広く使用されている。これらの予測手法に Dynamic Macro を加えたものを図 4.5に再掲する。

名前	システム	予測に使用する情報
dabbrev	Emacs	文書中の文字列 (動的辞書)
completion	Emacs	コマンド名 / ファイル名など (静的 / 動的辞書)
罫線引き	Emacs	直前のキー及びカーソル直下の文字
かな漢字変換	各種ワープロ	かな漢字辞書
“**.**”	vi	直前のコマンド
“**!!**”	csh	直前のコマンド
“**!**(string)”	csh	コマンド履歴
“ESC **l**”	tcsh	ファイル名
Reactive Keyboard	shell	キー入力の統計情報
Dynamic Macro	(Emacs)	繰り返し操作

図 4.5: 各種の予測手法の比較 (再掲)

これらの多くのものは、ユーザの指示により何らかの方法で次の操作を予測してそれをユーザに提示するという型式になっているため、予測手法毎に異なるキーを割りあてず、順番に各種予測を適用していくことにより、単一の「予測キー」 (以降PREDICTと表記) を用いて実装することができる。次のユーザ入力をファイル名から予測する手法と辞書エントリから予測する手法を併用した例を図 4.6に示す。

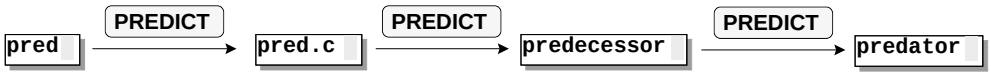


図 4.6: ファイル名からの予測と辞書を用いた予測の併用

PREDICTが最初に入力された場合はその時点で一番もっともらしい方式により次の操作を予測してユーザに提示する。続けてPREDICTが入力された場合は前の予測を撤回して次の予測をユーザに提示する。

PREDICTは連続して押されたときのみ次候補を提示し、それ以外のキーが押されたときは候補が確定されたとみなすので、これを利用して図 4.7のように候補を段階的に絞っていくことができる。

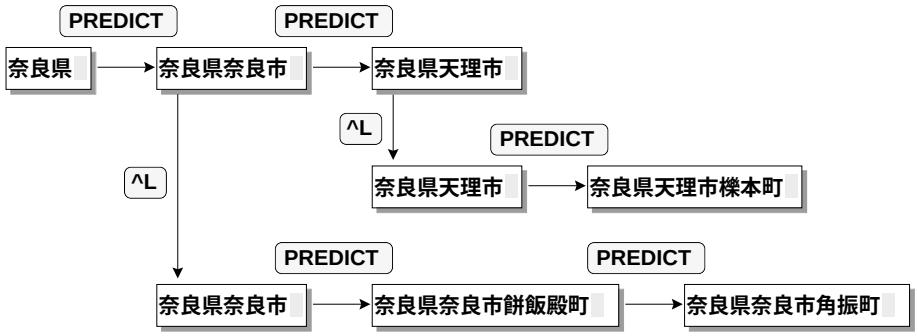


図 4.7: PREDICTによる候補空間の探策

ここでは図 4.8のような地名辞書の使用を仮定している。

奈良県奈良市	奈良市餅飯殿町	天理市櫟本町
奈良県天理市	奈良市角振町	天理市丹波市町
奈良県生駒市	奈良市不審ヶ辻子町	天理市守目堂町
...	...	...

図 4.8: 地名辞書の内容

「奈良県」の後でPREDICTを押し続けると「奈良市」「天理市」「生駒市」... を予測し続けるが、適当な場所でPREDICT以外のキーを押すとそこで予測が確定して新たな予測が開始される。

4.3.2 予測キーと繰返しキーの併用

PREDICTとREPEATを併用することにより、前節で述べた Dynamic Macro の問題点が解決されるだけでなく、さらに複雑な各種の予測手法を使うことが可能になる。 Dynamic Macro の最大の欠点は、予測がユーザの期待と異なっていたとき修正不可能なことであったが、PREDICTを用いることにより、期待と異なる予測が得られた場合の動作を変更することができる。前節で述べ

たように、TAB TAB a b c RET TAB TABの後でREPEATを入力すると規則 1 が適用されTABが予測されるが、この状態でPREDICTを入力すると次候補である規則 2 が適用されるようにすると、a b c RETが予測されるようになる。これがユーザの期待と一致する場合は、この後再びREPEATを入力することによりこの予測結果を繰り返させることができる。この様子を図 4.9に示す。

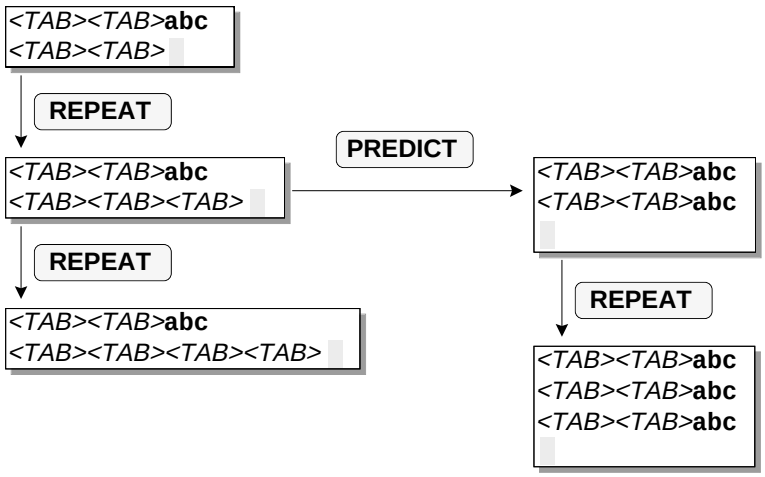


図 4.9: PREDICTによる予測手法の切り替え (1)

また前節で示した別の例に対して同じ手法を適用した結果を図 4.10に示す。

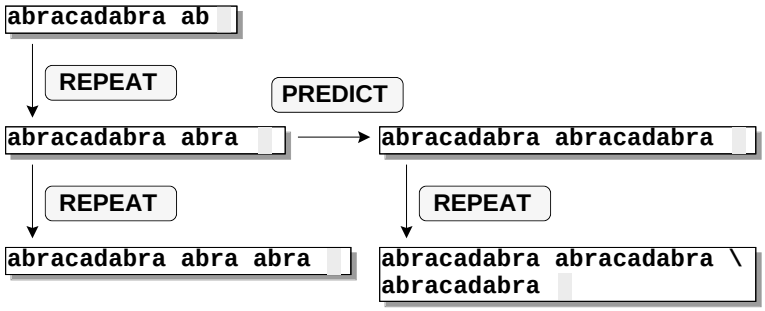


図 4.10: PREDICTによる予測手法の切り替え (2)

またREPEATの意味を拡張し、PREDICTの後にREPEATが入力された場合は直前の予測方式を繰り返すことにすると、図 4.11のように、PREDICTで予測手法を選択した後REPEATでその実行を繰り返させることができる。

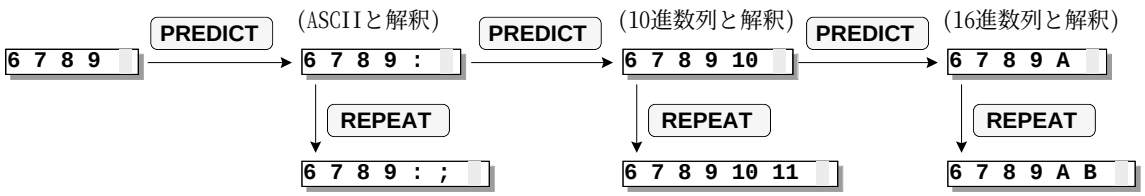


図 4.11: 予測手法を選択後同じ予測を繰り返す

最初のPREDICTが入力されたときシステムはその前の入力列を ASCII 文字列と判断して “**:**␣” を予測し実行する。このときシステムは “ASCII 文字列予測モード” になっているため、ここでユーザはREPEATを入力することにより次の ASCII 文字 “**;**␣” を予測させることができる。最初の予測がユーザの期待と異なっていた場合、ユーザがもう一度PREDICTを入力するとシステムは今度は

前の入力列を 10 進文字列と判断して “10 進数予測モード” となり “**10**␣” を予測し実行する。これがユーザの期待と一致していた場合は REPEAT を入力することにより “**11**␣”, “**12**␣”, ... を順に予測させることができる。

同様の手法を用いて図 4.12のように LaTeX 文書の作成に応用することもできる。システムは PREDICT が押されると “**\begin**” から **itemize** 環境であると判断して **itemize** モードに状態遷移しつつ “**\begin{itemize}**” ... “**\end{itemize}**” を予測する。このモードにおいて REPEAT が入力されるとシステムは “**\item**” を予測する。ユーザの期待が **itemize** でなかった場合、ユーザが最初の PREDICT に引き続いてもう一度 PREDICT を入力すると、今度はシステムは **document** モードに遷移しつつ “**\begin{document}**” ... “**\end{document}**” を予測する。ここで REPEAT が入力されるとシステムはモードの知識により “**\section{}**” を予測する。

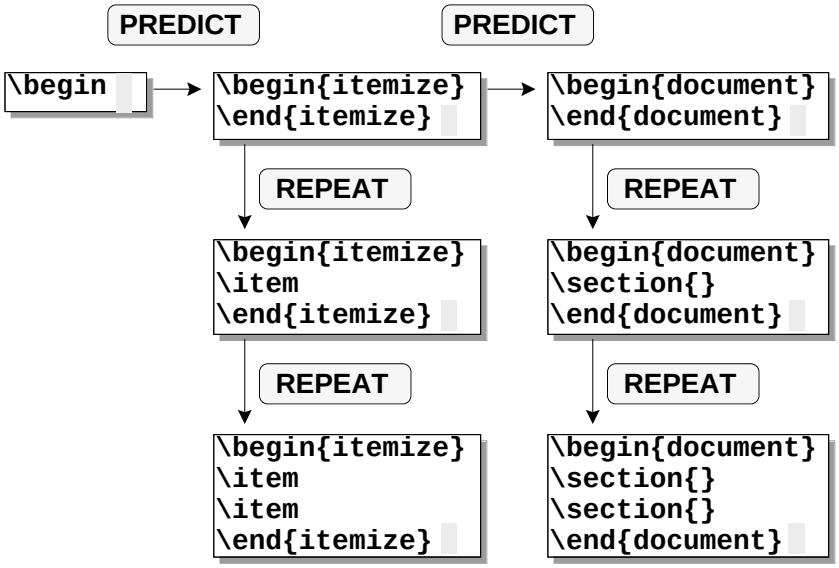


図 4.12: LaTeX プリミティブの予測

この REPEAT と PREDICT の機能を整理すると図 4.13のようになる。

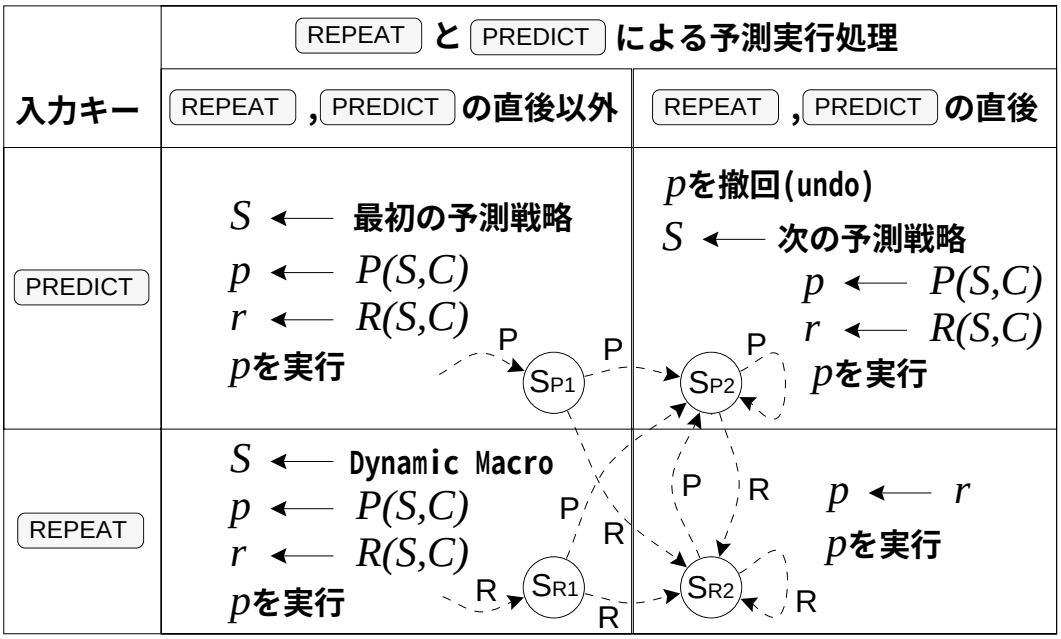


図 4.13: REPEATとPREDICTによる状態遷移

ここで、 S は予測戦略、 C は現在のコンテキスト、 P,R はそれぞれPREDICTとREPEATが押されたとき実行すべき操作列を S 及び C から計算する関数、 p,r は計算された操作列を保持するための変数を示す。 SP1, SP2, SR1, SR2 はPREDICTとREPEATが押されたときの遷移状態を示し、図 4.13の遷移図のように状態遷移する。例えば初期状態においてREPEATが入力されるとシステムは状態 SR1 に遷移し、予測戦略として Dynamic Macro を選択し、これにもとづいて操作列を予測し実行する。そこでPREDICTが入力されると状態は SP2 に遷移し、前の予測を撤回して新たな予測を実行する。このようにPREDICTとREPEATを併用していろいろな順番で使用することにより、 Dynamic Macro の欠点が解消されかつ各種の予測手法を適用することができる。

4.4 Dynamic Macro の評価

本システムを多くの Emacs ユーザに配付して 1 年以上にわたり反応を調査した。また数人のユーザに予測操作のログを取ってもらうことにより予測の的中率や実際の使われ方を調査した。この結果は以下のようになった。

- 評判はおおむね良好であった。特にキーボードマクロを普段あまり使っていないようなユーザには好評であった。
- 逆に Emacs のエキスパートは本システムをあまり有難いと思わないようであった。この原因は、そのようなユーザはキーボードマクロを普段から多用しているため使用を面倒だと思わないためと、各種の繰り返し操作に対応する機能 / 関数について熟知しているので自分でマクロを定義する必要があまり無いためであると思われる。
- 予測の的中率 (1 回目のREPEATまたはPREDICTによるシステムの予測がユーザの期待と一致する割合) はユーザによりかなり異なっていた。最高は 100% であったが、このユーザはごく稀に (月に数回) しか予測機能を使用せず、またその際には非常に慎重に入力を行なっていた。逆に、間違った予測には**undo** で対処すればよいと考えるユーザは気軽に予測インタフェースを使用するが、的中率は低かった。平均すると的中率は約 85% であった。

- REPEATにより予測されるキーストローク長の平均値は約 5 であり、キーストローク数が 3 または 4 の予測が最も多かった。これは Dynamic Macro による予測方式が、短いキー操作列の繰り返しに対し効果的であることを示している。長いキー操作列の繰り返しに対してはキーボードマクロが有効であり、ごく短いキー操作列の繰り返しに対してはマクロ定義が無意味であるが、 Dynamic Macro はその中間程度のキー操作列の繰り返しに対し有効であることがわかる。

4.5 議論

4.5.1 キーストローク数の削減効果

4.4節で述べたように、実際の使用場面では、 3 〜 4 キーストロークが繰り返される場合に Dynamic Macro が使用される場合が多いが、このような場合にどの程度キーストロークが削減されているかを、典型的な場合について計算する。

Dynamic Macro が効果的に働く代表的な例として、続いた n 行を 2 文字ずつ字下げする場合のキーストローク数を、マクロ機能を使用しない場合 / キーボードマクロを使用する場合 / Dynamic Macro を使用する場合について計算すると以下のようになる。

- マクロを使用しない場合

  { 行頭 (^A), 空白文字 (␣) × 2, 次行 (^N)} × n

- キーボードマクロを使用する場合

  定義開始 (^X (), 行頭 (^A), 空白文字 (␣) × 2, 次行 (^N), 定義終了 (^X)), 定義呼出 (^X e) × $n - 1$

  定義の開始 / 終了 / 呼出をそれぞれ 1 キーに割り当てることもできる。

- Dynamic Macro を使用する場合 (1) (手順を完全に 2 回繰り返した場合)

  行頭 (^A), 空白文字 (␣) × 2, 次行 (^N), 行頭 (^A), 空白文字 (␣) × 2, 次行 (^N), REPEAT × $n - 2$

- Dynamic Macro を使用する場合 (2) (手順を途中まで繰り返した場合)

  行頭 (^A), 空白文字 (␣) × 2, 次行 (^N), 行頭 (^A), REPEAT× $n - 1$

$n = 5, 20, 100$ についてこれらのキーストローク数を計算すると図 4.14のようになる。

n	5	20	100
マクロ不使用	20	80	400
キーボードマクロ使用 (Emacs 標準キー割当て)	16	46	206
キーボードマクロ使用 (マクロ関連キーに 1 キー割当て)	10	25	105
Dynamic Macro 使用 (手順を完全に 2 度繰り返した場合)	11	26	106
Dynamic Macro 使用 (手順を途中まで繰り返した場合)	9	24	104

図 4.14: 連続行の字下げに要するキーストローク数

繰り返しの回数が多いほどマクロを使用する効果は大きいが、このような単純な操作をわずかな回数繰り返す場合でも、マクロの使用により 50% 程度のキーストローク数削減効果がある。

キーボードマクロの開始 / 終了 / 呼出にそれぞれ 1 キーを割り当てた場合はキーボードマクロと Dynamic Macro のキーストローク削減効果は同程度になるが、 Dynamic Macro の方が使用するキー種が少なくてすむうえに繰返し単位を意識する必要が無いため、このような単純な繰り返し操作の場合は Dynamic Macro が効果的と考えられる。

4.5.2 履歴保持のオーバーヘッドと予測のための計算コスト

Dynamic Macro による予測を行なうためにはシステムは常にユーザの操作履歴を記憶しておく必要がある。 GNU Emacs は常に 100 個のキーストローク履歴を保持しており **(recent-keys)** 関数で参照することができるため、これを用いて Dynamic Macro を実装しているが、このとき規則 1 では 50 キーストローク、規則 2 では最高 99 キーストロークまでの繰り返しパタンを認識可能ということになる。実際には 50 キーストロークにも及ぶ操作を繰り返すことは稀であるから履歴はこの程度保持しておけば充分であり、そのためのオーバーヘッドは小さい。

現在の実装では規則 2 の X の長さを 1 からひとつずつ増やしながら XYX というパタンを検索しているため、規則 1/ 規則 2 を満たす文字列を捜すのに最悪 $O(n^2)$(n は履歴文字列長) 時間が必要であるが、実際の使用場面においては繰り返されるキーストローク長は 4.4節で述べたように平均して 3 から 4 程度であり、 X が 10 ストローク以上となることは稀であるため、使用に関して予測の遅さが問題になることはほとんどない。また Dynamic Macro では予測のために時間がかかるのは REPEAT を最初に押したときのみで、続けて REPEAT を押したときは予測のための時間はかからないため、予測機能がユーザの邪魔になることはない。これに対し、各キーの入力時にキーストローク履歴の部分文字列をトライ構造などで記憶しておけば規則 1/ 規則 2 の実行は高速になるが、あらゆるキーストロークに対しこのような処理を行なうと実行が遅くなり操作の邪魔になる可能性があるため得策ではないと考えられる。

4.5.3 かな漢字変換との類似性

PREDICT はかな漢字変換方式の日本語入力システムにおける「漢字変換キー」または「次候補キー」と似た動作をする。実際「漢字変換キー」を図 4.6と同じように使ってローマ字文字列をかな文字列、漢字文字列に順次変換していくようなかな漢字変換システムも存在する。このような性質のため PREDICT はかな漢字変換システムに慣れたユーザにとってなじみやすいし、また本システムのような予測インタフェースをかな漢字変換システムと融合して使用することも可能である。

4.5.4 Dynamic Macro が効果的に働く理由

グラフィックインタフェースに予測を用いる研究が近年盛んに行なわれているが、グラフィックインタフェースにおいてはユーザの意図を正しく理解することは本質的に困難であるため [7]、システムが間違った推論を行なわないようにするため問題を簡単化したりヒューリスティクスを使ったり頻繁にユーザに問いあわせを行なったりしているものが多い [98]。また、 Pursuit システム [57] のように、操作例から自動的に生成されたプログラムをビジュアルプログラミングによりユーザが簡単に編集可能とすることによりこのような問題を解決しようとしているものもある。これに対し Dynamic Macro では、テキスト操作に限ってはいるものの簡単な規則で予測インタフェースが成功している。この大きな理由のひとつとして、 Emacs ではユーザの意図と操作がほぼ 1 対 1 に対応している点がある思われる。 Emacs では例えばユーザがカーソルを行頭に動かしたい場合は ^A を入力するし、左に何文字か移動させたい場合は ^B を文字の数だけ入力する。このように、 Emacs では同じようにカーソルを左に動かす場合でもその意図によりユー

ザの操作は異なっているのが普通である。これに対しグラフィックの直接操作インタフェースでは、例えば図形をマウスで左にドラッグしたとき、それが画面の右に場所を空けるためなのか、移動量が重要なのか、別の図形と位置を揃えるためなのか、などが操作のみからは判断がむずかしいため推論が困難になっているわけである。 Emacs ではユーザの意図を反映した機能が別々の操作としてあらかじめ用意されておりそれをユーザが実際に区別して使用するため、システムがユーザの意図をくみとって予測を行ないやすいということができる。この点を考慮すると、グラフィックインタフェースにおいてもユーザがその意図を常に無意識にシステムに知らせ続けることができれば効果的な予測インタフェースが構築できると考えられるが、ポインティングデバイスでカーソルを直接任意の文字位置に移動させることのできるような文書エディタでは、文字移動の意図を自動的に判定しその繰返しを抽出することは困難と考えられる。

4.5.5 Dynamic Macro の他システムへの適用

Dynamic Macro における繰返し操作の抽出手法は、 Emacs のような編集システムだけでなく、あらゆるインタフェースに適用可能であるが、効果には大きな違いがある。 UNIX のシェルにおいて Dynamic Macro と同じ手法で操作の予測をする実験を行なったところ、以下のような理由により Emacs の場合に比べてはるかに有効性が劣ることが判明した。

- シェルでは全く同じ操作を何度も繰り返す機会が少ない

  シェル上のコマンド操作では、ある程度複雑な処理を行なうことが多いため、文書編集の場合のように「ほとんど頭を使わず手だけが単純作業を繰り返している」といったことは少ない。 Dynamic Macro はこのような非常に単純な作業の繰り返しの効率化には有効であるが、知的な作業の繰り返しにはあまり役にたたない。

- **undo** 不能な操作列が予測される可能性がある

  コマンド行では**undo** 不能な操作が多い。例えばファイルの消去は**undo** 不能な操作である。このような操作の自動実行は危険であるため実行前にユーザの確認をとる必要があるが、ユーザが余分な操作をしなければならないため、予測のメリットが薄れてしまう。

- 他の有用な予測手法がすでに広く使われている

  図 2.2に示したように、シェルでは各種の単純で有用な予測手法がすでに広く実用になっており、新たな予測機構を導入するメリットが少ない。

以上のように、 Emacs とシェルの操作は似た点が多いいもかかわらず、有用性は大きく異なっている。予測 / 例示インタフェースシステムは、ターゲットとなるシステムの特性によくマッチしていなければ有用なものとはならないことがわかる。

4.5.6 予測 / 例示インタフェースの問題点の検証

Dynamic Macro において 3.3.1節で述べた問題点が克服されているかどうかを問題点毎に検証する。

1. プログラムを作成できない

   Dynamic Macro は、あらゆる操作列をマクロとして定義可能である。

2. プログラム生成のための指示が面倒

   Dynamic Macro では REPEAT と PREDICT のふたつのキーしか使用されず、予測のための手間は最小限に押さえられている。

3. プログラム実行のための指示が面倒

   最初にREPEATを押したとき、プログラムは生成と同時に実行される。

4. 正しいプログラムの作成が困難

   4.4節で述べたように、予測結果がユーザの期待と一致する確率は非常に高い。

5. 実行に関するリスクが大きい

   間違った予測が行なわれたときは**undo** によりすぐに取り消すことができるためユーザが不安を感じることはない。

6. 例示 / 予測手法を使わない方が楽である可能性がある

   予測を指示する手間が非常に小さいため、このような可能性は少ない。

7. 機能がユーザの邪魔になる

   REPEATを押さない限り予測は実行されないので、予測機能を使用しないユーザの邪魔になることはない。システムが自動的に記憶している履歴データのみを予測に使用しているため予測機能の追加によるオーバーヘッドは無い。

8. 大量のデータが必要

   システムの記憶している履歴データは 100 キーストロークのみであり、それのみを使用して充分実用的な予測が行なわれる。

9. ヒューリスティクスが多用されている

   4.2.2節に述べたふたつの単純な規則のみが予測に使用される。これらの規則はアプリケーションによらず一般的に使用できる。

以上のように、 3.3.1節で述べた全ての点に関して問題が解決していることがわかる。これを表 3.1に従って表にすると表 4.1のようになる。

	プログラム不可能	定義時の指示が面倒	実行時の指示が面倒	正しい推論が困難	実行リスクが大	時間的リスクが大	勝手な実行が邪魔	大量データが必要	ヒューリスティクスが過度
Dynamic Macro	○	○	○	○	○	○	○	○	○

表 4.1: 予測 / 例示インタフェースの問題点の Dynamic Macro における解決

4.6 まとめ

本章では、操作履歴情報から繰返しを自動抽出して再利用することのできる予測 / 例示インタフェース Dynamic Macro を提案した。 Dynamic Macro では、繰返しを抽出する単純な規則によってユーザの操作履歴から暗黙的な繰返し操作の意図を抽出し、単純な操作で再利用することが可能である。

第 5 章　依存関係を利用した予測 / 例示インタフェース手法

5.1　はじめに

本章では、操作履歴情報やファイルの参照情報を利用して操作間の依存関係を自動抽出し再利用する手法を提案し、それを応用した **Smart Make** システムについて述べる。また、依存関係を自動抽出するためのその他の単純で有用な手法について述べる。

5.2　依存関係のインタフェースへの利用

処理の手続きを記述するかわりに規則や制約を宣言的に記述することが有効な場合が多い。例えばプログラミングにおいて宣言的記述が可能であることは、論理型言語や関数型言語の大きな利点のひとつとなっている。ユーザインタフェースにおいても宣言的な依存関係の記述が有効である場合が多い。例えば表計算ソフトウェアでは、セル間の関係を宣言的に数式として表現することによりわかりやすく表の自動計算を行なわせることができるし、図形エディタにおいても「図形 A と図形 B は同じ X 座標を持つ」といった関係を宣言的に定義することができると便利である。また、 UNIX などで広く使われているプログラム開発支援システム Make は、システム開発に必要なファイル間の依存関係を特別のファイル Makefileにおいて宣言的に記述することにより自動的なシステム構築を支援するものである。

5.2.1　依存関係の自動抽出

前述のようなシステムでは依存関係はユーザが陽に指定する必要がある。しかしユーザの操作履歴やアプリケーション知識を用いることにより依存関係を単純な方法で自動抽出することが可能な場合がある。以下にそのような場面を示す。

- 操作履歴の再実行

  電卓などを使用して、ある値をもとに様々な計算を行なった後で、値を変えてもう一度同じ計算を行なおうとした場合、その値と計算結果との関係として前回の操作履歴をそのまま使用することができる。このような場合は、何らかの方法により前回の操作履歴を切り出すことにより、値と計算結果の依存関係を抽出し再利用することができる。

- ファイル変換操作の再実行

  ネットワーク上で配付されているソフトウェアを入手して自分の計算機で実行することが近年頻繁に行なわれているが、ソフトウェアの格納されている場所を捜したり、それを自分の計算機に転送して変換を行なったり、自分の計算機用に修正を行なったりするためにはかなりの手間がかかるのが普通である。手間をかけてインストールを行なった後、そのソフトウェアがバージョンアップがされた場合には、普通はこれらの作業を全て最初から

やり直さなければならないが、操作履歴をうまく覚えておけば、そのソフトウェアをどこからどのように入手してどのような変換を行なったかという過程を再現することができるはずである。同様に、前回と同じ条件でシミュレーションを再実行させたりコンパイルやリンクなどの作業を行なわせることも可能になる。

- 繰り返し予測から依存関係を抽出

前章の Dynamic Macro を使用すると、**1**, **2**, **3** といった数字の列から**4**, **5**, **6**, ... といった数字の列を予測させることができたが、予測されたおのおのの数字において、それがどのような予測により生成されたかを依存関係として記憶しておけば、初期値を後で変更した場合でも自動的に再計算を行なわせることができる。例えば最初の数を**2** に修正したとき残りの数列を自動的に**3**, **4**, **5**, ... とすることができる。

以上のような例はすべて、操作履歴を例示データとして使用して依存関係を示すプログラムを自動生成することにより実現することができる。

5.3 Smart Make

前節で述べたように、操作履歴を解析することにより、操作間の依存関係をある程度自動的に抽出することができると考えられる。本節では、 UNIX などのプログラム開発システム Make で使用されるファイル依存関係記述を自動的に抽出するシステム Smart Make について述べる。

5.3.1 Make

Make[2] は、 UNIX などでよく用いられているプログラム開発ユティリティのひとつである。大規模なプログラム開発を行なう場合、複数のソースファイルを使って分割コンパイル/ リンクにより目的プログラムを作成するのが普通であるが、ひとつのソースファイルを修正したときどれだけ再コンパイルが必要になるかはファイル間の依存関係によって変わってくる。例えばオブジェクトファイル**`prog1.o`** がソースファイル**`common.h`** と**`prog1.c`** から作成され、オブジェクトファイル**`prog2.o`** がソースファイル**`common.h`** と**`prog2.c`** から作成され、実行ファイル**`a.out`** がオブジェクトファイル**`prog1.o`** と**`prog2.o`** から作成されるような場合、**`prog1.c`** を修正した場合は**`prog1.o`** だけを再コンパイルしてからリンクを行なえばよいが、**`common.h`** を修正した場合は**`prog1.o`** と**`prog2.o`** の両方を再コンパイルする必要がある。これらのファイルとコマンドの関係は図 5.1のように示すことができる。

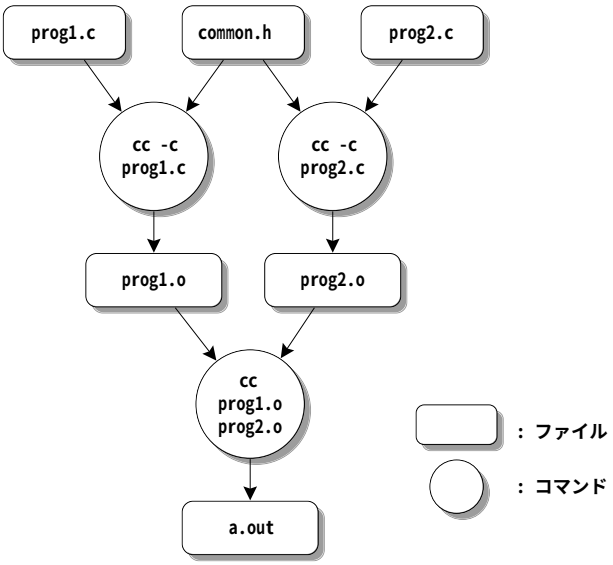


図 5.1: ファイルの依存関係を示すグラフ

ファイル間の依存関係をあらかじめ Makefileという特別のファイルに記述しておくことにより、ファイルが更新されたときに必要となる操作を自動的に実行させるためのツールが Make である。あるコマンドにより新たにファイルが作成されるとき、それがどのファイルに依存しているかを Makefile に記述する。

依存関係は図 5.2のような書式で記述される。

< ターゲットファイルリスト > **':'** < 依存ファイルリスト >
[TAB 文字] < コマンド 1>
[TAB 文字] < コマンド 2>
...

図 5.2: Makefile の書式

前の例の場合、ファイル**prog1.o** はファイル**common.h** と**prog1.c** に依存しており、コマンド**cc -c prog1.c** により作られるということを以下のようなテキストファイルで記述する。

```
prog1.o: common.h prog1.c
        cc -c prog1.c
```

同様に、ファイル**prog2.o** がファイル**common.h** と**prog2.c** に依存していること及びファイル**a.out** がファイル**prog1.o** と**prog2.o** に依存していることは以下のように記述する。

```
prog2.o: common.h prog2.c
        cc -c prog2.c
a.out: prog1.o prog2.o
        cc -o a.out prog1.o prog2.o
```

Make は Makefile の記述からファイル間の依存関係を知り、依存関係が満たされていない場合 (ターゲットが依存ファイルよりも古い場合) は対応するコマンドを起動する。

5.3.2 Makefile の自動生成

Makefile はプログラム開発を行なおうとするユーザが自分で記述するのが一般的であるが、一種の仕様記述言語であるため初心者には作成がむずかしい。また、最初から大規模なシステムを構築する予定であればきちんとした開発手順にのっとって Makefile を使用するであろうが、プロトタイプ的な簡単なプログラムを試作するような場合は、最初は単純にコマンド行からの指令によりコンパイルを行なうのが楽であったとしても、プログラムが複雑になるにつれて Make を使用する方が効率が良くなるような時点が存在するはずであり、この過程において 3.2.3節で述べたようなジレンマが生じてしまう。

```
% vi test.c            テストプログラムを作成
% cc test.c            コンパイル
% a.out                実行
% vi test.c            プログラム修正
% cc test.c            再コンパイル
% a.out                実行
...
% cp test.c test.h     プログラムを分割
% vi test.c test.h     プログラム修正
% cc test.c
% vi test2.c           プログラムを追加
% cc test.c test2.c
% a.out
...
% vi test.c
% cc -c test.c         部分コンパイル
% cc test.o test2.o    リンク
...                    Make の使用を検討し始める
```

図 5.3: 単純なプログラムの段階的開発例

図 5.3に示したようなプログラムの試作において、ユーザは最後の時点の近辺で Just-in-time Programming として Makefile を作成するべきかどうかの判断を行なわなければならない。 Makefile の作成に慣れていないプログラマでも Make のを利用できるようにするために、なんらかの手段により Makefile が自動生成されると便利である。

Makefile を自動生成する試みとして、**mkmf**[1]というプログラムが提案されている。**mkmf** は、ディレクトリの中の C プログラム及びヘッダファイルを解析して依存関係を調べ、 Makefile を自動生成する。例えばディレクトリ内に**test.c** と**test.h** というファイルが存在し、**test.c** 内に**#include "test.h"**という行が含まれていれば下のような Makefile を生成する。

```
test.o: test.c test.h
        cc -c test.c
```

mkmf の使用によって Makefile をある程度自動的に生成することができるが、**mkmf** はファイルの有無及びプログラムの字面の解析のみを使用するので実用的にはあまり役にたたない。

[1]“Make Makefile” の略

たとえば、プログラム開発に関係無いファイルがディレクトリ内に含まれていても、それに気付かず解析を行なってしまうし、コンパイルオプションなどはユーザが後から指定してやる必要がある。そもそも、沢山のファイルを使用してプログラム開発を行なうような局面であれば最初から Makefile を使うのが普通であるから**mkmf** が実際に役にたつことはまれであると思われる。図 5.3のような局面においても有用であるようなシステムが望まれる。

5.3.3 テンプレートによる依存関係の抽出

Makefile に必要な依存関係は、図 5.3のような操作履歴を解析することによってある程度抽出することが可能である [97] 例えば、一般の UNIX システムにおいて

```
cc test.c
```

というコマンドにより C プログラムファイル**test.c** をコンパイルすると、C コンパイラ**cc** の仕様により実行ファイル**a.out** が新たに生成される。ここで、**cc** の仕様の知識があれば、前記のコマンドがプログラムファイル**test.c** から実行ファイル**a.out** を生成するという意図を持ったものであることが自明なので、この制約を以下のような Makefile として表現することができる。

```
a.out: test.c
        cc test.c
```

使われるすべてのコマンドに対して依存関係のテンプレートを用意しておけば、このようにコマンド行からその意図を抽出することができるが、この方式は以下に示すように問題が多い。

- 数多くのテンプレートが必要

  コマンド毎に仕様は細かく異なっているためテンプレートが多数必要であり、またシステムや環境により微妙に動作が変わることもあるので、実際のコマンドと全く同じように引数を解釈したり使用ファイルを特定したりすることが困難である。

- 新しいコマンドや独自のコマンドに対応できない

  コマンド知識データベースに含まれていない特殊コマンドや新しく作成したコマンドに対して適用できない。

- コマンド文字列やコマンド知識に明示的に出現しない依存関係は抽出できない

  例えば上記の**test.c** が**#include** 文により**test.h** を参照している場合でも、**test.h** はコマンド知識にもコマンド文字列にも出現しないため、依存関係を抽出できない。

5.3.4 ファイルの更新 / 参照時間にもとづく依存関係の抽出

前節で述べたように、制約関係の抽出に各コマンドに対応した知識を用いることはあまり実用的ではないので、本節ではもっと単純で汎用性の高い手法について述べる。

図 5.3のコマンド列を実行すると、各コマンドの起動及びファイル**test.c**, **a.out** の参照 / 更新は以下のような順で行なわれる。

1: “**vi test.c**” 起動
2: **test.c** の参照
3: **test.c** の更新
4: “**cc test.c**” 起動
5: **test.c** の参照
6: **a.out** の更新
...

図 5.4: コマンド起動とファイル参照 / 更新の順番

多くのファイルシステムでは各ファイルの最終参照時刻及び最終更新時刻を保存するようになっているので、各コマンド起動後のファイルの参照 / 更新時刻を調べることにより、各コマンドがどのファイルを参照してどのファイルを生成したかを判定することができる。例えばコマンド “**vi test.c**” の起動によりファイル**test.c** が参照されかつ更新されていることがわかれば、ファイル**test.c** の依存関係は以下のような Makefile で表現することができる。

```
test.c:
        vi test.c
```

また、コマンド “**cc test.c**” の起動によりファイル**test.c** が参照され、ファイル**a.out** が更新されることが図 5.4の 1 〜 3 行目よりわかるため、これらのコマンドとファイルの間の依存関係は以下のように表現することができる。

```
a.out: test.c
        cc test.c
```

このようにコマンド起動の度にファイルの参照 / 更新時刻を調べることにより、ファイル、コマンド間の依存関係を容易に抽出することができる。依存関係の記述は以下のような簡単な規則で生成できる。

< 更新されたファイルのリスト > **':'** < 参照されたファイルのリスト >
[TAB 文字] < コマンド >

図 5.5: Makefile の生成規則

同じファイルがひとつのコマンドで参照 / 更新されている場合は参照されたファイルのリストには加えない。

5.3.3節で述べたテンプレートを用いる方式に比べ、ファイルの参照 / 更新時刻を用いる手法は以下のような利点がある。

- テンプレートが不要
- 新しいコマンドやユーザ独自のコマンドにも対応できる
- コマンド文字列やコマンド知識に明示的に出現しない依存関係も抽出できる

一方、この方式は以下のような問題点を持っている。

- 関連のありそうな全ファイルの参照 / 更新時刻を調べる必要がある

  コマンドを起動する度に、その結果としてどのファイルが参照 / 更新されたかを調べなければならない。

- 調べた範囲外のファイルを参照 / 更新していた場合依存関係を検出できない。

- 正確な時刻の順序に注意が必要

  コマンドの起動によりファイルが参照されたときそれらの動作はほぼ同時に起こることが多いが、因果関係を正確に知るためにはコマンド実行 / ファイル参照などの時刻を正確に記録する必要がある。

ファイルの参照 / 更新時刻は高速に調べられるのが普通であるし、調べた範囲外のファイルは静的であると考えられ、依存関係にとって重要ではないことが多いので、これらの問題点は実用上大きな問題とはならない。

5.3.5 Smart Make 使用例

UNIX 上のファイル**test1.c**, **test2.c** からアーカイブファイル**test.tar** を作成し、これを圧縮して**test.tar.Z** を作成し、さらにこれを**uudecode** 型式のテキストファイルに変換してファイル**test.uu** を作る場合を考える。これは以下のようなコマンド列により実行される。

(**test1.c** と**test2.c** から**test.tar** を作成)
% tar cf test.tar test1.c test2.c
(**test.tar** を圧縮して**test.tar.Z** を作成)
% compress < test.tar > test.tar.Z
(**test.tar.Z** から**test.tar.Z** を作成)
% uuencode test.tar.Z < test.tar.Z > test.uu

これらの操作によるコマンド発行及びファイルの参照 / 更新を時間順に並べると以下のようになる。

1: **tar** コマンド起動
2: **test1.c** の参照
3: **test2.c** の参照
4: **test.tar** の更新
5: **compress** コマンド起動
6: **test.tar** の参照
7: **test.tar.Z** の更新
8: **uuencode** コマンド起動
9: **test.tar.Z** の参照
10: **test.uu** の更新

これをコマンド毎に区切り、図 5.5の規則に従って Makefile を生成すると以下のような Makefile が得られる。

```
test.tar: test1.c test2.c
        tar cf test.tar test1.c test2.c
test.tar.Z: test.tar
        compress < test.tar > test.tar.Z
test.uu: test.tar.Z
        uuencode test.tar.Z < test.tar.Z > test.uu
```

test.c を更新後**test.uu** をターゲットとして Make を起動すると、**tar**, **compress**, **uuencode** が順次起動されて**test.uu** が自動更新される。

5.4 評価

5.3.5節の例のような単純なコマンド操作を何度も行なう場面は実際に多いため、 Makefile 作成による操作の単純化のメリットは大きい。 Dynamic Macro の手法は文書編集作業では有効であったが、 4.5.5節で述べたようにコマンド実行の効率化には向いていない。これに対し、 Smart Make では単なる操作の繰返しではなくファイルやコマンドの間の依存関係に注目しているため、作業の途中に関係無いコマンドが混じっているような場合でも問題は起こらないし、同じ操作が 2 度繰り返される必要もない。文書編集もコマンド実行もキーボードを使った作業であるという点は同じであるが、作業の質が異なるため有効な予測 / 例示インタフェース手法も異なると考えられる。

5.5 議論

5.5.1 制約を抽出する異なる手法

5.2.1節で述べたように、ファイルの参照 / 更新時間を使用する以外にも、有用な制約関係を自動抽出する方法は他にも各種考えられる。本節では、履歴情報から制約関係を単純な方法で抽出し再利用するための異なる手法について具体的に述べる。

コマンドシンタクスの利用

システムによっては、コマンドや引数の間の依存関係がコマンド文字列から簡単にわかる場合がある。図 5.6に、文献データベース[2]から検索を行なった結果の例を示す。下線の引いてある文字列がユーザからの入力で、それ以外はシステムの応答である。

[2]学術情報センタの学会発表データベース

```
   1/ search map
*     792    1/  K.MAP
   2/ search display
*    1797    2/  K.DISPLAY
   3/ search information
*    7057    3/  K.INFORMATION
   4/ search wing
*     13     4/  K.WING
   5/ or 2,3
*    7891    5/  2 OR 3
   6/ and 1,5
*     116    6/  1 AND 5
```

図 5.6: 情報検索システムとの対話例

ユーザは最初にキーワード “**map**” で検索を行ない、 792 個のデータマッチが検出されている。次に “**display**” で検索を行なうと 1797 個のマッチが検出され、 “**information**” で検索を行なうと 7057 個のマッチが検出されている。その後、 “**display**” か “**information**” かどちらかのキーワードを含む文献を**or** 演算により抽出し、それに加えて “**map**” も含むものを**and** 演算により抽出している。これらの処理の流れと依存関係は図 5.7のように表現できる。

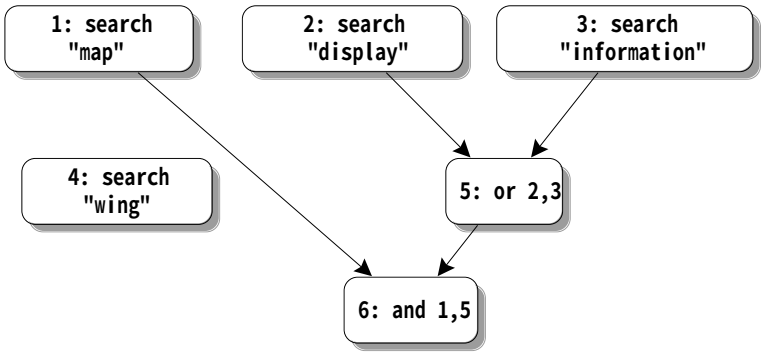


図 5.7: 検索コマンド間の依存関係

ここで、キーワード “**display**” のかわりに “**visualization**” を使用して同様の検索を行ないたいとユーザが考えた場合、普通はそれに続く**or**, **and** 演算もユーザが再度指定する必要があるが、検索コマンドの依存関係がコマンド履歴から簡単にわかるため、キーワード “**display**” のかわりに “**visualization**” を指定して**6** までの再実行を指示するだけで、 “**visualization**” の検索後、前回と同じ手順を実行させることができる。このとき、図 5.8の灰色の部分が自動実行可能になる。

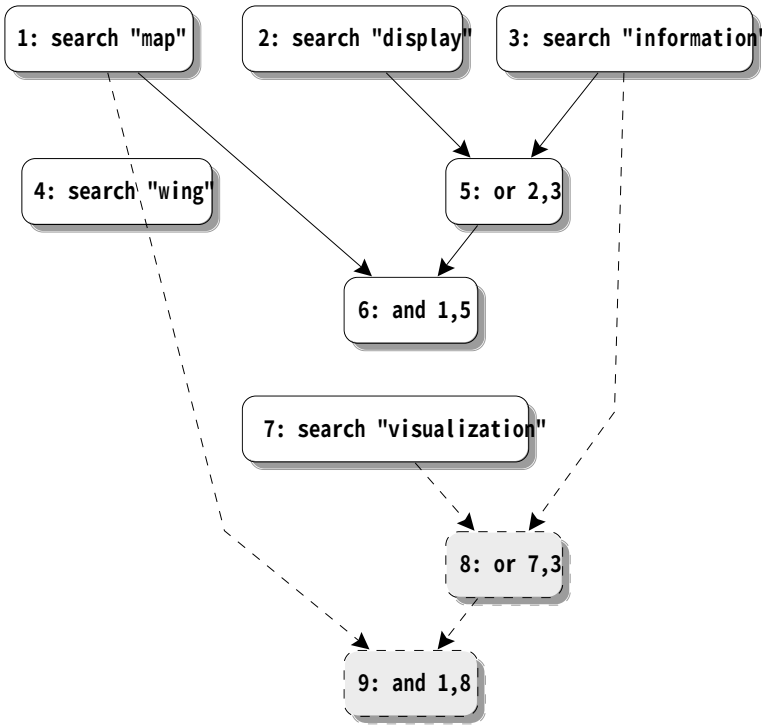


図 5.8: 検索コマンドの自動実行

undo の利用

前節のデータベース検索の例では、パラメタを変えて再実行する部分を明示的に指定していたが、操作履歴を記憶しておきかつ**undo** 機能を積極的に利用することにより、単純な方法で電卓に計算の再実行を行なわせることができる。

電卓でひとつの値をもとにして色々な計算を行なった後、その値が間違っていたことに気付いた場合や、異なる値を用いて再計算したいと思ったような場合は、値を変えて全ての計算をやり直さなければならないのが普通であるが、電卓に後退キーを付加することにより、以前に入力した数値を変更して計算のやり直しをさせることができるようになる。このような機能をもつ電卓について以下に述べる。

摂氏 10 度が華氏で何度になるかを一般の電卓で計算したときのキー入力と表示は図 5.9のようになる。

キー入力	表示
10	**10**
×	**10**
9	**9**
／	**90**
5	**5**
＋	**18**
32	**32**
＝	**50**

図 5.9: 華氏の計算

この計算により摂氏 10 度は華氏 50 度であることがわかる。しかし、異なる温度についても計算を行ないたい場合は、全ての操作を繰り返すか、プログラミング電卓などを使用しなければならないのが普通である。ここで、キー入力の履歴を保存しておいて、後退キーによって入力したパラメタを後から変更可能とすることによって、プログラミング機能のない電卓をプログラミング電卓のように使うことができる。

図 5.9の計算は図 5.10のような状態遷移で表現できる。

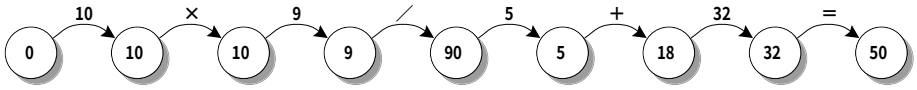


図 5.10: 華氏の計算における入力と表示の遷移

後退キー (「←」と表記) を押すたびに前の値の入力時の状態に戻ることができるようにすると、後退キーを 4 回押すことにより、図 5.11のように “**10**” を入力したときの状態まで復帰することができる。

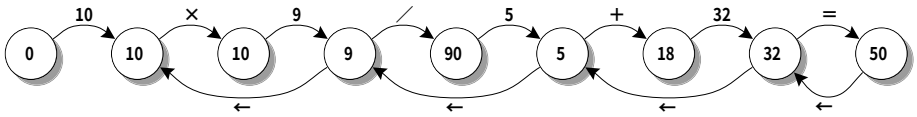


図 5.11: 後退キーにより前の状態に復帰

この状態で入力値を変更して “**=**” キーを押したときに、後退キーを押し始めたときまでの操作が繰り返されるようにすることにより、図 5.12のように華氏の計算を再実行することができる。

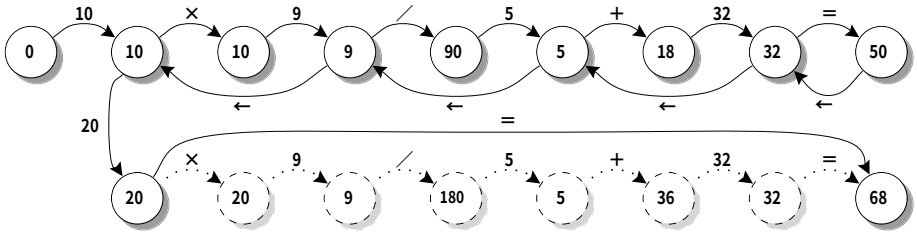


図 5.12: 値を変えて再計算の指示

図 5.9の計算に続いて華氏の計算を異なる温度について再計算させたときの入力と表示の変化の様子を図 5.13に示す。
後退キーの入力により、以前に入力した値が順次表示され、新しい値を入力した後「＝」キーを押すことにより前回の処理が再実行される。

このような手法により、プログラミングを行なうことなく順次パラメタを変えながら計算を行なわせることが可能である。また、逆計算が難しい計算も試行錯誤によって比較的簡単に解くことができる。例えば方程式の解の大体の値が知りたいとき、適当な解の侯補の値で計算を行なった後、パラメタを少しずつ変えて式の値を計算することにより大体の解を知ることができる。

このような単純な手法で操作手順を抽出する手法には限界があるが、操作が単純で特別な入出力装置 / 手法が必要ないことがこの方式の利点である。また、一般のプログラミング電卓を使用する際には抽象的な思考が要求されるのに対し、一度具体的な計算を行なって計算が正しいことを確認した後でパラメタを変えて再計算させることができるという意味で、キーボードマクロと同様の特長を持っているし、最初からプログラムの作成を念頭においておかなくても計算手順を後から再構築可能であるという点で Dynamic Macro と同様の特長を持っている。

キー入力	表示	
	50	前の表示
後退キー	**32**	最後に入力されたパラメタが表示される
後退キー	**5**	続けて後退キーを押すとその前のパラメタが表示される
後退キー	**9**	
後退キー	**10**	最初に入力したパラメタの表示
20	**20**	新しいパラメタ (20℃) を入力
＝	**68**	20℃に対して華氏の計算が再実行される
30	**30**	さらに別のパラメタ (30℃) を入力
＝	**86**	30℃に対する再計算結果
...		

図 5.13: 華氏の再計算

5.5.2 予測 / 例示インタフェースの問題点の検証

Smart Make において、 3.3.1節で述べた問題点が克服されているかどうか検証する。

1. プログラムを作成できない

   Smart Make では、依存関係の定義という形式で、条件により動作を変えるプログラムを自動的に作成することができる。

2. プログラム生成のための指示が面倒

   依存関係はコマンドにより Makefile の作成を指令するだけで抽出される。

3. プログラム実行のための指示が面倒

   制約の解消は通常の場合と全く同じように Make コマンドの起動により実行され、特殊な予測実行指令コマンドは必要ない。

4. 正しいプログラムの作成が困難

   5.4節で述べたように、高い率で正しい Makefile が生成される。

5. 実行に関するリスクが大きい

   制約解消のためのコマンドはすぐ実行される必要はなく、**-n** オプションを用いて Make コマンドを起動することにより、正しい依存関係が抽出されているかを実行前に確認することができる。

6. 例示 / 予測手法を使わない方が楽である可能性がある

   Makefile 作成を指示する手間は非常に小さいため、このような可能性は少ない。

7. 機能がユーザの邪魔になる

   Makefile 作成を指示しない限り推論は実行されないので、機能を使用しないユーザの邪魔になることはない。

8. 大量のデータが必要

   依存関係抽出に必要な履歴データはわずかである。

9. ヒューリスティクスが多用されている

   5.3.4節に述べた単純な規則のみが予測に使用される。

以上のように、3.3.1章で述べた全ての点に関して問題が解決していることがわかる。これを表3.1に従って表にすると表5.1のようになる。

	プログラム不可能	定義時の指示が面倒	実行時の指示が面倒	正しい推論が困難	実行リスクが大	時間的リスクが大	勝手な実行が邪魔	大量データが必要	ヒューリスティクスが過度
Smart Make	○	○	○	○	○	○	○	○	○

表 5.1: 予測 / 例示インタフェースの問題点の Smart Make における解決

5.6 まとめ

本章では、操作履歴情報から操作やファイルの間の依存関係を自動抽出して再利用することのできる Smart Make システム及び関連システムを提案した。Smart Make では、依存関係を抽出する単純な規則によってユーザの操作履歴やファイル作成 / 参照時刻から暗黙的な依存関係を抽出し、単純な操作で再利用することが可能である。

第 6 章　進化的学習にもとづいた予測 / 例示インタフェース手法

6.1　はじめに

本章では、遺伝的プログラミングの手法を用いてユーザの好みを示す関数を例データからプログラムの形として抽出し再利用する手法について述べる。またその実例として、以前の配置例をもとにして図形の配置の評価関数を自動抽出するシステムについて述べる。遺伝的アルゴリズムや焼きなまし法のような確率的手法を用いることにより、図形配置の良さを示す評価関数を与えることができれば自動的に最適に近い配置を求めることができるが、そのような評価関数を求めることがむずかしい場合が多い。本章で示すシステムでは暗黙的に示されるユーザの好みを例示データのみから評価関数として自動的に生成することが可能である。

6.2　グラフ自動配置問題と遺伝的アルゴリズムによる解法

6.2.1　自動配置問題

多数の図形を限られた領域に適当な制約のもとに効果的に配置したいという要求は多い。 VLSI のレイアウト・都市計画のレイアウト・布地の効率的裁断などはすべてこのような配置問題の一種である。計算機のユーザインタフェースにおいても配置手法は重要である。複雑なデータ構造や大量のデータを視覚化するためには計算機が人間にわかりやすい配置を計算しなければならないし、ウィンドウシステムのようなグラフィック画面を使用した対話的環境では常にユーザが認識・操作しやすい配置を画面に出力する必要がある。 “文字入力部はウィンドウの中心” といった制約記述により対話画面の配置が決定されるような、自動配置システムを UI に応用した各種のツールの研究が盛んである [67] [70][88]。

広く使われている文書整形システム TEX も制約を用いた配置システムの一種である。 TEX は文字・単語・パラグラフ等をある評価関数にもとづいて箱のように並べながら 2 次元領域に配置していく。 TEX のユーザは配置アルゴリズムのパラメタをある程度操作することはできるが、配置戦略はプログラムとしてシステム内に深く組み込まれているため根本的に配置手法を変更することはできない。 TEX と同様に、自動配置システムのほとんどにおいて配置アルゴリズムをユーザが変更することは不可能である。

制約解決の複雑さは制約の性質により大きく異なる。線型の制約は簡単に解くことができるため、ユーザインタフェースツールのようにユーザの操作に高速に反応しなければならないシステムでも使用することができるが、グラフの美しい配置のような複雑な制約をもつ問題においては、最適解を求めるアルゴリズムが存在しなかったり解くのに膨大な時間かかったりするのが普通である。このように最適解を実際上求めることが不可能な配置問題に対しては発見的手法 (ヒューリスティクス) を使用して最適に近い解を求める手法が従来用いられてきた。しかしこれには以下のような欠点がある。

- 手法の発見がむずかしい
- 制約がほんの少し変化しただけでも以前の手法が有効でなくなる場合が多く、柔軟性に欠ける

このため、制約を直接解かず、遺伝的アルゴリズム(GA) や焼きなまし法(アニーリング法, Simulated Annealing)[34]のような確率的手法を使用する方法が近年注目を集めている。これらの手法を用いると、配置手続きを考案しなくても、配置結果においてどの程度制約が満足されているか評価することさえできれば、制約の種類によらず、繰り返し計算を行なうことにより徐々に最適に近い解を計算していくことができる。

6.2.2 遺伝的アルゴリズムによる配置問題の解決

遺伝的アルゴリズム概要

遺伝的アルゴリズムは、自然淘汰による進化を模擬するアルゴリズムで、近年様々な最適化問題や機械学習の分野で幅広く応用されてきている。

遺伝的アルゴリズムにより最適化問題を解く手順は以下のようになる。まず計算に使用する一定長の文字列 (**染色体**) から解空間への写像を適当に決め、解の候補となるランダムな文字列の集合を用意する。要素の文字列それぞれについてその表現する解を計算し、最適化したい評価関数によりその評価値を求める。評価値の良いものは残し、悪いものは捨てるようにして残った文字列の間でその部分文字列の交換 (**交差**演算)及びランダムな文字置換 (**突然変移**演算)を実行する。こうして得られた新たな文字列の集合について同じ操作を繰り返すうちに評価の良いものが蓄積されて最適に近い解を表現する文字列が残る。部分文字列の交換、文字置換のような操作を遺伝的演算と呼び、繰り返しの 1 周期を世代と呼ぶ。交差演算、突然変異演算を図 6.1に示す。

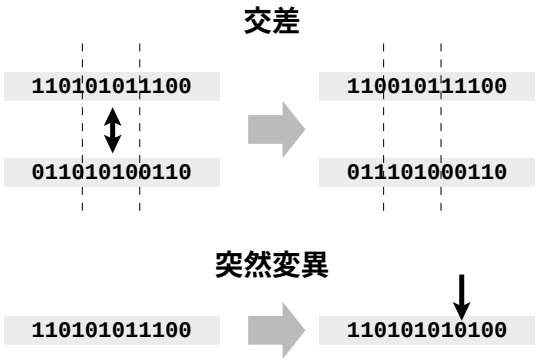
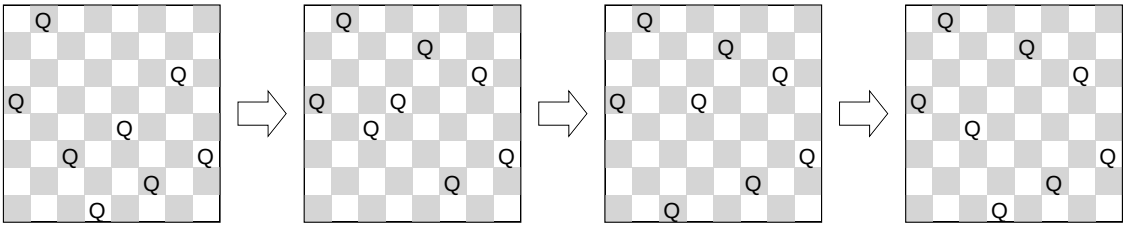


図 6.1: 一般の遺伝的アルゴリズムの遺伝的演算

単純な例と特長

遺伝的アルゴリズムによる配置の単純な例として、遺伝的アルゴリズムで 8-Queen問題を解いた例を図 6.2に示す。



(a) 衝突の数 = 3　(b) 衝突の数 = 2　(c) 衝突の数 = 1　(d) 衝突の数 = 0

図 6.2: 8-Queen 問題への遺伝的アルゴリズムの適用

ここでは配置のよしあしを示す評価関数として行 / 列 / 斜めの駒の競合の数を使用している。図 6.2(a) では下から 3 行目にふたつの駒があり、また斜めに並んだ駒の組がふたつ存在するため評価値は 3 となっている。

多少複雑な例として、遺伝的アルゴリズムを木構造の配置に適用した結果を図 6.3に示す。左側中段のグラフの X 軸は世代を示し、Y 軸はその世代における平均評価値を示している。配置を示す 5 個の図は、それぞれの世代における最適配置を示している。

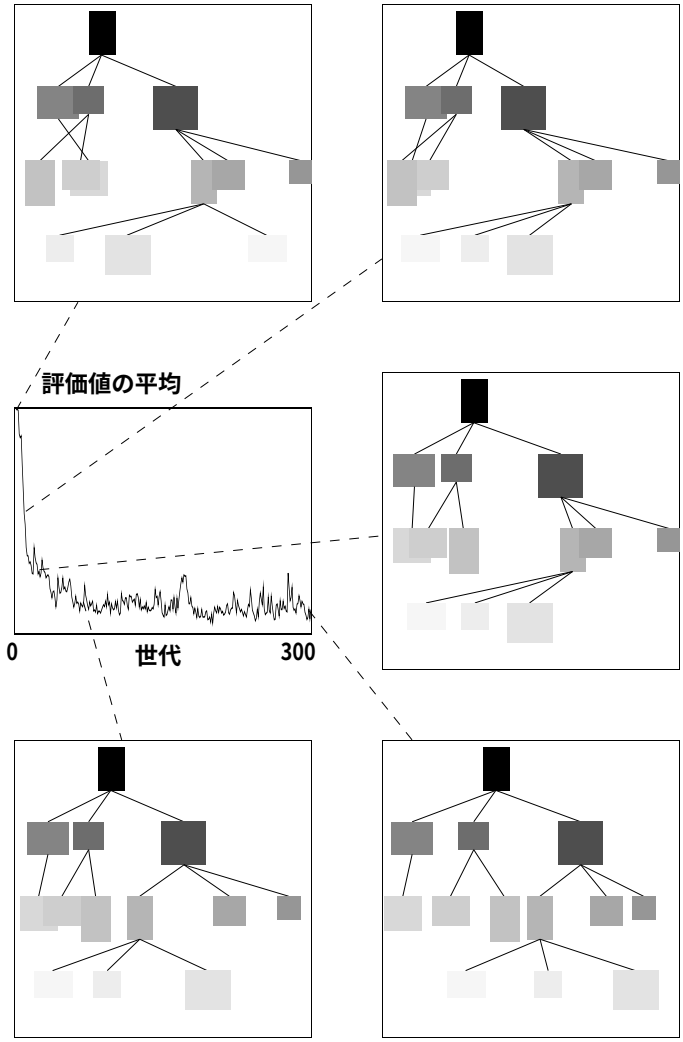


図 6.3: 木構造の配置への遺伝的アルゴリズムの適用

ここでは、 1) 節点は重なりあわない, 2) 子節点の左右の順番が正しい, 3) 親節点の位置が自分の右端の子と左端の子の中間, という 3 個の制約を評価関数で使用している。図 6.3において、薄い色の節点が濃い色の節点よりも左側に位置するのが正しい順番である。

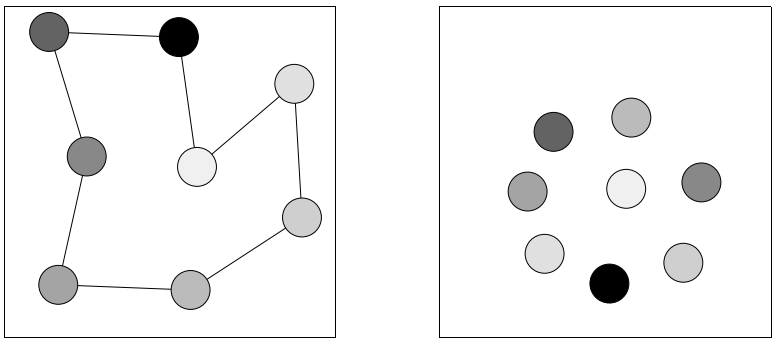
評価関数を図 6.4に示す。制約がよく満たされているほど小さい値が返される。

```
int evaluategene(char *gene)
{
    int val = 0;
    節点の位置,大きさの情報をgeneから取り出す;
    for(全ての節点){
        if(隣の節点と重なっている){
            val += (重なり幅) * 100;
        }
    }
    for(全ての節点){
        val += sqr(節点の位置 -
                   子節点の中心位置);
    }
    /* 他の評価基準の計算 */
    return val;
}
```

図 6.4: 木構造配置の評価関数

このように、制約を解くためのアルゴリズムを陽に指定しなくても求める結果を得ることができるのが遺伝的アルゴリズムの特長である。木構造を美しく配置する問題はそう難しいものではなく、高速な配置アルゴリズムも各種提案されているが [59] [83]、美しさの評価基準が変わった場合でも評価関数を変更するだけですむということが遺伝的アルゴリズムを使用する利点である。

また遺伝的アルゴリズムでは矛盾した制約を指定した場合でさえ何らかの意味のある配置を得ることができる。



(a) 実線の長さが等しくなるよう指定した場合　(b) 全ての円の間の距離が等しくなるよう指定した場合

図 6.5: 図形間の距離が等しくなる配置を求める

例えば図 6.5(a) は 8 個の図形間の距離のうち実線で示したものの長さが等しくなるような配置を遺伝的アルゴリズムで求めた結果である。この場合ほぼ要求を満足する結果が得られている。これに対し図 6.5(b) は 8 個の図形間のすべての距離が等しくなるような配置を求めようとした結果である。そのような配置は不可能であるにもかかわらず、それなりに意味のある結果が得られていることがわかる。これに対し、例えば線型制約の解決システムにおいては矛盾する制約があると単に解けないという結果が得られるだけである。

GA 手法 / パラメタの選択

遺伝的アルゴリズムでは各種のパラメタの選択が計算結果に重大な影響を与えるが、本節で

は一般の図形配置問題においてどのような手法が用いられているかを解説する。

解の染色体へのコーディング方式　遺伝的アルゴリズムにおいては、求める解を何らかの規則により「染色体」にエンコードする必要がある。図形配置問題では以下のようなエンコード手法が考えられる。

1. 各図形の座標値を並べる方法

   この方式によると、ある図形が (10, 20) の位置にあることを示すのに、これらの座標値を例えば 8 ビット 2 進表現で “**`00001010 00010100`**” のように表現し、これを図形の数だけ並べたものを染色体として用いる。この方式は単純であるため多くのシステムで使用されている。前節の 8 クイーンや木構造の配置でもこの方式を使用している。通常の 2 進表現のかわりにグレイコードを使用したり、2 進表現のかわりに整数の配列をそのまま遺伝子操作に使うこともできる。

2. 染色体上の位置で図形の位置を表現する方法

   この方式では、図形を配置可能な位置を示す 2 次元配列を染色体として用いる。例えば図形 5 が (3, 4) の位置にあるとき、この配列の 3 行 4 列の値を 5 とすることによりこれを表現する。染色体操作はこの配列内の行 / 列の交換により行なう。

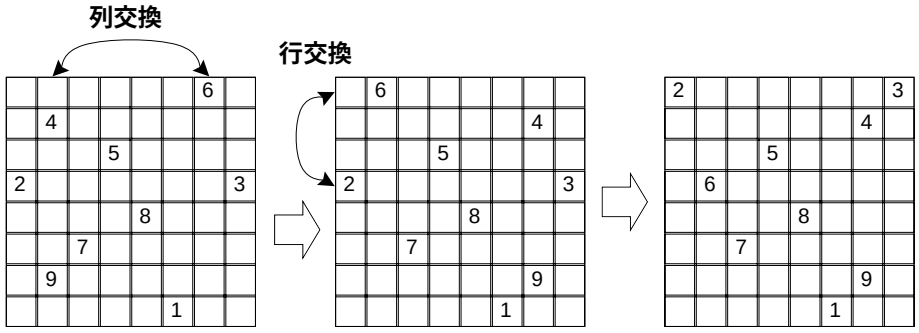
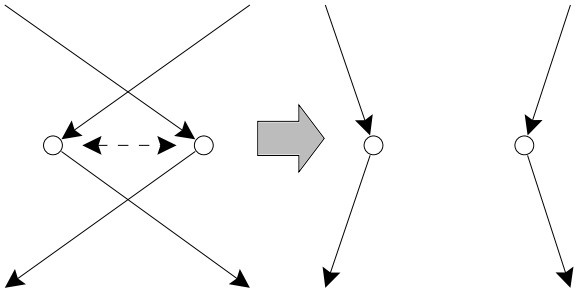


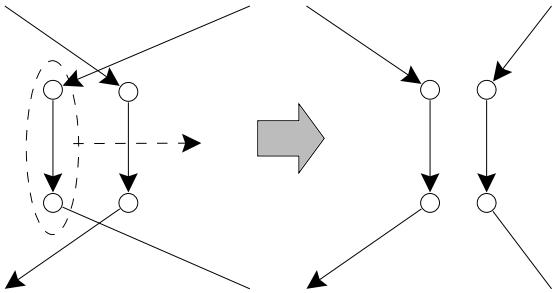
図 6.6: 行と列の交換

2. の方式は [24] で用いられているが、配置可能な位置が多いとき染色体が巨大になってしまうし、本来の GA の特長が生かされていないため良い結果を得ることができない。

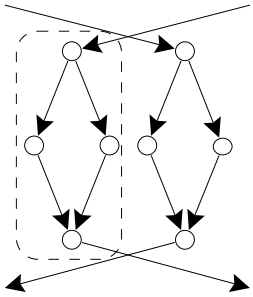
染色体操作　6.2.2節で述べたように、一般の遺伝的アルゴリズムでは**交差**と**突然変異**の 2 種類の染色体操作を行なうことが多いが、図形配置問題においては、問題に適合した特別の染色体操作が有効であることが多い。例えばグラフの配置においては、ふたつの図形の位置を**交換**する操作が有効である。図 6.7(a) は一回の交換操作が有向グラフの配置においてよい結果を生み出す様子を示している。また図 6.7(b) に示されるような**ブロック移動**操作も適用可能である。



(a) 反転



(b) ブロック移動



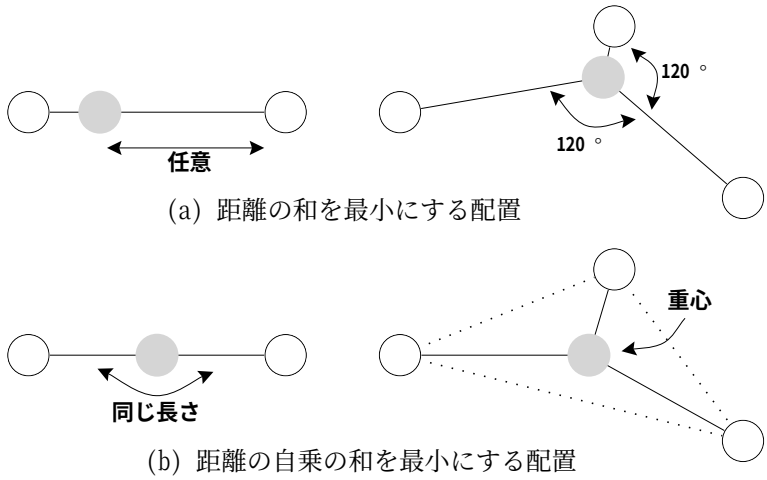
(c) ブロック移動が困難な例

図 6.7: 交換とブロック移動

評価関数の選択

遺伝的アルゴリズムを用いた配置手法が手続き的手法に比べ最もすぐれている点は、配置の制約を解く手法を考えることなく、評価関数を設定するだけで良好な配置を得ることができるということである。また評価の重みづけの変更により制約の強弱を簡単に制御することができることも利点である。

VLSI のレイアウトのような問題では配線長や総面積を減らすという目的がはっきりしているため評価関数の選択の余地はあまりない。しかし図形の美的配置のような問題では目で見て美しい配置が高い評価値を得るような評価関数を選ばなければならないが、新しい配置問題に対し適当な評価関数をみつけることはそう単純ではない。例えば、ある図形を別の複数の図形の間に適当な間隔を置いて配置したいような場合、新しい図形から古い図形までの距離の和を評価関数として用いてそれが最小となるような配置を求めるようにすると、図 6.8(a) のように、図形が 2 個の場合全くうまく働かないし、3 個の場合でも適当な位置に配置されない。(3 個の図形への直線間の角度が 120 度になるような点において距離の和が最小になる。) しかしここで距離の自乗の和を評価関数として用いると、図 6.8(b) のように重心に配置されるようになりこの場合は都合が良い。



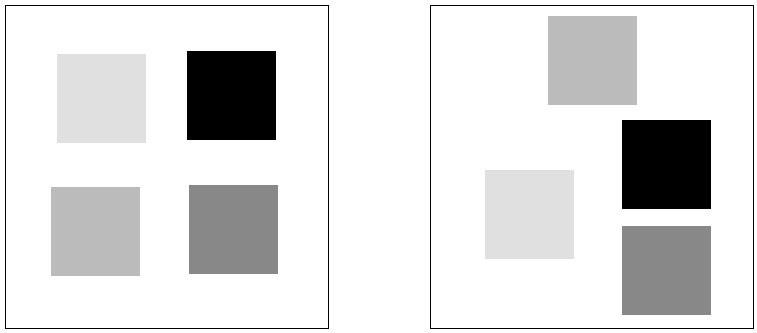
(a) 距離の和を最小にする配置

(b) 距離の自乗の和を最小にする配置

図 6.8: 評価関数よる配置の違い

このように、評価関数により実際どのような配置が得られるかは自明でないことが多いし、いろいろな種類の評価関数を複合して使う場合はさらに評価関数の選択はむずかしくなるため、結局試行錯誤により良い評価関数とその重みを決定するより仕方がなくなってしまう。

また評価関数や重みの選択はクリティカルであることが多く、このような場合、評価関数の作りかたにより挙動が大きく変化し、ユーザにとって意外な配置結果が得られてしまう。



(a) 期待通りの解 **(b) 制約を満足する別の解**

図 6.9: 期待に反する解が得られた例

例えば図 6.9は 4 個の正方形を美しく配置しようとして以下のような制約を使用した結果計算された結果である。

1. 各正方形の左右の余白の大きさが等しい
2. 各正方形の上下の余白の大きさが等しい
3. 濃い色の正方形は薄い色のものより上側
4. 濃い色の正方形は薄い色のものより右側

これらの制約により図 6.9(a) のような結果を期待したにもかかわらず図 6.9(b) のような結果が得られてしまっている。

6.2.3 有向グラフ配置問題への GA の応用

本節では遺伝的アルゴリズムを有向グラフの配置問題に適用した例について解説する。無向グラフの二次元平面上への配置については [36] [45]などが報告されている。

有向グラフの配置問題

有向グラフとは節点の集合 N と枝の集合 E で構成されるグラフで、枝は節点を要素とする順序対 (n,m) で表現される。図 6.10は 4 個の節点と 5 本の枝をもつ有向グラフの例である。

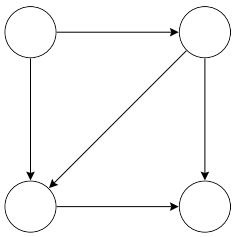


図 6.10: 4 個の節点と 5 本の枝を持つ有向グラフ

節点や枝の数が多いとき、これらを見やすく紙や画面のような 2 次元空間に配置することは非常に困難である。見やすい配置を得るためには図 6.11や以下に示すような制約を考えることができる。

1. 節点の間に充分な間隔があること
2. 枝の先頭は後尾よりも下方にあること
3. 枝の交差の数がなるべく少ないこと
4. なるべく図が対称であること
5. 同じ節点から出るふたつの枝の間の角度が小さすぎないこと
6. 節点が配置画面上に適当に分散しているいこと

これらの制約を解くための各種の配置アルゴリズムが提案されている [81]。しかし枝の交差の数を最小にするような節点の配置を求める問題は NP 困難であり、他の多くの制約についても同様であるため、提案されているアルゴリズムはなんらかの発見的手法を使用している。例えば Eades と杉山 [18]による方法では以下の手順で配置を行なう。

段階 1 節点を枝の向きでソートする

段階 2 グラフの頂上から下端の間の “層” の数の最小値を計算する

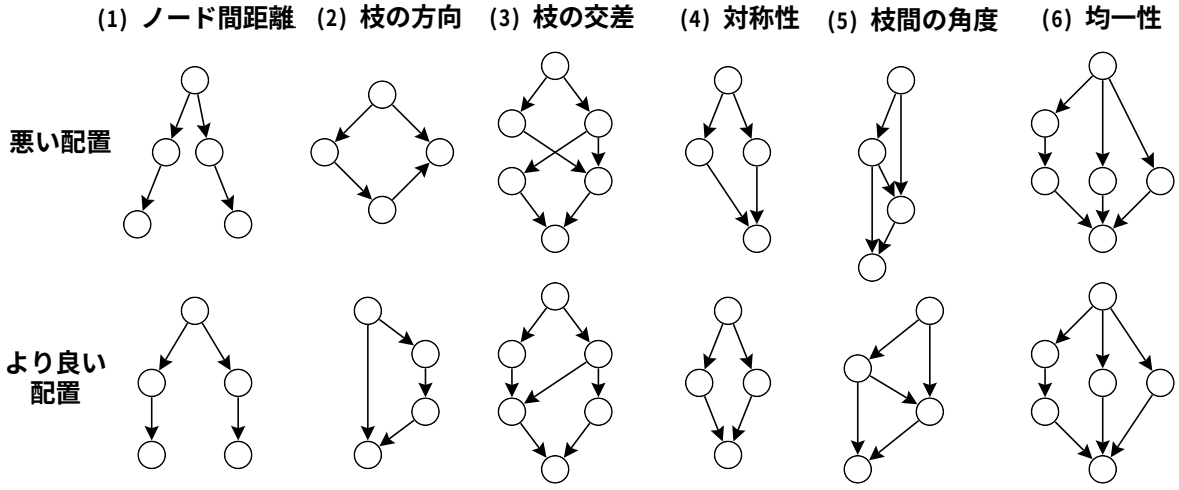


図 6.11: 有向グラフの配置における制約

段階 3 同じ層内のふたつの節点間に枝が存在しないようにすべての節点をどれかの層に割り当てる。このとき節点がなるべく均一に各層に割りあてられるようにする。

段階 4 層をまたいだ枝が存在するときはそこにダミーの節点を導入し、そのような枝が無くなるようにする。

段階 5 各層内で節点を移動させ、枝の交差の数が最小になるようにする

ここで段階 2, 3, 5 は NP 困難であり、いくつかの発見的手法が用いられている。

Kosak らの有向グラフ配置システム

本節では、実用的な有向グラフ配置システムの例として、 Connection Machine 上の並列遺伝的アルゴリズムを用いて有向グラフを視覚的にわかりやすく表現するシステム [35] [36]を紹介する。

このシステムでは “知覚的構造”(Perceptual Organization) を前節で述べたような審美的要素よりも重視した配置を行なうことを目標としている。知覚的構造とはグラフの理解を助けるのに役だつ構造で、ノードの集合を囲う箱があるとかノードが一定間隔で並んでいるとかいった構造のことをいう。このような構造をもった配置を得るため、ノードがある矩形領域に含まれるとかきれいに横に並ぶとかといった属性を指定することができるようになっている。

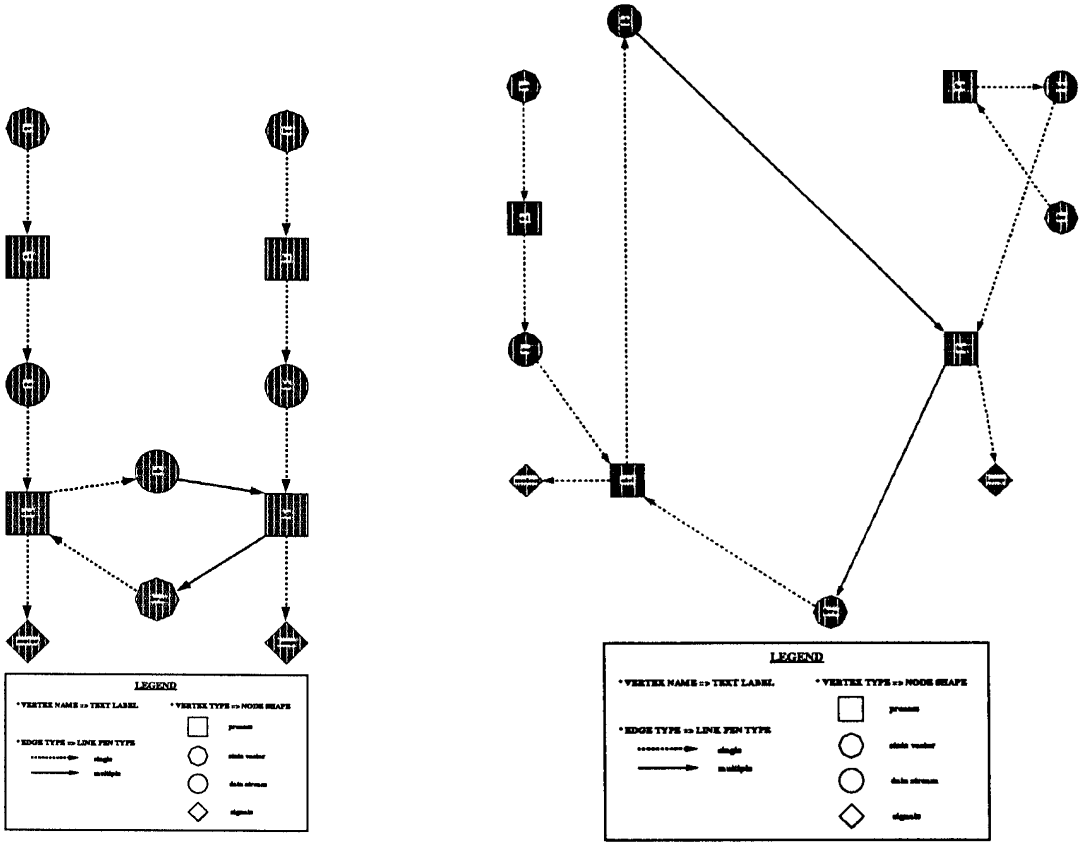
構造を持ったノード群を扱うため、ノードの位置は単純に 1 次元の染色体にマッピングせず、構造をもったまま染色体操作を行なうようになっている。例えば、突然変異操作によりノードの位置は移動するが、このときはひとつの構造内のノードすべてが同じだけ移動する。また交差操作ではノードの集合同士の入れ替えが行なわれる。

評価関数としてはペナルティの総和が使用される。ペナルティは、

1. 構文的に正しいこと (ノードが重なっていない,etc.)
2. 指定した属性 (知覚的構造) が満たされているか
3. 審美的に良いか、

の順で大きくなっている。

このシステムによる配置の実行例を図 6.12に示す。



(a) 配置がうまくいった例　　(b) 不適切なパラメタのため失敗した例

図 6.12: Kosak のシステムによる配置の実行例 ([36]による)

6.2.4 対話型 GA 配置システム GALAPAGOS

遺伝的アルゴリズムは強力である反面、配置手続きを直接指定しないため、結果にユーザの意図を反映させることがむずかしい場合がある。得られた配置がユーザの気に入らない場合は制約を修正して再計算させなければならないが、その結果望む配置が得られる保証は無く、何度も試行錯誤を行なわなければならない。一方、美的感性が必要になるような配置問題では、完全自動配置は手動による配置に及ばない場合がある。

このような問題点を解決するための、遺伝的アルゴリズムを拡張しユーザが対話的に制約を変更しつつ配置計算を行なうことにより望ましい最終解を得ることのできるシステム GALAPAGOS (Genetic ALgorithm And Presentation-Aided Graphic Object layout System) [48] [104] [106]について述べる。 GALAPAGOS を使用すると、単体の遺伝的アルゴリズムによる自動配置では満足できる結果が得られない場合でも、得られた解を直接修正したり制約を計算途中で修正したりすることにより、自動配置と手動配置の両者の特長をもった望ましい解を得ることができる。まず大体の配置をシステムに計算させ、ユーザがそれを修正し、細かな最終処理を再度システムに処理させるなどの処理が可能になる。

GALAPAGOS 概要

GALAPAGOS は配置結果や制約をユーザが実行時に変更することのできる遺伝的アルゴリズム配置システムである。遺伝的アルゴリズムのもつ特長に加え、ユーザが実行時に対話的に配置結果や制約を変更することがきるため、自動配置と手動配置の両者の特長を兼ね備えた良好な配

置結果を得ることができる。

システムはまず予め与えられた制約にもとづいて計算を開始し、各世代における最適配置を表示する。ユーザは適宜その配置を修正したり制約を修正 / 追加したりすることができる。例えば、ある対象が好ましくない位置に配置されようとしていることにユーザが気付いた場合、ユーザはそれを別の望ましい位置に移動させて固定させることができる。また、あるふたつの対象の Y 座標を一致させた方が美しい配置になるとユーザが感じた場合、それらが同じ Y 座標をもつという新しい制約を追加することができる。交差率や突然変異率のような遺伝的アルゴリズムのパラメタも途中で変更することができる。

GALAPAGOS は GENESISシステム [23] と同じアルゴリズムを使用している。 GALAPAGOS の中心アルゴリズムを図 6.13に示す。

```
initialize();
for(gen=0; gen < maxgen; gen++){
     checkevent();   // ユーザ入力チェック
     selectchrom();  // 使用染色体選択
     mutate();       // 突然変異
     crossover();    // 交差
     evaluatechrom();// 染色体の評価
     measure();      // 最悪値の更新
     display();      // 結果の表示
}
```

図 6.13: GALAPAGOS のアルゴリズム

selectchrom() は Baker のアルゴリズム[3]にもとづいて染色体の選択を行なう。このアルゴリズムでは、各世代において評価の最悪値が設定されており、それよりも評価の悪い染色体は捨てられて次の世代に生き残らない。染色体は**selectchrom()** において、その数が自分の評価値と最悪値の差に比例するように選択される。最悪値は各世代において更新される。

checkevent() において GALAPAGOS はユーザの要求をチェックし、要求がある場合は制御をユーザに戻して、配置結果や**evaluatechrom()** で使用される制約を修正できるようにしている[1]。ユーザはそのまま処理を続行してもよいし、修正後続行してもよい。配置結果を修正した場合は、新しい配置を示す染色体が染色体集合全体の $1/3$ に戻されて処理が続行される。これにより修正結果が生き残るようになっている。

GALAPAGOS の実装

図 6.14に NeXT 社のワークステーション上に実装された GALAPAGOS システムの画面を示す。遺伝的アルゴリズムの実行及び得られた最適配置を表示するプログラム (GA Visualizer) と図形エディタは別プログラムになっている。

[1]GALAPAGOS と GENESIS のアルゴリズムの違いは **checkevent()** の有無のみである。

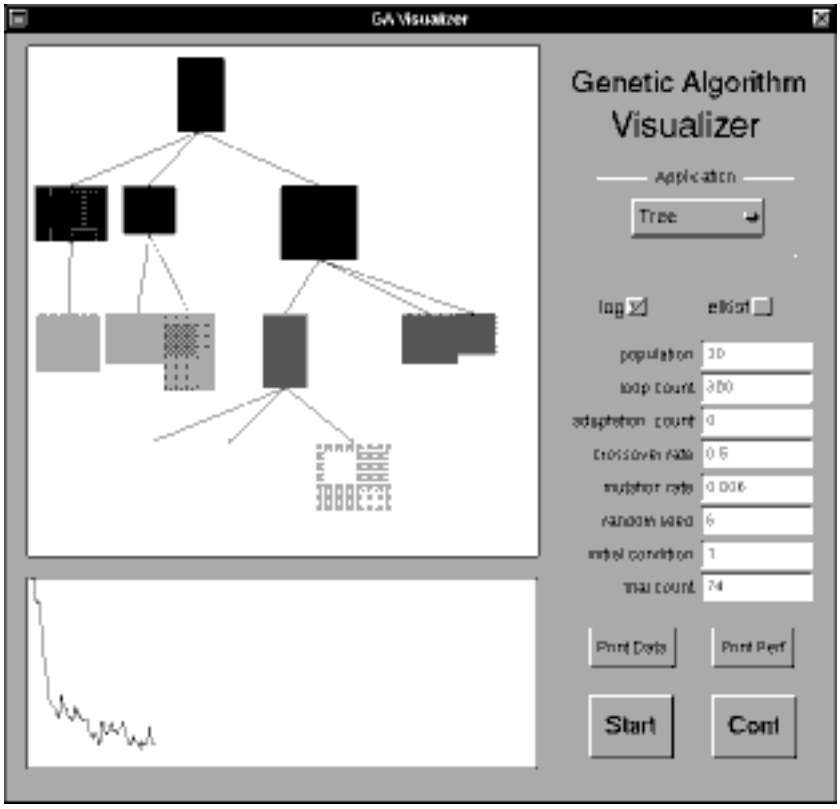


図 6.14: GA ビジュアライザ

初期データは図形エディタまたは他のプログラムにより GA Visualizer に渡される。 GA Visualizer は遺伝的アルゴリズムを実行し、各世代における得られた最適配置を表示する。その結果を見たユーザが制約を追加したいと思った場合は再び図形エディタを使って制約を追加し、 GA Visualizer に新しいデータを返す。

ユーザは右下の “Stop” 及び “Start” ボタンを押すことにより任意の時点で計算を中断、再開することができる[2]。遺伝的アルゴリズムの種々のパラメタは画面右のテキスト枠への入力で変更することができる。制約の修正は、まず現在の配置をファイルへセーブし、図形エディタで修正を行なった後もう一度それを読み込むという手順で行なわれる。

GALAPAGOS による有向グラフの配置

本節では GALAPAGOS を有向グラフの配置問題に適用した例について解説する。

配置に使用される制約　以下の制約がデフォルトで設定されている。

1. 枝の先頭は後尾よりも下に位置する
2. すべての節点間の距離がある定数値より大きい
3. 枝の交差の数がなるべく少ない
4. 節点を共有する枝の間の角度がある定数値より大きい

それぞれの制約に重みを示す定数が割り当てられており、$\sum_{\text{制約の種類}}$(制約の違反数 $\times$ 重み) が評価関数の値となる。以下に示す例では、重みの値として (3, 4, 3, 4) が使用されている。例えば、枝の交差ひとつごとに評価関数の値に 3 が加算される。

[2]図 6.14では “Cont” と表示されているが、これは “Stop” を押した後の同ボタンの状態である。

これらの組み込みの制約の他に、以下の制約をユーザが対話的に追加することができる。

5. 指定した位置に節点を固定
6. ふたつの節点の X 座標が等しい
7. ふたつの節点の Y 座標が等しい

配置の実施例　図 6.15に GALAPAGOS を使用した配置の流れの例を示す。まず有向グラフの接続情報がシステムに与えられ、システムは遺伝的アルゴリズムの実行を開始する。図 6.15(a) がシステムが最初に求めた配置である。何世代か経過した後かなり改善された配置 (b) が得られる。ここでユーザは、グラフの上部の 3 個の節点の Y 座標が等しく、下部の 2 個の節点の Y 座標も等しければさらに美しい配置となると考え、(c) の大きな矢印で示された 3 個の制約を追加する。新しい制約は点線で示されている。この後計算を続行することにより最終的に (d) の結果が得られる。

図 6.16に別の例を示す。ユーザは最初に得られた配置 (a) を修正し、3 個の節点につけられた番号が並ぶように (b) のように節点を固定する。固定された節点には影がついている。同時に上部のふたつの節点の Y 座標が等しくなるような制約も加えられている。この後計算を続行し、最終結果 (c) が得られる。

両方の例において同じ個体数 ($= 200$)、交差率 ($= 0.8$)、突然変異率 ($= 0.006$) が使用されている。

GALAPAGOS の特長

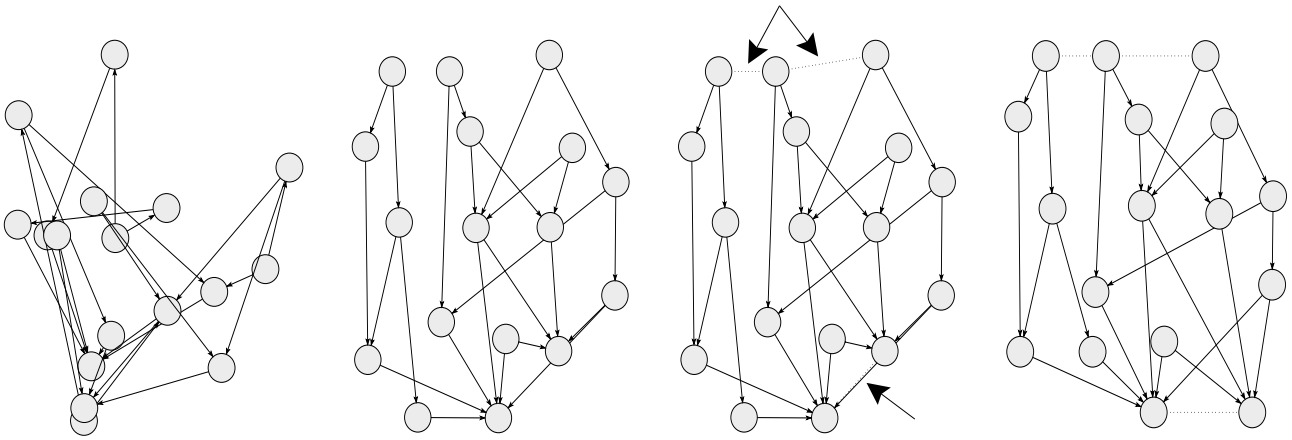
GALAPAGOS は、一般的な遺伝的アルゴリズムの特長に加え、ユーザの制約操作による細かい指示が可能であるという特長を持っている。これらをまとめると以下のようになる。

- 配置アルゴリズムが不要

  図 6.4のような評価関数を与えるだけで自動的に配置が得られるため、複雑な制約に対しても配置のためにアルゴリズムを考案する必要がない

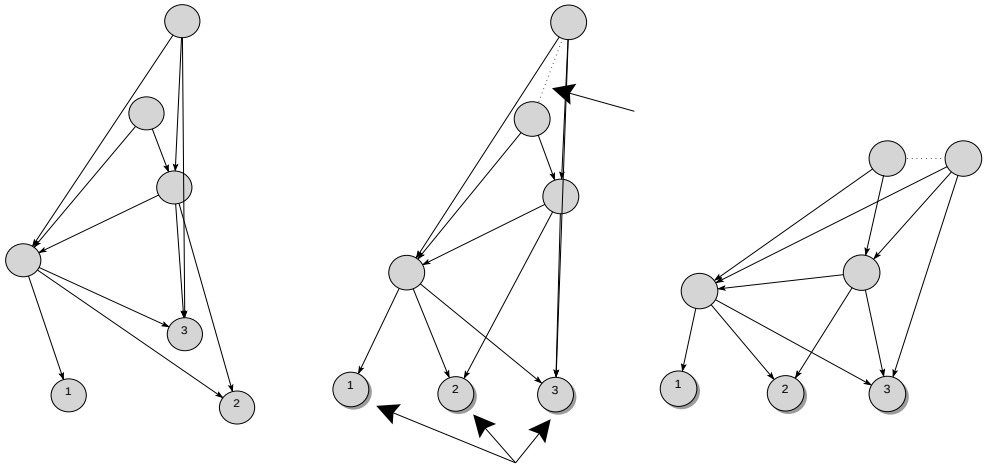
- ユーザの好みを反映した配置

  制約を実時間に対話的に変更できるため、自動配置に手動配置の要素を加えてより望ましい配置を得ることができる。



(a) 最初に得られた解　(b) 200世代経過後の解　(c) 制約の追加　(d) 最終解

図 6.15: 配置の実行例



(a) 最初の解　　(b) 制約の追加　　(c) 最終解

図 6.16: 位置の固定

6.3 遺伝的プログラミングによる配置の好みの自動抽出

6.3.1 図形配置への例示の利用

前節で述べたように、配置規則や評価関数と実際の配置との関係は明らかでないことが多く、好ましい配置を得るためにはこれらを試行錯誤により修正する必要がある。前節の GALAPAGOS を使用すると、望ましくない配置が得られた場合でも配置そのものを修正することが可能であるが、それは一度限りしか有効でない。規則や制約を陽に指定するかわりに、ユーザが正しい配置例をシステムに与えることによりシステムが自動的に一般的な配置規則や制約を推論することができれば、ユーザが明示的にそれらをシステムに与える必要がなくなるため、 2章で述べたように配置システムに例示を利用する各種の試みが行なわれている。 Myers は WYSIWYG エディタにおいて文書整形マクロをユーザの示す例のみから作成するシステムを提案している [61]。 Myers のシステムでは、例えばユーザがセクションタイトルの例をひとつ描くだけで、システムにセクションタイトルの整形マクロを推論させることができる。 Hudson はグラフ配置システムに少数の例を示すだけでシステムが汎化を行ない配置アルゴリズムを計算する Editing By Example システムを提案している [32] 。ユーザの示す少数の例のみからユーザの望む配置規則を推論することは困難なので、システムは考えられる配置規則を複数作成し、それぞれの規則の適用結果をユーザに提示して正しいものを選択させることにより候補数の爆発を防いでいる。また宮下らの IMAGE システム[54][121]は、配置データと配置結果の間の双方向変換機構を使いながら、Hudson のシステムと同様の機構を用いてシステムとユーザが対話しながら配置規則の候補を絞り込んでいく。これらのシステムでは、比較的単純な配置規則をユーザの示す例のみから導くことが可能であるが、多数のヒューリスティクスが使われておりシステム自体が複雑であることに加え、複雑な配置の判断基準を推論することができないという欠点がある。

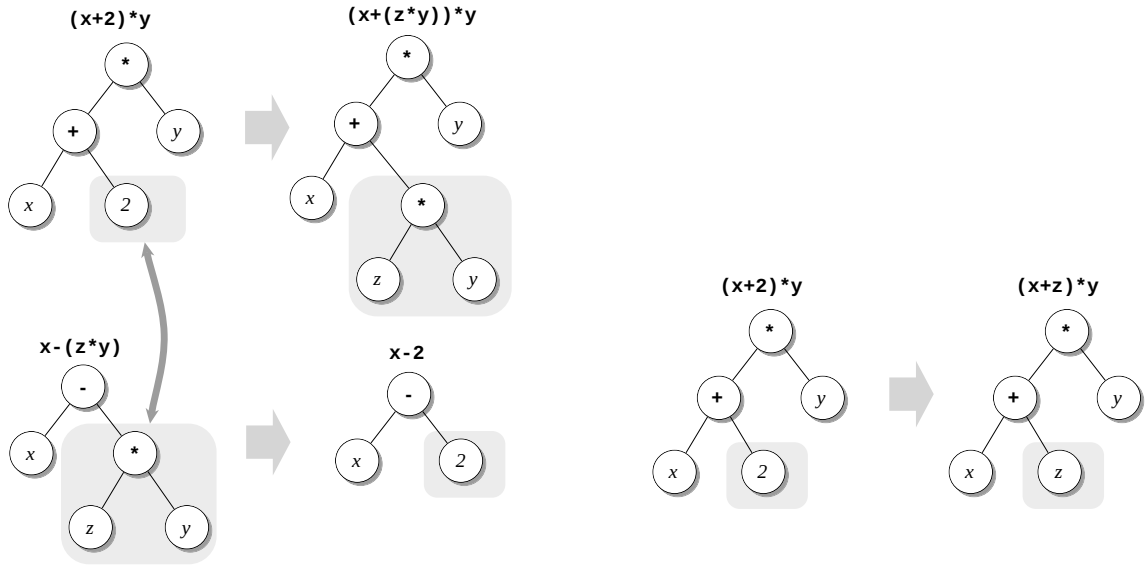
前述のようなアプローチに対し、本節ではユーザの示す例を使用して進化的学習機構により配置規則や制約を抽出する枠組を提案する。ユーザは美しい配置と悪い配置の例の組をいくつかシステムに提示することにより、システムが遺伝的プログラミング[37]の手法を用いて配置の評価関数を自動的に作成する。配置の評価関数は木構造の計算式で表現され、これがユーザの嗜好を反映するように進化することにより適当な配置規則が得られる。ユーザの嗜好を反映する評価関数を一旦得ることができれば、前述のような確率的手法を用いた配置システムを用いて任意のデータに適用することが可能になる。本節ではこの手法を有向グラフの配置問題に適用した例に

ついて述べる。

6.3.2 遺伝的プログラミングによる評価関数の自動生成

遺伝的プログラミング

遺伝的プログラミング (Genetic Programming, GP)[37] は遺伝的アルゴリズムをプログラムに適用したもので、ランダムに生成されたプログラム群がどれだけユーザの目的に近い動作をするかどうかを評価関数として淘汰による進化を行なわせることにより、最終的に目的のプログラムが得られるという手法である。プログラムは通常 Lisp の S 式のような木構造で表現される。最初はランダムにプログラム群の生成を行ない、それぞれのプログラムが目的の仕様とどれだけ近い動作をするかどうかを判定する。仕様に近い動作をするものほど次の世代に生き残る確率が高くなる。このようにして次世代に生き残るプログラムを選択した後、適当な確率で図 6.17(a) のようにプログラムの部分木の交換 (**交差**演算) を行なう。また、いくつかのプログラムについては図 6.17(b) のように部分木を別のランダムなプログラム木で交換する (**突然変異**演算)。これらの操作を何世代も繰り返すうちに、与えた仕様に近い動作をするプログラムが最終的に生成される。



(a) 交叉演算　　(b) 突然変異演算

図 6.17: 遺伝的プログラミングの遺伝的演算

ユーザの示す例からの好みの抽出

遺伝的プログラミングの手法を使うことにより、配置の評価関数をユーザの示す例のみから作成することが可能になる。ユーザは同じ構造をもつグラフの美しい配置の例と悪い配置の例の組をシステムに与える。ある計算式を美しい配置のデータに適用した場合の値が悪い配置のデータに適用した場合よりも大きいとき、その式は美しい配置と悪い配置を見分けている可能性があるが、多く例の組に対して常にその式が美しい方の配置に対して大きな値を計算するようであればさらにその可能性は高くなる。ここでは N 個のグラフを使用し、各グラフ $i \in 1..N$ に対して美しい配置の例 G_i と悪い配置の例 B_i を与える。配置の評価関数 f を進化させるための評価関数として、$E(f) = \sum_{i=1}^{N} p(f,i)$ を使用する。ここで、$f(G_i) > f(B_i)$ のとき $p(f,i) = 1$ であり、それ以外では $p(f,i) = 0$ とする。もし全ての i に対して $f(G_i) > f(B_i)$ となれば $E(f) = N$

となる。遺伝的プログラミング手法を用いて、多くの例に対して $f(G) > f(B)$ を満たす関数 f をみつけることができれば、ユーザの好みを反映する関数 f を得られることが期待される。

6.3.3 実験

評価関数作成の材料となる基本演算子及び引数として図 6.18に示したものを用いた。

演算子

add, sub, mul, div, abs,
sum, max, min, compare

引数

- **ノード位置の x/y 座標**
- **枝の x/y 方向**
- **交差している枝の数**
- **ノード間距離**
- **ノードから出る枝の間の角度の最小値**

図 6.18: 評価関数に用いられる演算子と引数

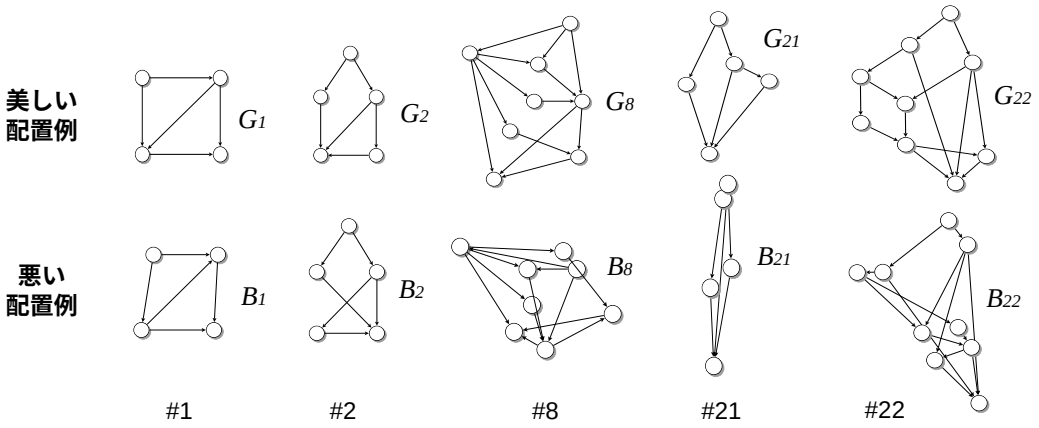


図 6.19: システムに与えられる美しい配置と悪い配置の組

最初はこれらの演算子や定数を用いて評価関数をランダムに生成させる。システムには 22 個の例の組を与えた ($N = 22$)。これらのいくつかを図 6.19に示す。全ての例 i に対し個体群中の全ての関数を適用して $f(G_i)$ と $f(B_i)$ を計算して $E(f)$ を求めている。

世代の経過による $E(f)$ の平均と最高値の進化のようすを図 6.20に示す。ここで、個体数、交差率、突然変異率はそれぞれ 600, 0.8, 0.005 である。交差率や突然変異率は進化の効率に影響が大きい。個体数は多いほど少ない世代で進化がおこるが、それだけ世代毎の計算時間はかかる。

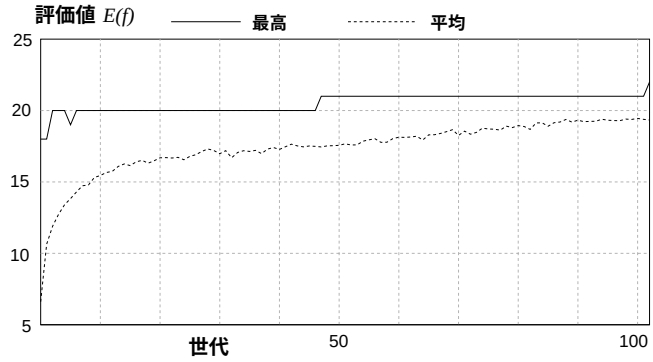


図 6.20: $E(f)$ の最高値と平均値

世代 1 では評価関数はランダムに生成されているので評価値の平均は 7 と低く最高でも 18 であるが、計算が進むにつれ評価関数は進化し、世代 102 において、全ての $i \in 1..22$ に対し $f(G_i) > f(B_i)$ となるような関数 f_b が得られている。 f_b を Lisp プログラム風の型式で表現したものを図 6.21に示す。

```
(ADD (SUB (ADD (MUL (MUL (MUL (ADD (ADD (ADD SUM(diry)
SUM(minangle)) (ADD 44.00 69.00)) (MUL 43.00 MIN(diry))) 5.00) (ADD
(ABS MAX(minangle) MIN(dist)) (ADD (ABS 74.00 MIN(dirx)) (ABS 15.00
SUM(locx))))) SUM(minangle)) (MUL 12.00 (CMP (DIV 57.00 MIN(locx))
(CMP 94.00 MIN(intersec))))) (DIV (ABS (MUL (SUB SUM(locy) 27.00)
(CMP 28.00 65.00)) 62.00) SUM(dirx))) (CMP (ABS (DIV 67.00 SUM(locy))
(CMP (ABS (ABS 73.00 (CMP 67.00 SUM(dist))) MIN(intersec)) (ABS MIN(dist)
MIN(diry)))) MIN(diry)))
```

図 6.21: 計算された評価関数 f_b

得られた f_b を評価関数として用いて GALAPAGOS によりグラフを自動配置した結果を図 6.22に示す。 GALAPAGOS にはノードの接続情報及び f_b のみしか与えていないにもかかわらず、ユーザが例を通じて美しい配置の特徴として示したいくつかの特徴が結果に反映されていることがわかる。例えば図 6.22においてほとんど枝は下を向いているが、これは図 6.19において美しい配置の例として示したグラフのほとんどの枝が下を向いていることを反映していると考えられる。

6.3.4 議論

適応型インタフェースの枠組

本システムでは、ユーザが例を沢山与えるほどシステムの性能が向上する。これは、適応型インタフェース[8] [75] の構築に遺伝的プログラミングの手法が有望であることを示している。適応型インタフェースの構築の試みは、ユーザの操作を履歴情報やコンテキスト情報のみから予測しやすい場合のような単純な場合以外成功しているものが少ない [114]。さらに複雑なシステムではユーザモデルやアプリケーションモデルの知識を利用してシステムがユーザに適応することを試みているが、これらの知識はあらかじめて与えておかなければならないのでそれを構築することがそもそも難しいし、柔軟性も乏しい。これに対し、人工生命や機械学習の考えを用いて、システムがインタラクションを通じて徐々に進化 / 学習することにより適応型システムを実現しようという試みが近年盛んで有望である。例えば Maes の学習型インタフェースエージェン

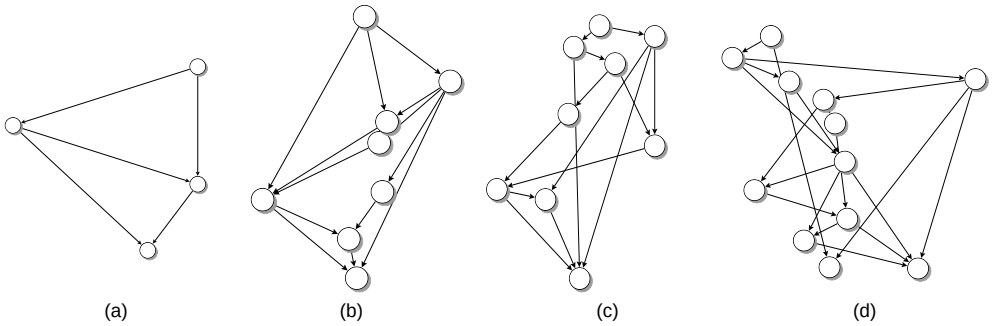


図 6.22: f_b を用いて計算した配置結果

ト [38] [44]はユーザの普段の行動や自分の提案に対するユーザの反応などをモニタしながらメモリベースの学習によりシステムが徐々にユーザに適応していく。このように、進化型システム / 学習型システムは適応インタフェースの重要な枠組として期待がもてる。

遺伝的プログラミングの有効性

遺伝的プログラミングの手法はランダムサーチに比べ本当に有効なのかという疑問が投げかけられることが多い。 6.3.3節の例では、目的の評価関数を得るのに 102 世代が必要であったが、ここで個体数は 600 だったので約 60,000 回評価関数の計算を行なったことになる。一方、遺伝的手法を使わず評価関数をランダムに生成して目的を満たすものが得られるまで繰り返すとすると、目的の関数が得られるまでの回数の期待値は 2^{22} ≃4,000,000 となるので、少なくとも本節の例においては遺伝的手法により完全なランダムサーチよりもはるかに良い結果が得られたことになる。

対話的な例データの作成

システムへの例示により配置の基準を知らせる方式は、配置のアルゴリズムを作成することに比べれば簡単だとはいうものの、多くの例をシステムに提示するのは面倒である。 6.3.1節で紹介したように、 Hudson のシステム [32] や宮下らの IMAGE システム [54][121]のように例の作成をシステムが支援することによりユーザは正しいものを選択するだけでよいというインタフェースが提案されているが、本システムにおいても同様の手法が可能である。いくつかの例からシステムに評価関数を作成させた後、その評価関数と確率的配置システムにより別のグラフの配置を行なわせて、その結果とそれを人手でより良く修正したものの組を新たな例としてシステムに与えることにすればよい。実際、図 6.19の B_{21} は例 #1 から #20 までから作成された評価関数を用いて自動配置を行なったものであり、 B_{22} は #1 から #21 までを用いて自動配置させたものである。

また、例の生成を完全にシステムにまかせて、ユーザは自分の好みのものを選択するだけというインタフェースも可能である。例えばシステムがランダムな基準を用いて配置したグラフを複数ユーザに提示し、ユーザはその中から最も自分の好みに近いものを選択するという作業を何度も繰り返すことにすればシステムはユーザの好みを抽出できるようになるであろう。このような手法は、面白い絵[3]や生物の形[4]などを進化的に生成させるための枠組としてよく用いられてい

[3] Karl Sims、畝見達夫らはこのような手法で芸術作品を生成する試みを行なっている [89]。遺伝的アルゴリズムを使用しない場合でも、乱数を用いてシステムに自動的に作成させた絵の中から優れたものを人間が選ぶという方針で CG 作品を作ることはよく行なわれている。

[4]生物 / 進化学者の R. Dawkins が著書「ブラインド・ウォッチメーカー」 [17] で紹介した BioMorphシステムは、システムがランダムに生成した「虫」の形態の中からユーザが好みのものを選択することを繰り返すという人為淘汰

る。ランダムに近い配置から好みの配置を選ぶことはなかなか難しいと考えられるので今回このような方式は試みなかったが、分野によってはこのような手法が有効な場合があるかもしれない。

過適応と無意味な複雑化の防止

図 6.21の f_b は非常に複雑でありかつ無意味な計算要素をいくつも含んでいる。これは、評価関数を進化させるときにそれが与えた例を正しく判別できるかのみに着目して型式の単純さを考慮に入れなかったことが原因である。$E(f)$ に式の単純さの要素を加えれば複雑化はある程度防止できると考えられるが、目的の評価関数を得るまでの時間は余分にかかるようになる。

処理速度

本システムの最大の欠点は計算に長い時間がかかることである[5]。しかし遺伝的プログラミング / 遺伝的アルゴリズムは簡単に並列化可能であるし、計算機の処理速度の向上する速さや本システムの枠組の強力さから考えると、処理速度の問題は大きな問題ではないと考えられる。

6.3.5 予測 / 例示インタフェースの問題点の検証

本章で提案したシステムにおいて、 3.3.1節で述べた問題点が克服されているかどうか検証する。

1. プログラムを作成できない

   任意のプログラムを評価関数として作成できる。

2. プログラム生成のための指示が面倒

   良い配置と悪い配置の例の組を複数指定してどちらが良いかを示すだけでプログラムが生成される。沢山の例を指定するのは面倒であるが、悪い配置例をシステムに自動生成させ、それを修正したものを良い配置例として使用することにより手間は軽減される。

3. プログラム実行のための指示が面倒

   得られた評価関数は人間が作成した評価関数と全く同様に使用することができる。

4. 正しいプログラムの作成が困難

   例の数が少ないときは意図を正しく反映しない評価関数が生成されてしまうことがあるが、例の数を増やすことにより、なんらかの意味のある評価関数を得ることができる。

5. 実行に関するリスクが大きい

   人間の予測に反する配置が得られる可能性はあるが、これは無害であり、修正操作によりさらに良い評価関数を得ることができる。

6. 予測 / 例示手法を使わない方が楽である可能性がある

   例示を使用しない場合は各種の評価関数を試行錯誤で選ばなければならないが、例示による場合はそのような手間は必要ない。

によりそのユーザの好みの虫を生成するゲームである。各世代においてシステムは現在の虫を変異させた次世代の虫の候補を 9 個ユーザに提示し、ユーザがそのうちひとつを選択することを繰り返すことによりだんだん虫の形態が複雑になっていく。

[5] 6.3.3節の例を 68040 (25MHz) NeXTstation の Objective-C で計算した場合、評価関数が求まるまでに数分を要する。

7. 機能がユーザの邪魔になる

   例示を全く用いない場合でも、評価関数を用意することにより同じ図形配置システムを使用することができる。

8. 大量のデータが必要

   システムに与える例はせいぜい数十個でよい。

9. ヒューリスティクスが多用されている

   評価関数を進化させるときは単純な規則のみが用いられている。

以上のように、 3.3.1節で述べたほとんどの点に関して問題が解決していることがわかる。これを表 3.1に従って表にすると表 6.1のようになる。

	プログラム不可能	定義時の指示が面倒	実行時の指示が面倒	正しい推論が困難	実行リスクが大	時間的リスクが大	勝手な実行が邪魔	大量データが必要	ヒューリスティクスが過度
GP Layout	○	△	○	△	○	○	○	○	○

表 6.1: 予測 / 例示インタフェースの問題点の配置システムにおける解決

6.4 まとめ

本章では、遺伝的プログラミングの手法を単純に適用することにより、ユーザの示した例示データからグラフ配置に関する暗黙的なユーザの好みを評価関数として抽出し再利用することが可能であることを示した。一般に配置の好みは人によって様々であり、それを関数の形で表現することは容易ではないが、本章の手法を使うことにより好みを示す評価関数を自動的に構築できる。

第 7 章　予測 / 例示インタフェース統合アーキテクチャ

7.1　はじめに

予測 / 例示インタフェースを実現するためには、非同期的にユーザ情報を収集 / 解析したり動的にソフトウェアを変更したりしなければならないため、柔軟なソフトウェアアーキテクチャが必要である。本章では、広い範囲のユーザインタフェース構築のための基盤として**共有空間通信**方式が適していることを示し、これを用いることにより予測 / 例示インタフェースを容易に構築できることを示す。

7.2　ユーザインタフェースアーキテクチャ

本節ではユーザインタフェースのソフトウェア作成における問題点及びそれに対する従来の対策について述べる。

7.2.1　ユーザインタフェースソフトウェア構築上の問題点

近年のインタラクティブシステムにおいては、ソフトウェア全体のうちユーザインタフェース部の占める割合は 50% から 80% にものぼることが普通になってきている [66]。ユーザインタフェースのソフトウェアは以下のような点において作成が困難であるといわれている [66]。

対話的設計　インタフェースは実際に使用し評価してみなければよしあしがわからないことが多いため、良いインタフェースを作成するためには何度も繰返し改良を行なうことが必要である。このためにはプログラムの修正とテストを数多く繰り返す必要があるのでシステムの開発に手間と時間がかかってしまう。

グラフィック設計　インタフェースを使いやすくするためにはソフトウェアの見栄えや部品の配置が重要な意味をもつ。人手で試行錯誤的に配置を決めるのは手間がかかるし、 6章で述べたような自動配置システムを作ることもむずかしい。

非同期入力のサポート　システムがどのような状態にあるときでもユーザがシステムを制御する要求を出す可能性があるので、システムは常にユーザの非同期的な入力に注意している必要がある。

並列処理　高度なユーザインタフェースソフトウェアでは並行処理機能が必須である。マウスとキーボードのように独立した複数の入出力装置を非同期的に扱う場合はそれぞれに別々のプロセスを割り当てる必要があるし、テキスト入力枠やボタンなどのインタフェース部品が並んだグラフィックインタフェース画面においては、ユーザが任意の順番で部品を操作可能にするためにはそれぞれのインタフェース部品を並列に動作させなければならない。またユーザをアプリケーショ

ンと独立に動くプロセスと考えると都合が良い場合もあるし、アプリケーション部とユーザインタフェース部の分離にも有効である。

効率　使いやすいシステムは常にユーザの操作に高速に反応する必要があるので、入力に対する応答時間や描画などの出力に要する時間が小さくなるように、常に効率的な実装が要求される。

エラー処理　人間の操作には間違いがつきものであるが、ユーザがどのような間違った操作を行なった場合でもどのような間違いが起こったのかをユーザに知らせたり、正しい状態に回復したりする機能が必要である。ユーザのエラーに対するきめ細かい処理を作成することは、機械的な処理に対するエラー処理に比べはるかに複雑である。

例外処理、undo 機能　ユーザが間違った操作を繰り返した場合でももとの状態に復帰できるようにするため、全ての操作履歴を記憶しておいたり、全ての操作に対して逆操作 (**undo**) が可能なようにするなどの対応が必要である。

7.2.2　プログラミング言語の問題点

前節で述べたようなユーザインタフェースソフトウェアの作成にまつわる問題点は、アプリケーション作成に使用される汎用言語がユーザインタフェース作成に向いていないことに起因するものが多い。一般の汎用プログラミング言語は以下のような点においてユーザインタフェース作成に適していない [63]。

適切な入出力機構の欠如　汎用のプログラミング言語は、端末やファイルシステムとデータのやりとりをする機能のように、かなり制限された入出力機能しかサポートしていないものが多いため、高度な入出力を行なうためには専用のライブラリを多数用意しなければならないのが普通である。

また、一般の手続き型言語では複数の入出力チャネルを同時に扱うことができないものが多い。プログラムは通常ひとつの入出力装置が使用可能になるのを待ち続けなければならず、また複数の装置を使用する場合はどれが空いているかを自力でチェックしなければならない。

これを避けるため、多くのウィンドウシステムではあらゆる入力が同じ構造をもつ「イベント」として一本化されている。この場合、すべての入力はひとつのイベント処理プログラムに送られて処理されることになり、イベント処理プログラムは各イベントを必要に応じて振り分ける作業が必要になる。このような「イベントディスパッチャ」は複雑になりがちである。

図 7.1に X Window Systemのイベント処理の様子を示す。キーボードやマウスなどの装置からの入力はすべて X サーバに送られ、これらはすべてイベントとしてアプリケーションに送られる。アプリケーション内では、イベントディスパッチャが各イベントを識別して キーボードハンドラ、マウスハンドラなどにイベントを振り分ける。このように、もともと別種の入力であるにもかかわらず、システムの都合のため、一緒にまとめたり再分配したりすることが必要になっている。

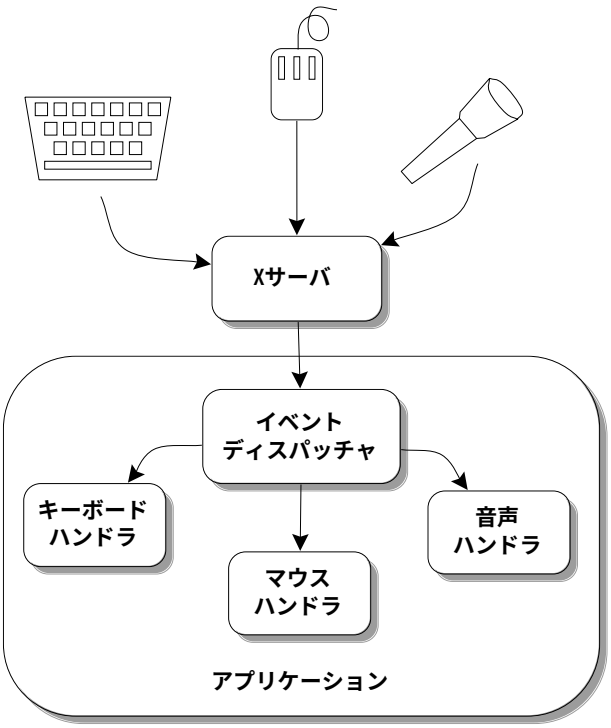


図 7.1: X ウィンドウシステムのイベント処理

マルチプロセス / リアルタイムサポートの欠如　7.2.1節で述べたようにユーザインタフェースソフトウェアには並列処理機能が必須であるにもかかわらず、並列実行をサポートする汎用プログラミング言語は少ない。

オブジェクト指向性の欠如　オブジェクト指向の考え方はユーザインタフェース開発にも非常に有効であるが、汎用言語の多くはオブジェクト指向をサポートしていない。

ラピッドプロトタイピングのサポートの欠如　ユーザインタフェース開発の初期段階においては短い周期で開発とテストを繰り返すことが有効であるが、多くの汎用プログラミング言語はそのような機能を支援していない。

不適当なプログラム表現型式　一般のプログラミング言語はテキストでプログラムを表現するが、ユーザインタフェースソフトウェアではしばしば図によりプログラムを表現する方が都合の良い場合がある。複雑な状態遷移が必要な場合には USE[85]や StateCharts[26]のような図形的状態遷移記述言語が使用できると便利であるし、出力にグラフィクスを多用する場合はプログラム自体をグラフィクスで表現するような言語の使用が便利である。また、テキストで表現される言語を使用するよりも、アイコンを使ったビジュアルプログラミングやインタフェースビルダのような視覚的なプログラミング言語を使用する方が都合が良いことも多い [124]。

ユーザインタフェースに向いた新しい機能の欠如　ユーザインタフェースソフトウェア作成に有効であることが最近の研究により明らかになりつつある**制約解決機能**などは既存の汎用言語に組み込むことがむずかしいという問題がある。

7.2.3 予測 / 例示インタフェースアーキテクチャの要件

4, 5, 6章で述べた予測 / 例示インタフェースシステムを実現するためには、以下のような機能が必要である。

- ユーザの操作履歴の記録

  Dynamic Macro ではユーザ操作履歴の記録が最も重要であるし、他のシステムでもユーザの操作履歴が予測に重要な意味をもつものが多い。

- システムの状態遷移の記録

  複雑な予測 / 例示インタフェースシステムにおいてはシステムの内部状態も予測に使用する必要がある。例えば 6章で述べた配置システムでは、システムが内部に持つ評価関数が外部で参照できなければならない。

- システムの出力の記録

  市販のソフトウェアの場合のようにシステムの内部状態を知ることができない場合でも、出力の記録が例示や予測に使用できることがある [73] 。

- 付帯情報の記録

  Smart Make の場合のように、ファイルの参照 / 更新のような、ユーザとの対話に直接現れない情報が予測に重要であることがある。

- 各種記録情報の解析

  予測のためには上述のような各種記録情報の解析が必要である。各アプリケーションにおいて情報解析を行なうこともできるが、汎用性や負荷の分散を考慮すると、独立して解析を行なう解析モジュールにおいて実行できることが望ましい。

- 予測結果の提示 / 実行

  解析モジュールによる予測結果をユーザに提示したり、ユーザの指示により実行することが可能でなければならない。

- 解析モジュールの交換

  状況に応じて最適な解析モジュールが異なる可能性があるので、モジュールは簡単に変更可能になっていることが望ましい。

7.2.4 従来の対策手法

ユーザインタフェースソフトウェアを楽に構築するために行なわれてきた各種の工夫について以下に解説する [117]。

ツールキット

X ウィンドウや NEXTSTEP などのウィンドウシステムにおいては、各種のインタフェースツールキット (または単にツールキット) と呼ばれるライブラリ群がインタフェースプログラムの作成に広く使用されている。インタフェースツールキットとはテキスト入力枠やボタンなどのグラフィックインタフェース部品の外見及び動作をライブラリとして定義したもので、インタフェース部がプログラムのメインルーチンとなって動作し、各部品に対しマウスやキーボードなどから操作を加えると部品に対し定義された「コールバック」関数がコールバック関数呼び出されるようになっている。インタフェースツールキットではシステムが入力装置からの要求をとりまとめ

てツールキットに渡し、ツールキット側においてその要求がどの部品に対応するのか判断して各部品にイベントを配送するという構造になっているものが多い。

ツールキットを使用すると、複雑なプログラミングがかなりの程度軽減されるし、同じツールキットを使う限りシステムの見栄えが同じようになるため操作が統一化できるという利点があるが、ツールキットは以下のように数多くの問題点を持っている。

制御の流れの問題 多くのツールキットは並列実行をサポートしていないためプログラムの制御の流れに制約を加えてしまうという問題点がある。単純なアプリケーションの場合はコールバック方式により簡単にインタフェースプログラムを構築することができるので便利であるが、複雑な制御が必要なアプリケーションにおいてはアプリケーションをコールバック関数として実装することが困難な場合がある。例えば、ユーザと対話するエキスパートシステムでは、何をユーザに質問するかが実行時の複雑な計算により決まるため、インタラクションの順番などをあらかじめ決めておくことができない。このようにエキスパートシステムはいくつかのコールバック関数として実装することは非常に難しいためツールキットの使用が困難である [43] 。

一般の手続き型プログラミング言語では、アルゴリズムをメインルーチンとして実装する場合とサブルーチンとして実装する場合では型式を変えなければならないのが普通である。これと同じ理由で、端末ベースのインタフェースをもつアプリケーションと、ツールキットを利用するグラフィックベースのインタフェースをもつアプリケーションでは根本的に構造を変えなければならないことが多い。既にメインルーチンとしてアルゴリズムが書かれたアプリケーションをツールキットの都合だけのために書き変えなければならないというのは都合が悪い。

制御の移動が問題になるためツールキットを使いにくい場合の例を図 7.2に示す。この 8-Queen を解くプログラムでは、チェック関数**`extend()`**を再帰的に呼び出しており、再帰呼び出しの最も深いレベルにおいて計算結果が得られ、**`printqueens()`**において順々に結果を出力することができる。しかしこのプログラムのインタフェースを変更し、一般的なツールキットを使って、ユーザがマウスをクリックする度に新しい計算結果が表示されるようにしようとすると、新しい計算結果を求めるようなプログラムをコールバック関数として実現しなければならない。このような関数の型式は図 7.2のものとはかなり異なったものにならざるを得ない [49] 。

```
...
int queens = 8;
main(argc, argv) char **argv;
{
    int i;
    for(i=0;i<queens;i++){ col[i] = qpos[i] = 0; }
    for(i= -queens;i<queens;i++){ up[i] = down[i] = 0; }
    extend(0);
}
extend(n)
{
    int c;
    for(c=0;c<queens;c++){
        if(!col[c] && !up[n+c] && !down[n-c]){
            qpos[n] = c;
            if(n+1 >= queens)
                printqueens();
            else {
                col[c] = up[n+c] = down[n-c] = 1;
                extend(n+1);
                col[c] = up[n+c] = down[n-c] = 0;
            }
        }
    }
}
```

図 7.2: 8-Queen 問題を解く C プログラム

部品の制限　ツールキットに用意されている部品がそのまま利用できる場合は良いが、仕様が違うインタフェースを作ろうとすると非常に手間がかかることがある。このためツールキットではユーザの様々な好みに応じるために各部品に対して沢山のオプションを用意しているのが普通であるが、オプションの数が多くなって設定が繁雑になるにもかかわらず、一般的でないインタフェースを作ろうとするとオプションでは対応しきれず苦労することになる。

使用言語の制約　ツールキットは特定の言語や描画システムと密接に結び付いているのが普通で、他のものを使うことができない。例えば NeXT のツールキット「Appkit」を使用するということはすなわち UNIX, Objective-C, Display PostScript の採用も意味してしまう。ところが現実にはいろいろな言語を混ぜて使えた方が便利なことが多い。同じツールキットを Lisp や Perl のようなインタプリタ言語で使えると便利だろうし、 PostScript を使いたいと思う Visual Basic ユーザも多いであろうが、現状ではこういうことは不可能である。このように既存のツールキットではいろいろなツールや言語を組み合わせて使うための機能が一般に弱いため、システムを総合的に判断して有利なツールを選択し、うまくいかないところは目をつぶらざるを得ず、良いツールキットを使いたいために嫌でも指定された言語を使うというようなことになってしまっている。

インタフェースビルダ

対話的にグラフィック設計を行なうためのツールとして、いわゆる「インタフェースビルダ」が近年広く用いられるようになっている。インタフェースビルダという名前は NeXT 社のワークステーションに登載されている「Interface Builder」のようなシステムという意味で一般に用いられているが、ここではウィンドウシステムのツールキット部品を直接操作によりウィンドウ画面に配置しながらアプリケーションの外見を設計し、かつツールを操作したときのアプリケーションの動作記述を支援するシステムのことを指す。

インタフェースビルダでは、直接操作インタフェースによるグラフィック画面設計とテキスト編集によるプログラミングを同時進行させながらアプリケーションを作成する。ツールキットの使用が前堤となっており、ボタンやスライダのようなインタフェース部品それぞれに対し、それらの部品が操作されたときに呼ばれるコールバックルーチンを対応付ける。コールバックルーチンの雛型やインタフェース部品の初期化のためなどのコードは自動的に生成され、コールバックルーチンをきちんと書かなくてもアプリケーションの見かけの動作をテストすることができる。またアプリケーション作成のプロジェクト全体を管理する機能をもつものもある。

インタフェースビルダは最も成功したビジュアルプログラミングシステムのひとつということができる。マウスなどでツール部品を画面に配置することはテキストで指定するのに比べ圧倒的に楽なので、画面の配置などはビジュアルに行ないプログラム本体はテキストエディタを使って書くという風に、ビジュアルな部分とテキスト編集部分を相補的に使用することができる。またテスト機能によりラピッドプロトタイピングを支援しているし統合化プログラミング環境としても優れたものが多い。

インタフェースビルダは一般のツールキットをかなり使いやすく改善してはいるものの、前節で述べたようなツールキットの弱点をそのまま継承しているうえに以下のような問題点を持っている [109]。

一枚岩の環境　インタフェースビルダを中心としたプログラム開発環境は一枚岩の巨大な統合的環境になっているものが多い。実際インタフェースビルダの多くはプロジェクト管理機能をもっており、プログラム編集・コンパイル・デバッグなどプログラム開発における各側面をサポートするようになっている。これは UNIX に代表されるような、小さなツールの集合体でプログラミング環境を構成するという近年のソフトウェア開発のアプローチに反している。巨大な環境は便利なことも多いが、様々な要求に答えるようカスタマイズを行なうことがむずかしいし、また全貌を把握しにくいという問題がある。

ビジュアル部とテキスト部の非互換性　大抵のインタフェースビルダはビジュアルに編集した部分をプログラムなどのテキストに変換する機能をもっている。これらの部分の役割の区別が明確ならばよいが、どちらを使っても同じことができることも多いのでしばしば混乱が起こる。またインタフェースビルダがプログラムを生成するときは通常雛型の生成だけを行ない、ユーザがこれを編集して最終的なプログラムを作成することになるが、この方式では以前修正したテキストを新しい雛型で上書きしてしまう危険があるし、テキストの変更がビジュアルな部分に反映されないのでビジュアル部とテキスト部の対応を常に保つように注意しなければならない。

直接操作の非効率性　直接操作によるグラフィック操作作業はテキスト編集による作業より効率が劣ることがある。例えばウィンドウが大量に必要な巨大なアプリケーションを作るのにインタフェースビルダを使用するとき、同じような操作を何度も何度もマウスで実行しなければならないことがある。また既存のデータや別のプログラムからインタフェース画面を自動生成することもできないのでインタフェースビルダでは手工業的作業しかできない。

WIMP インタフェース以外に非対応　インタフェースビルダで設計できるのはグラフィックディスプレイを持つ卓上計算機上のマウスとキーボードを使うインタフェースだけであり、音声を使うインタフェースやペンを使う携帯端末などのインタフェース作成には役にたたない。

動的設計への非対応　インタフェースビルダで設計できることはインタフェースのほんの一部である。インタフェースビルダを使っても図形エディタを作れるわけではないし、アニメーション

を記述できるわけでもない。結局インタフェースビルダは静的なツール部品を配置することしかできないわけで、「レイアウトビルダ」と言った方が正確であろう。ユーザインタフェースの作成にでは静的なものよりも動的な反応の作成が難しいが、インタフェースビルダはインタフェースの状態遷移の設計などを支援しない。

並列言語

7.2.1節で述べたように、並列処理がインタフェースソフトウェア作成に有効であることは広く認識されているため、並列処理機能をもつインタフェース記述言語やユーザインタフェース管理システム (User Interface Management System: UIMS) が従来より数多く提案されている。

明示的にユーザインタフェース専用の並列処理言語を使用した例として、 CSP[30]を使ったインタフェース記述言語 Squeak[9], Newsqueak[71]や、 ERLというイベント処理言語を使用した Sassafras UIMS[28]などがある。

暗黙的に並列処理を行なっているものとしては制約記述言語やユーザインタフェースツールキットがあげられる。近年のグラフィックインタフェース作成システムでは、グラフィックオブジェクトの間に制約を設定しオブジェクトの位置などの変化により自動的に別のオブジェクトが更新されるというものが数多く提案されているが [6][27][31][79][88]、オブジェクト間の制約は暗黙的な並列プロセスが制約の数だけ存在することと同等である [4]。またツールキットにおいてはイベント配送システムがインタフェース部品の並行動作を支援していることになる。

以上のような並列処理表現は、特定の応用には便利であるものの、汎用のインタフェース記述方式としては不充分である。例えば制約解決システムは図形の配置やスプレッドシートのようなアプリケーションには便利であるがプログラムモジュール間の複雑な通信を表現することはむずかしい。 Squeak や ERL のようなユーザインタフェース記述言語を使用すると明示的に並列動作の記述をきめこまかく行なえるという利点はあるが、並列処理だけのために特殊な言語を使用する必要があるためアプリケーションプログラムの開発の障害となる。インタフェースツールキットはグラフィックインタフェース作成に現在最も広く使用されているが、前述のように、プログラムの制御移動の枠組が固定的で柔軟性に乏しく、部品は常にイベント配送システムから呼び出されるという形になるためプログラム構造が制約をうけてしまい、アプリケーションを作成しにくくなるという欠点があるし [43][49]、並行動作する部品間で情報を交換することが簡単でないことが多い。

以上と異なるアプローチとして、既存のアプリケーション記述言語に並行処理機能を追加するという方法が考えられる。言語仕様に並行処理機能を追加する試みは従来より数多く行なわれてきたが、そのいずれもが処理系に大きな変更を加えるものであり広く一般的に使用されているものは少ない。これに対し、言語に並行処理ライブラリを追加することにより既存の言語に並行処理機能を加えるというアプローチがある。この手法では言語仕様はほとんど変更せず、並行処理ライブラリが呼び出されたときのみプロセス切替などの処理を行なう。このようなものの例として Linda がある。 Linda は数個のプリミティブから成る並行処理モデルで、既存の言語にライブラリとして追加することにより比較的簡単に言語を並行動作可能にすることができる。次節では Linda について簡単に解説し、 Linda がユーザインタフェースソフトウェアの構築に都合が良いことを示す。

7.3 共有空間通信によるユーザインタフェース

7.2.1節で述べた各種の問題は、共有された空間を介した通信を指示する柔軟な通信プリミティブにより疎に結合された複数のプロセスを使用してユーザインタフェースソフトウェアを構築す

ることにより解決される。本節では、並列言語プリミティブ Linda[10] [11]の使用によりユーザインタフェースソフトウェア構築における多くの問題点が解決されることを示す。

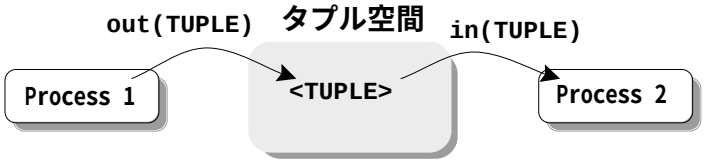
7.3.1 共有空間通信モデル Linda

Linda [11] [20] は、タプル空間(Tuple Space) と呼ばれる共有空間を用いて複数プロセスが通信を行なう並列処理モデルである。一般の共有メモリを用いたプロセス間通信と異なり、 Linda では共有空間に対するデータの書込み / 読出し操作においてプロセスの同期が行なわれる[1]。この際共有メモリ上のデータに対しパタンマッチングを行ない、連想メモリのようにデータの内容にもとづいて書込み / 読出しを行なう。タプル空間は全てのプロセスにより共有されており、各プロセスはタプル空間に対してタプルと呼ばれるデータ組の入出力を行なうことができる。タプルはひとつのラベルと任意個のパラメタから構成される。ラベルは任意長の文字列で、パラメタは任意の型の変数または定数である。各プロセスはタプル空間に対し以下の操作を行なうことができる。

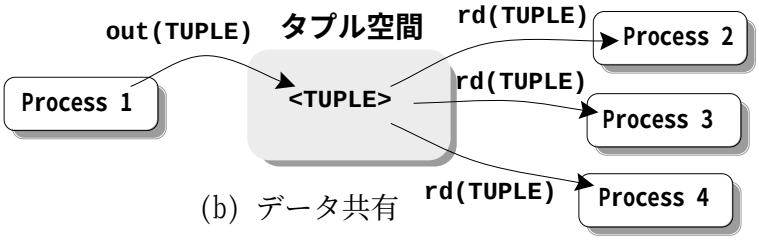
out(*Label*, P_1, P_2, ... P_n**)**
in(*Label*, P_1, P_2, ... P_n**)**
rd(*Label*, P_1, P_2, ... P_n**)**

out(*Label*, P_1, P_2, ... P_n**)** は新しいタプル**<***Label*, P_1, P_2, ... P_n**>** を生成してタプル空間に格納する。プロセスは**in(***Label*, P_1, P_2, ... P_n**)** または**rd(***Label*, P_1, P_2, ... P_n**)** により、マッチするタプルを非決定的にタプル空間から読み出すことができる。タプル空間内のタプルと**in** または**rd** のラベルが一致しかつパラメタの型と値が一致するときマッチしたと判断される。パラメタ中の変数と定数は同じ型ならば常にマッチし、マッチング後定数が変数に代入される。**in** は処理後タプルをタプル空間から削除するが**rd** は削除しない。目的のタプルがみつかるまで**in**, **rd** はブロックする。**out** はブロックしない。

Linda のプリミティブは以上のように単純であるが、図 7.3のように様々な形態のプロセス間通信を行なうことができる。



(a) メッセージ通信



(b) データ共有

図 7.3: タプル空間を使った通信

[1]例えば UNIX System V では複数プロセスでデータを共有するために共有メモリを使用することができるが、プロセス間の同期をとるためにはセマフォなど別の機構を使う必要がある。

図 7.3(a) は一対一のメッセージ通信、図 7.3(b) は複数プロセスによるデータ共有の様子を示している。ここでデータの送信者は受信者を指定していない。また受信者はタプルの内容によって選択的に受信を行なうためメッセージの送信者よりもメッセージの内容にもとづいて通信を行なっていることになる。このように、多くの通信モデルと異なり、相手プロセス名や通信チャネルを意識せずに通信を行なうことができるためプロセスのモジュール化が容易になっている。またタプルは**in** により読込まれない限りタプル空間内に存在し続けるため、送受信者とも相手の状態を考慮せずにタプル空間に対するアクセスを行なうことができ、非同期的情報伝達・情報共有が可能である。以下では Linda にもとづいたモデルを共有空間通信と呼ぶことにする。

7.3.2 共有空間通信をインタフェースに適用する利点

従来の方式に比べ共有空間通信方式をユーザインタフェースプログラムに用いることには以下のような利点がある [46][49] 。

モジュールの独立性 Linda の各プロセスは共有空間内データを介して疎な結合による通信を行なう。例えば入力装置それぞれに対しその処理プロセスを割り当て、結果のみを共有空間に書き出すようにしておくことにより、入力処理部が独立し、キーボードによる文字入力モジュールとペンによる手書き文字入力モジュールを交換して使うといったことが可能になる。また入力モジュールのかわりに自動的に文字列を生成するモジュールを使用すれば、システムの自動テストを行なうこともできるし自動デモにも有用である。このようにユーザや入出力モジュールをシステム内の単なるモジュールとして他のモジュール同様に独立して扱うことができるため、インタフェース部とアプリケーション本体を分離させることが可能である (図 7.4)。

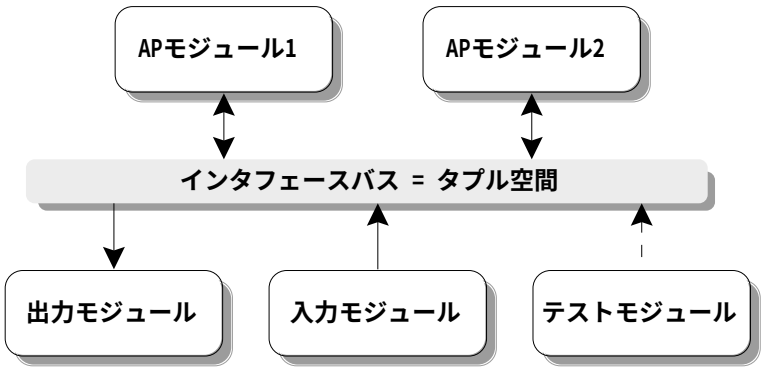


図 7.4: タプル空間によるモジュールの分離

制御の主体の柔軟性 各プロセスはそれぞれ独立した制御の主体となって動作するため 7.2.4節で述べたような制御移動の問題は発生しない。プロセスは処理を他プロセスに依頼した後そのまま実行を続けることもできるし、他プロセスからのイベントを待ち続けることも可能である。このように、現存のユーザインタフェースツールキットで採用されているようなイベントとコールバック関数を用いた型式にもアプリケーションがインタフェースを呼びだす型式にも対応可能である。

名前と年齢を入力するとその人間の情報を返すという非常に簡単なデータベースプログラムについて考えてみる。

```
...
printf("name? ");
scanf("%s", name);
printf("age? ");
scanf("%d", &age);
result = search(name,age);
printf("%s\n",result);
...
```

図 7.5: データベース検索を行なう C プログラム

図 7.5に示した単純な端末インタフェースを持つ C プログラムでは、インタフェース部 (**scanf()**, **printf()**) とアプリケーション部 (**search()**) は交互に並んでおり、これらは全く分離されていない。

このプログラムは Clinda [2]を用いて容易に書き直すことができ、図 7.6のようなインタフェース部とアプリケーション部が分離したプログラムとすることができる。

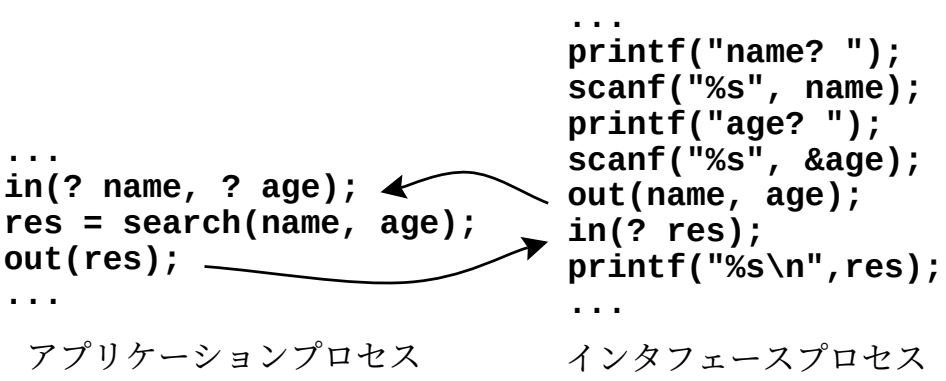


図 7.6: データベース検索を行なう CLinda プログラム

ここでは左側のプログラムはアプリケーション部に対応しており、右側はインタフェース部に対応している。この例ではアプリケーション部とインタフェース部はコルーチンとして動作し、それぞれが自分の制御構造を保っている。文字列と整数の組で構成されるタプルを使用する限り、端末ベースであろうとウィンドウベースであろうと、任意の形のインタフェースプログラムをこのアプリケーションと組みあわせて使用することができる。このように、 Linda を用いることにより、アプリケーション部とインタフェース部を簡単かつ明瞭に分離することが可能である。

以下にアプリケーションとユーザインタフェースの分離を行なうもうひとつの例を示す。図 7.2に 8-Queen 問題を解く C プログラムを示したが、 7.2.4節で述べたように、一般的なツールキットを使って、ユーザがマウスをクリックする度に新しい計算結果が表示されるようにこのプログラムを変更することは困難である。

[2]Clinda は Linda を C 上に実装したものである。本章では全ての Linda プログラム例に Clinda を使用するが、 C 以外の言語上に実装した Linda に対しても同じ議論が成立する。

```
...
int queens = 8;
main(argc, argv) char **argv;
{
    int i;
    for(i=0;i<queens;i++){ col[i] = qpos[i] = 0; }
    for(i= -queens;i<queens;i++){ up[i] = down[i] = 0; }
    extend(0);
}
extend(n)
{
    int c;
    for(c=0;c<queens;c++){
        if(!col[c] && !up[n+c] && !down[n-c]){
            qpos[n] = c;
            if(n+1 >= queens)
                printqueens();
            else {
                col[c] = up[n+c] = down[n-c] = 1;
                extend(n+1);
                col[c] = up[n+c] = down[n-c] = 0;
            }
        }
    }
}
```

図 7.7: 8-Queen 問題を解く C プログラム (再掲)

しかし Linda を使用すると、このプログラムを入力部 / 出力部 / 計算部に分離してそのような動作を行なわせることは簡単に行なえる。この場合は**printqueens()** を以下のように変更するだけでよい。

```
in(CLICK);
out(qpos[0],qpos[1],qpos[2],...qpos[7]);
```

入力プロセスはマウスクリックの度に**out(CLICK)** をタプル空間に出力する。出力プロセスの方は常に解を示すタプルを**in(? p1, ? p2, ..., ? p8)** で監視し続け、見つかればクイーンの位置を示すこれらの値を適当な形式で出力する。このような少しの修正により、8-Queen 問題を解く図 7.7のプログラムは図 7.8と同じタプルインタフェースをもつ任意のインタフェースプログラムと使用することができる。

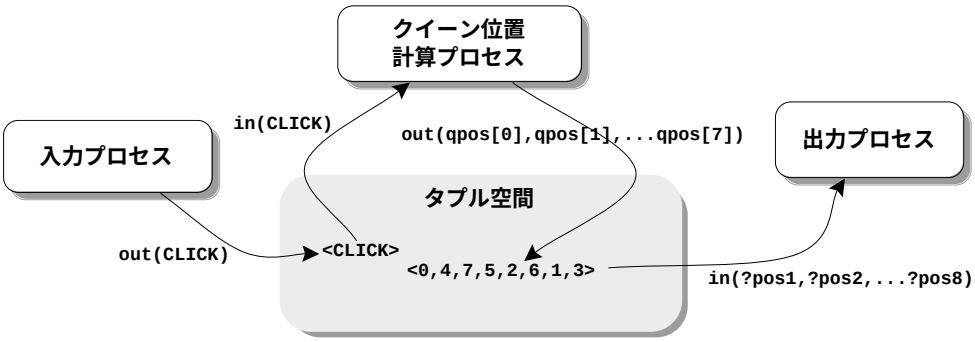


図 7.8: 8-Queen 問題におけるタプル交換

Linda の仕様は単純で柔軟なので、コルーチン以外の協調方式も容易に実現できる。また Linda はパタンマッチ機能も持っているため、さらに複雑なプロセス間通信方式も可能である。

他システムとの協調　制約解決システムのように特定分野で有用な別システムをモジュールとして独立に動作させ共有空間を介して通信することが可能なので、既に開発されている各種の有用なツール資産を活用することができる。

データの粒度、レベルの柔軟性　共有されるデータの型式は任意である。単純な文字列や座標のようなものでもよいし高度な内容をもつメッセージでもよい。複数のエージェントが協調して知的作業を行なうというモデルが各種提案されているが (例えば [69])、これは共有空間通信の枠組で実現できる。

各種言語の協調　Clinda は Linda の特長と C の特長の両者を備えているが、 Linda を介することにより各種の言語を協調的に使用することができるようになる。この点については次節で詳しく述べる。

7.3.3　異なる言語の融合

他の言語との共存

ユーザインタフェースソフトウェアには様々な機能が要求され単一の汎用言語で全てを処理することはむずかしいため様々な言語や手法が提案されているが [63]、 Linda は数個の簡単なプリミティブで構成され、既存のいろいろな言語上でライブラリの形で融合して使うことができるため、複数の言語を協調させて使うことが容易にできる [49]。たとえばインタフェース記述専用言語とアプリケーション作成用言語の共用が可能になる。

従来の計算機言語は、ある範囲の問題を効率良く解くことを目標に作られている。例えば C は効率良い計算やシステム処理を目標としているし、 Lisp は複雑なデータ構造の処理に向いている。データベースを扱うアプリケーションでは、アプリケーションを記述するためには汎用言語以外に SQL も使用することが望ましいかもしれない。一方、ユーザインタフェースのソフトウェアでは高速な計算機能、グラフィクス機能、インタラクション機能など数多くの機能が要求されるため、全ての要求を満たすような小さな言語を設計することは難しい。このためひとつの言語で全システムを記述するよりも、複数の言語を組み合わせて使用することの方が好ましい。このように、問題それぞれについて適した言語が存在するので、ユーザインタフェースソフトウェア構築にあたっては各種の言語を協調的に使用することが望ましいと考えられる。

複数の言語を組み合わせて使用する手法はいくつか種類があり、実際に効果的に使用されているこも多い。例えばコンパイラ生成システム YACCや SMCC[99] では、 BNF 記法に似た言語と属性文法を用いた構文と意味の記述が C の構文解析プログラムに変換されるが、もとの記述の中に C プログラムをそのまま埋め込むことができるので、 BNF 記法の利点と C の利点をともに利用することができる。コンパイラ作成に必要な全ての要素をひとつの言語に盛り込もうとしたシステムも研究されているが、複数言語を併用する YACC のような方式の方が実際には便利なことが多い。また複数の言語を使用してユーザインタフェースを構成している例として SUN の NeWSウィンドウシステムや NeXT の NEXTSTEPシステムがある。 NeWS では画面への表示やユーザ入力の処理には拡張 PostScript を使用し、アプリケーションの記述には C を使うことができる。また NEXTSTEP ではやはり PostScript を拡張した Display PostScriptを使用することができる。 NEXTSTEP ではユーザ入力は全て C のライブラリで処理し、画面への出力は PostScript の描画関数をを C ライブラリにしたものを使用することができる。このように NeWS や NEXTSTEP ではアプリケーション作成に有利な C の利点と描画に有利な PostScript の利点を組み合わせて使用することができるが、言語のタイプがかなり異なるため一方が他方をライブラリのように使用しなければならず、整合性が良いとはいえない。これに対し共有空間通信を用いて複数の言語間

で通信を行なう場合は、統一的な単純なデータ型式でプロセス間が結合されるため、言語間の非整合が問題になりにくい。

既存言語のパワーアップ

7.2.2節において一般のプログラミング言語でユーザインタフェースを作成する場合の問題点について述べたが、ここでは一般プログラミング言語と Linda を組み合わせて使用することによりいくつかの問題が解決されることを示す。

適切な入出力機構の欠如 7.2.2節で述べたように、一般の手続き型言語は単純な文字列入出力しかサポートしていないのが普通であるが、 Linda で結合された複数の入出力プロセスを使用することにより複雑な入出力装置を同時に扱うことができるようになる。また Linda のパタンマッチ機構を柔軟な入力プリミティブとして使用することができる。

Squeak[9]や Sassafras UIMS[28]でも同様に CSP や ERL を使用して複数の入出力を同時に扱うことができるが、これらの並列言語は Linda ほど柔軟ではないうえに、既存の言語と組み合わせて使用できないという欠点がある。

マルチプロセス / リアルタイムサポートの欠如 現在リアルタイム処理を支援する言語や OS はあまり広く使用されていないが、マルチメディアデータの処理にはリアルタイム処理が不可欠なので将来のユーザインタフェース作成においてはそのような言語やシステムが普及すると考えられる。そのような場合にも Linda をプロセス間通信方式のひとつとして採用することは容易である。

オブジェクト指向性の欠如 Linda は任意の手続き型言語においてライブラリとして簡単に使用することができるので、 CLOS や C++ のようなオジェクト指向プログラミング言語と組みあわせることにより使いやすいインタフェースツールを構築することができる。また、並列プログラミングは本質的にある程度のオブジェクト指向性を持っているため [19] 、 Linda の枠組はオブジェクト指向にうまく適合している。

ラピッドプロトタイピングのサポートの欠如 ラピッドプロトタイピングをサポートするため各種のツールが存在するが、ユーザインタフェース専用のツールを使用するよりもラピッドプロトタイピングに向いた汎用の言語を組み合わせて使用する方が有用である。例えば状態遷移記述言語 Flex[47][110]はユーザインタフェースのダイアログの状態遷移を記述するのに有用であるが、ユーザインタフェース以外の多くのアプリケーションにおいても同様に有用であるので、ユーザインタフェース開発専用とせずより汎用の仕様になっている。 Linda を用いると汎用のラピッドプロトタイピングツールを容易に組み合わせて使用することができる。

不適当なプログラム表現型式 7.2.2節で紹介したビジュアルプログラミングシステムや、 7.2.4節で紹介したインタフェースビルダを用いるとユーザインタフェース作成における一部分を簡単化することができるが、特にアプリケーション本体においては従来の手続き型言語の使用が最も効率が良いことも多い。ユーザインタフェース開発の全部にひとつのプログラム表現形式を用いるよりも、 Linda により様々のツールを組み合わせて使用する方が有用と考えられる。

ユーザインタフェースに向いた新しい機能の欠如 制約解決機構や推論機構のような有用ではあるが特殊な機能についても同様に、おのおの別のプロセスとして実現してこれらを Linda により

組み合わせて使用することができる。

7.4 共有空間通信方式の適用例

7.4.1 並列ツールキット

Linda を使用して並列ツールキットを簡単に構築することができる [47]。ウィンドウシステム [101]の Linda 並列ツールキットを使用してプリティプリンタ[103] [105]の印刷フォーマット指定アプリケーションを作成した例を図 7.9に示す。

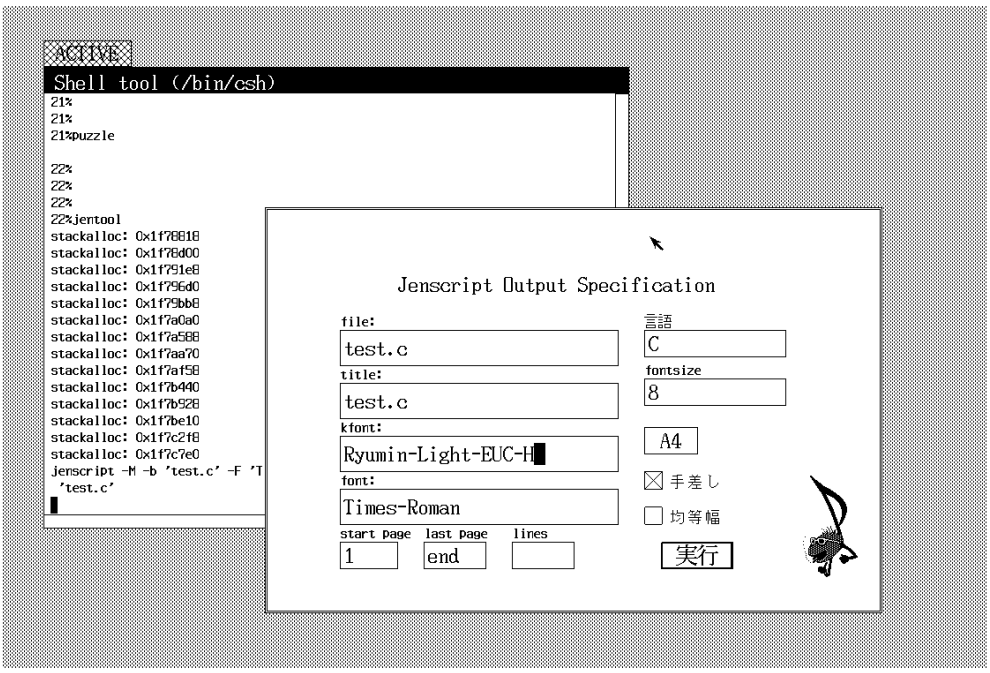


図 7.9: 文書印刷フォーマット指定アプリケーション

カーソルのある文字枠 (テキストボックス) には文字列を入力することができる。それぞれの枠 (ツール) にひとつの Linda プロセスが対応しておりそれぞれの状態を保持しているため、例えばひとつの文字枠内で仮名漢字変換を実行途中であっても別の文字枠内で異なる漢字変換を開始することができる。「手差し」の横の四角の枠 (チェックボックス) もひとつの Linda プロセスに対応しており、マウスクリック毎に選択 / 非選択が切り替わる。

各 Linda プロセスは以下のようなループを実行し続ける。

```
for(;;){
     in(toolid,? eventname, ? arg);
     // ツールに対応した処理の実行
}
```

このように、各ツールは通常タプルを待ち続けている。

チェックボックスのような単純なツールは以下のように実現される。

```
for(;;){
     int value;
     in(toolid,? eventname, ? arg);
```

```
        if(eventname != MouseDown) continue;
        in(toolid, ? value)
        value = !value; // value を反転
        out(toolid, value)
        // value の値に応じてチェックマーク描画または消去
    }
```

ツールオブジェクトが生成された後、各ツールは上のような無限ループを実行し続け、タプル空間を介して他のツールと通信を行なう。チェックボックスの値は**rd()**を使う以下のようなプログラムで読み取ることができる。

```
int v;
rd(cb, ? v); // 現在の v の値を得る
```

このツールキットを使用して作成したビットマップエディタを図 7.10に示す。

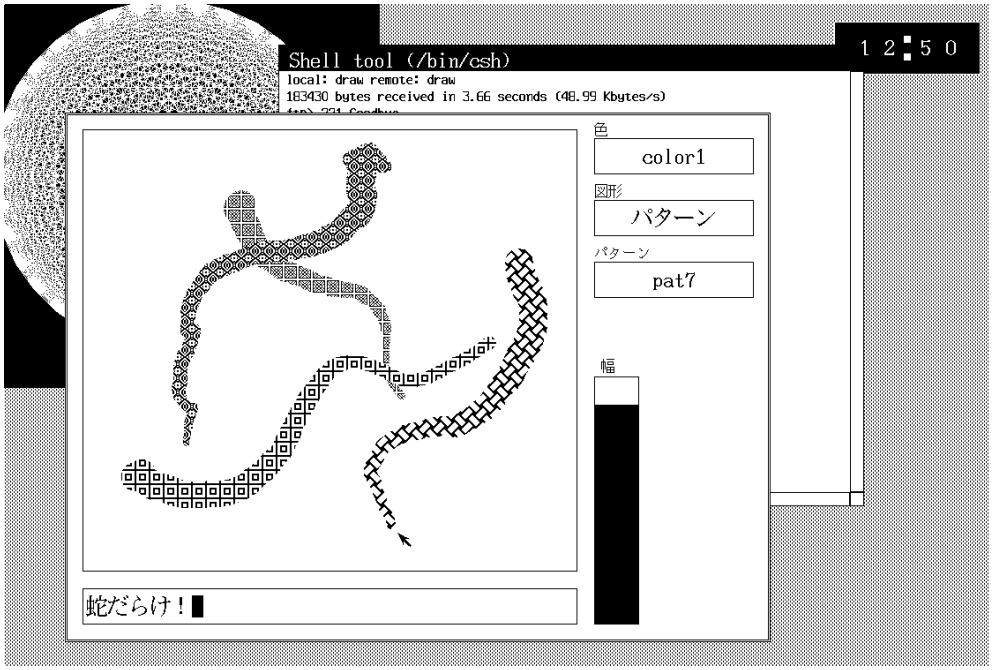


図 7.10: ビットマップエディタ

右端のスライダで描画の線の幅を設定することができるが、これはマウス及びキーボードの両方で制御することができるので、マウスで線を描きながらキーボードで幅を変化させることにより蛇のような絵を書くことができる。幅を制御するツールと描画を行なっているツールが並行動作しているためにこのようなことが可能になっている [29]。

アプリケーションの中心となる描画プロセスはマウスボタンが押されたことを示すタプルを無限ループで待ち続けている。マウスが押されたというタプルを受け取ると、現在どの色が選択されているかを調べ、マウスの位置情報及びスライダで指示される幅情報をタプル空間から参照しながらマウスの位置に円を書き続ける。各ツールが並列に動作しているためこのように単純にアプリケーションを作成することができるが、イベント駆動のシングルスレッドのツールキットを使用するとすると、マウスが押されたりドラッグされたりする度にメインルーチンの描画プログラムが起動されるため、常にマウスの現在の状態を把握しておかなければならないし、これを

避けようとしてマウスドラッグ中は全てのイベントを描画プログラムが受け取るようにすると描画中に線の太さなどを変更できなくなってしまう。

本ツールキットはアプリケーションの制御構造に影響を与えることが無く、アプリケーションとインタフェースの分離を支援する。図 7.7の 8-Queen プログラムは、端末インタフェースでも本ツールキットでも使用することができる。図 7.11に端末ベースインタフェースのプログラムとツールキットベースのプログラムを示す。 同じアプリケーションに対し異なるインタフェースを使用できることからわかるように、アプリケーション部とインタフェース部は完全に分離されている。

```
<定義>
main()
{
   <初期化>
   eval(extend(0)); // 8-Queen計算プロセス起動
   for(;;){
      // 入力は存在せず、解が見つかる度に印刷される。
      out(CLICK);
      in(?p1,?p2,?p3,?p4,?p5,?p6,?p7,?p8);
      printf("%d %d %d %d %d %d %d %d\n",
              p1,p2,p3,p4,p5,p6,p7,p8);
   }
}
```

(a) 端末インタフェースを持つ8-Queenプログラム

```
<定義>
int click();
ExecBox *e;
main()
{
   <ツールキット初期化>
   eval(extend(0)); // 8-Queen計算プロセス起動
   // 「実行ボックス」を生成し、マウスでクリック
   // される度にclick()が起動されるようにする。
   // プロセスは自動的に生成される。
   e = new ExecBox(posx,posy,w,h,click);
   for(;;){
      in(?p1,?p2,?p3,?p4,?p5,?p6,?p7,?p8);
      <ウィンドウにチェス盤を表示し、p1..p8の
       対応する位置にクイーンを表示する。>
   }
}
click()  // マウスクリックにより起動される
{
   out(CLICK);
}
```

(b) ウィンドウインタフェースを持つ8-Queenプログラム

図 7.11: 8-Queen プログラムの端末ベースインタフェースとツールキットインタフェース

7.4.2 共有空間通信の CSCW への応用

共有空間通信方式は以下の点においてグループワーク支援ソフトウェアの構築にも有用であ

る。

- モデルの適合性

  複数プロセスのデータ共有にもとづいたモデルであるため、アプリケーション間でのデータ共有が必須であるグループワークと相性が良い。またデータ共有とプロセスの同期を同時に実現できるため共有データの扱いが楽である。

- シングルユーザのアプリケーションと複数ユーザのアプリケーション

  シングルユーザのアプリケーションがすでに共有空間通信方式を用いている場合、これを複数ユーザ用に変更することが簡単にできる。

- 異なるプラットフォームの共存

  共有データの型式を定めておけば異なるプラットフォームのマシン間でもグループワークが可能である[3]。共有データ構造とその操作方法さえ決まっていればそれを使うアプリケーションの外見はどのようなものでもかまわない。例えばデータを共有するエディタにおいて、一方のシステムが文字端末を使い、他方のシステムがグラフィック端末を使うといったことも可能である。共有データのネットワーク上の実現方式は様々な手法が考えられるが、どの方式を採用する場合でもアプリケーションプログラム本体は変更しなくてよい。

- 共有データの集中管理方式

  ウィンドウシステムベースのいくつかの CSCW システムでは効率の改善のために複製方式 (replicated architecture) を採用している [12]。複製方式とはネットワークで接続された全てのマシン上で同じアプリケーションプログラムを並列に実行させ、それら全てに同じ入力が伝わるように制御することにより結果的にグループメンバ全員に同じ画面が見えるようにする方式である。この方式ではアプリケーションの状態の整合性を保つことが困難であるが、共有空間通信方式ではつねにユーザが同じデータを共有するためこのような問題が生じない[4]。

- 非同期グループワークへの対応

  CSCW システムは、電子会議のようにグループメンバが同時に仕事を行なう同期型システムと、電子メールのように各人が別のタイミングで仕事を行なう非同期型システムの 2 種類に大きく分類することができる。既存の CSCW ツールキットの多くはこのどちらかのみしか支援していないが、共有空間通信モデルでは共有空間中にデータが残るため同期 / 非同期両方のシステムに対応可能である。

共有空間通信を使用した電子会議システム

共有空間通信方式による CSCW システムの例として電子会議システムを実装した例を以下に示す。システムは TCP/IP で接続された複数の UNIX ワークステーション上の X ウィンドウシステムで動作する。

[3]この点は実際の運用においては特に重要である。共有データの表現には ASN.1や SUN RPC の XDRのような標準型式を使用することができる。

[4]複製方式の利点は効率のみであるが、複製方式が必ずしも良い性能を示すとは限らないという報告もある。

Linda の拡張　CSCW アプリケーションを構築しやすくするため、Linda に以下のような若干の拡張を加えた。

1. ブロック / ノンブロック入出力

   7.3.1節で述べたように、Linda の出力プリミティブ**out** は常にブロックせず、入力プリミティブ**in**、**rd** は常にブロックする。ところがユーザインタフェースで使用する場合、入力の有無を調べるためのノンブロック入力があると便利であるし、ブロック出力が可能だとブロック入力と組みあわせてパイプライン処理ができるようになるので、入出力両方においてブロック / ノンブロックを指定できるように拡張を行なった[5]。

2. タプル出現順序の保存

   Linda では、マッチするタプルが複数存在した場合、どれが選択されるかは非決定的である。しかし非同期通信を行なうふたつのプロセスにおいて、ひとつのプロセスがタプル空間内に出力したタプルをもうひとつのプロセスが出現順に取得することができると便利である。このため、マッチするタプルが複数ある場合は一番最初に出現したタプルを使用するモードを追加してある[6]。

3. 同一タプル**rd** の制限

   タプル空間内のマッチするタプルを全て**rd** により処理しようとする場合、**rd** の前後でタプル空間の状態は変化しないので、何度**rd** を発行しても同じタプルが選択されてしまう可能性がある。このような要求もよく発生するため、一度**rd** により選択されたタプルは再度マッチしないようにするモードを設けた。

共有空間通信サーバ　共有空間は UNIX のひとつのサーバプロセスとして実装している。各アプリケーションプログラムは TCP/IP でサーバプロセスと通信することにより拡張 Linda の各プリミティブを実行する。

複数ユーザ図形エディタ

複数のユーザが同時に操作可能な図形エディタを共有空間通信サーバを用いて実装した例を図 7.12に示す。

---

[5]拡張された Linda ではノンブロック**in** である**inp**、ノンブロック**out** である**outp** が仕様に追加されている。

[6]通常の Linda でもタプルのパラメータに番号をふっておくことでタプルに順序付けすることは可能であるが、順序付けが必要になる場合は多いためこのような機能を追加した。Linda の拡張モデルである Cellula[122] においてもこのような機能が追加されている。

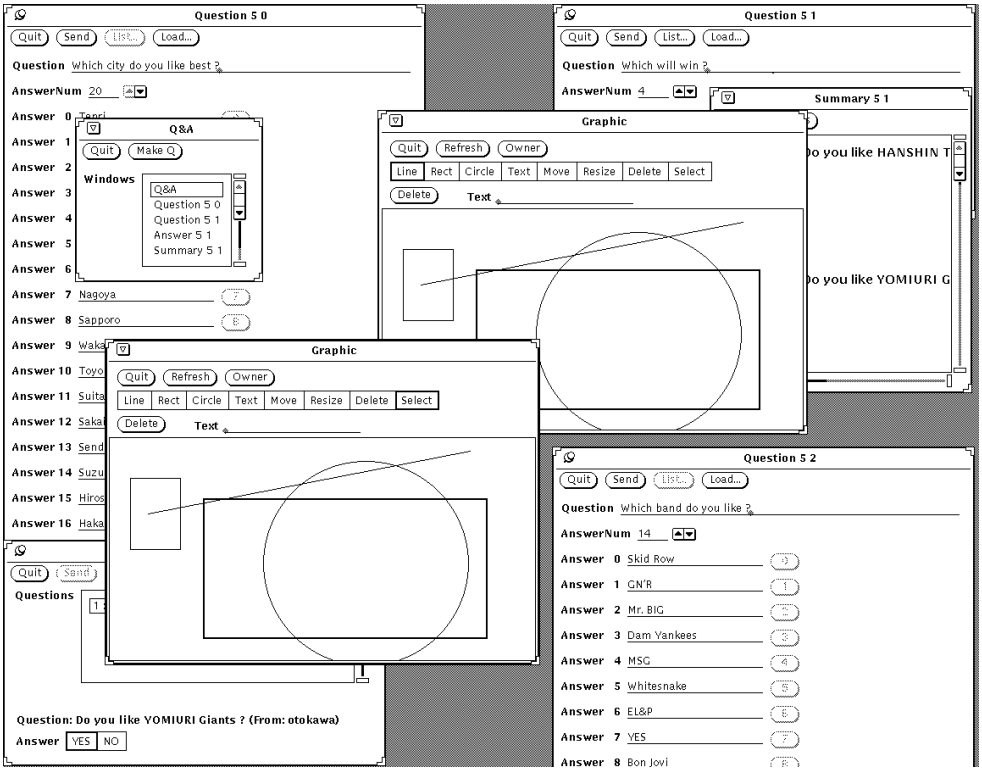


図 7.12: 図形エディタの実行例

全ての図形データはサーバ上のタプル空間内に存在しどのプロセスからでも参照可能である。ユーザはシングルユーザ用のエディタと同様に図形の編集を開始するが、現在の状態はすべて共有空間内に保持されているため、後で別のユーザが編集作業に加わっても古いユーザと全く同じ画面から開始できる。このように同期型 / 非同期型両方の性質をもつシステムが同じ枠組で実現できる。

図形の追加の際のタプルのやりとりを図 7.13に示す。

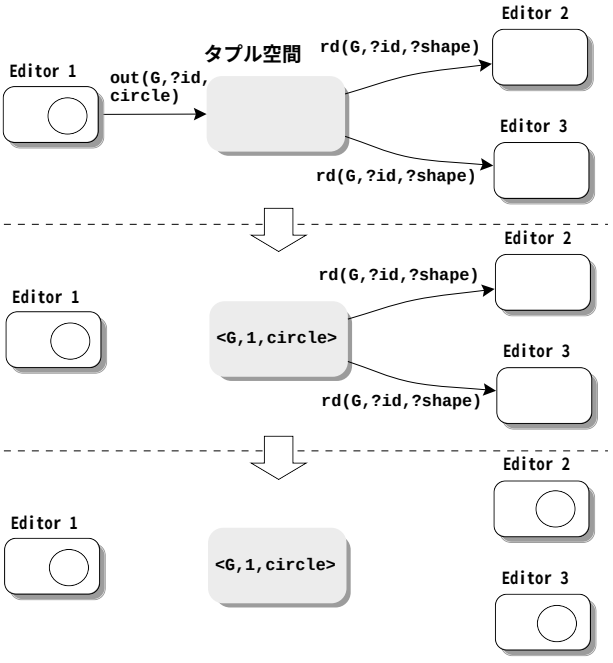


図 7.13: 図形の追加時のタプルのやりとり

ユーザはエディタの “Circle” ボタンを選択し、画面上でマウスをドラッグすることにより円を描く。描画が終了するとそのプロセスは新しい円のデータを示すタプルを**out** により共有空間内に出力する。別のエディタプロセスは**rd** によりこれを検出し、自分の画面にも同じ円を描く。

また図形の選択の際のタプルのやりとりを図 7.14に示す。

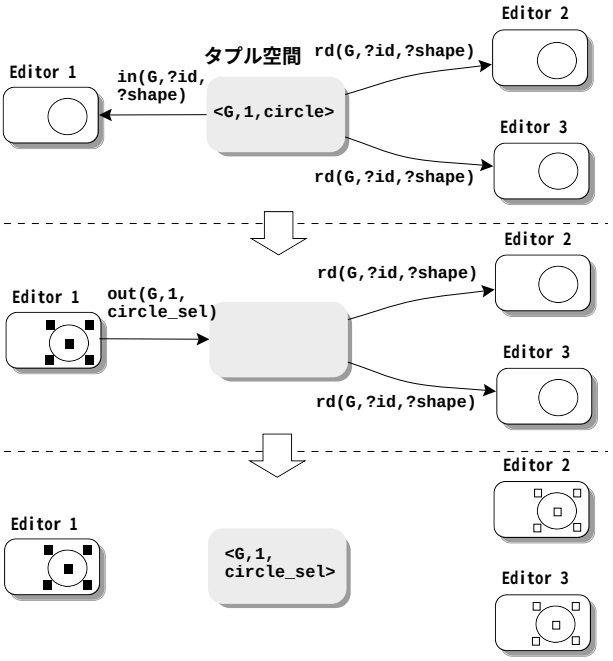


図 7.14: 図形の選択時におけるタプルの交換

ユーザはエディタの “Select” ボタンを選択してから図形上でマウスをクリックすることにより図形を選択する。このとき**in** により共有空間内からその図形を示すタプルを除去し、それが成功するとそのタプルのかわりにその図形が現在選択されていることを示すタプルを出力する。他のエディタプロセスは**rd** によりこれを検出するとその図形が他のエディタにより選択されていることを表示する。**in** は排他的操作であるため複数のプロセスが同時に同じ図形を選択することはない。

アンケート調査システム

図 7.12内には共有空間通信サーバを用いたアンケート調査システムも同時に示されている。アンケート調査システムとは、各ユーザが選択したデータが集計されて表示されるというものである。アンケート調査システムにおけるタプルの流れは図 7.15のようになっている。

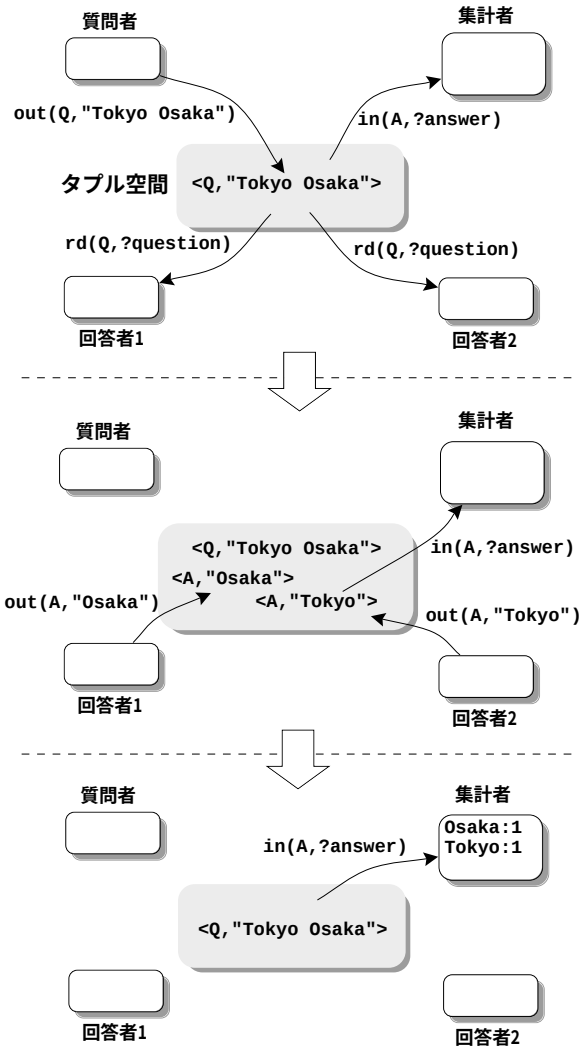


図 7.15: アンケート調査システムにおけるタプルの交換

アンケート調査システムで使っている共有空間通信サーバは図形エディタで使っているものと同じであり、アプリケーションの数が増えても共有空間通信はひとつだけでよい。

7.4.3 共有空間通信を使用したオーサリングシステム

マルチメディアプレゼンテーションを行なうためのアニメーションなどを生成するために、MacroMedia 社の Directorなど各種のオーサリングシステムが広く使用されている。また各種のデータ視覚化ツールもプレゼンテーションに広く使用されている。これらのシステムを使うと比較的簡単にプレゼンテーション用のアニメーションなどを作成することができるが、視覚化手法はあらかじめ用意されたものに限られているうえに、ランタイムにおけるユーザとのインタラクション機能は非常に限られたものしか用意されていない。一方、各種のユーザインタフェース作成ツールはいろいろな高度なインタラクション手法をサポートしているものの、スライダやボタンのような比較的静的で単純なインタラクションツールしか用意されておらず、複雑な視覚化や動画などのプレゼンテーションが可能になっているものは少ない。

これらのツールはそれぞれが得意とする分野が限られているため、たとえば「3 次元エディタのアニメーションヘルプシステム」のように視覚化・動画・インタラクション手法の全てが必要となる領域ではこれらの従来のツールでは不充分である。

オーサリングシステムに共有空間通信を使用する利点

7.3.2節においてLindaによる共有空間通信がユーザインタフェースツールの構築に有用であることを述べたが、共有空間通信は以下のような点においてオーサリングシステムの構築にも有用である。

能動的アニメーションオブジェクト アニメーションを表現しようとする場合、タプル空間を介して通信するプロセスとして個々のオブジェクトを実現することが可能である。このような能動的オブジェクトでは、自分がどのように表示されるべきか、どのように外部イベントに反応すべきかなどについて自分自身で対応させることができるので、ひとつの制御部が全オブジェクトの動きを制御しなければならないようなシステムに比べアニメーションの構築が容易にできる。

柔軟な制御構造 Directorなど多くのオーサリングシステムでは、シナリオは絶対時間または相対時間に基づいて進行するようになっている。一方、 Macintoshの HyperCardのように、ユーザのインタラクションが主導でシナリオが進行するようになっているオーサリングシステムもある。両者ともに他者のアプローチもある程度はとり入れている。例えばDirectorでもLingoという記述言語を使用してユーザインタラクションをある程度記述できるようになっているし、 HyperCardの中から動画を再生することもできるようになっている。しかし、 Lingoでは複雑なインタラクションを記述することはできないし、 HyperCardは特に動画再生やアニメーションに向いているわけではない。

Lindaを使用し、時刻やイベントを全てタプルとして表現することにすれば両者のアプローチは融合することができる。能動的オブジェクトは現在時間を**`rd`**によってチェックし、かつ同時に**`rd`**または**`in`**によりイベントに反応することができる。

他システムとの融合 Linda上に構築されたオーサリングシステムは他のシステムやツールと通信を行なうことができるので、各種の視覚化システム、デモシステム、ヘルプシステムなどと協調させることができる。これに対しDirectorやHyperCardのようなシステムは単体で独立したシステムなので、他のシステムと通信させて使用することは簡単にはできない。あらゆる機能を持つ一枚板のシステムを構築することは可能でもないし効率も悪いので、各種のツールを用意しておいて共有空間通信で通信することにより協調的に使用することが得策と思われる。

各種言語の併用 7.3.3節において、異なる言語を融合して使用することの利点について述べたが、オーサリングシステムの場合も同じように、各オブジェクト、時間管理、ユーザインタフェースなどに対し異なる言語を用いることが有効である。

アニメーション作成例

ここではLindaプリミティブを使用して作成したアプリケーションの例を示す。

時間ベースの単純なオーサリングシステム 時間ベースのオーサリングシステムでは少なくとも以下の要素が必要である。

時刻管理 現在時刻を管理する
シナリオ管理 与えられたシナリオを解釈し、何を表示するか決定する
対話管理 ユーザアクションを検出し、シナリオに影響を与える
表示管理 適切な絵や文字を表示する

これらの各要素はプロセスとして実現できる。例えば非常に単純な時間管理機構は以下のように実装することができる。

```
out('Time',time());       // 最初のタプルを出力
while(1){
     in('Time',?dummy);   // 古いタプルを除去
     out('Time',time()); // 新しい時間を示すタプルを出力
     sleep(1);
}
```

time() 関数により現在の時刻が得られる。この時間管理機構と協調動作するシナリオ管理は以下のように実現できる。

```
while(1){
     in('Time',?t);
     out('ScenarioTime',t % 10);
}
```

このシナリオ管理では、あらゆる動きが 10 秒毎に繰り返されることを指示している。このシナリオ管理を用いると例えば以下のような表示管理部を使うことができる。

```
while(1){
     rd('ScenarioTime',?t);
     show(t);
}
```

show() は時刻**t** における適当な画像を出力するための関数である。

対話管理部は一般のユーザインタフェースにおけるものと同様に構築することができる。

このように Linda プリミティブを用いることにより、オーサリングシステムの主要部を簡単に分離して実装することが可能である。各モジュールは他に影響を与えることなく簡単に別のものと取り替えることが可能である。

プログラムのデモシステム　タプル空間の動作をアニメーションとプログラムリストで図示するデモシステムのスナップショットを図 7.16に示す。

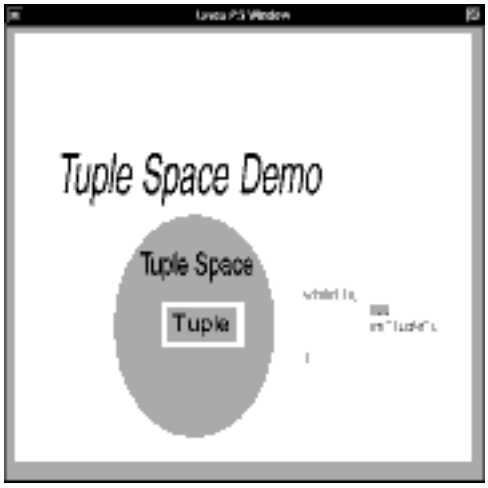


図 7.16: タプル空間通信のデモ (1)

右側の**while** ループが実際にこのデモ中で実行されており、実際に実行されている文に対応して灰色の矩形が表示される。図 7.16の状態では、まだ**in** が実行されていないためタプルがひとつタプル空間中に存在するが、実行が進むと図 7.17の状態になってタプルは消滅する。

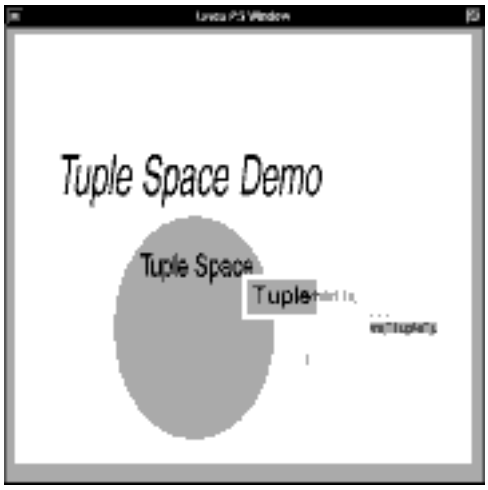


図 7.17: タプル空間通信のデモ (2) – **in** が実行されタプルが消滅

この例では、図の右側に示されたプログラムの他にいくつかのプロセスが動いている。表示プロセスは常にタプル空間の状態とプログラムの状態を監視しており、その状態を表示する。別のウィンドウにある「make tuple」ボタンにはひとつの入力プロセスが結合されており、ボタンが押される度にひとつタプルが生成されてその結果タプルを示す矩形が表示プロセスによって表示される。このように単純なプロセスをいくつか組みあわせることにより、プログラム実行のアニメーションによる視覚化が実現されている。

アニメーションヘルプシステム　アプリケーションが複雑になるにつれて高度なヘルプシステムが必要になる。ユーザフレンドリなヘルプシステムには以下のような条件が必要と考えられる。まず、どのようなタイミングでもユーザの質問を受け付けることが可能であるべきなので、ヘルプシステムはアプリケーションと独立したプロセスとして実装されるべきである。次に、ヘルプシステムはユーザの意図をなるべくうまくくみとる必要があるので、システムの現在の状態についてできるだけ多くの情報を得ることができなければならない。最後に、ヘルプシステムは最終的にはユーザに操作のやり方を例によって示すことが望ましい。このためにはアニメーションにより使い方を指示することが有効である。

タプル空間によりアプリケーションとコンテキストを共有するヘルププロセスを使用することにより上述のような要件は全て満たすことができる。ヘルププロセスは常にユーザ及びシステムの状況をタプル空間から知ることができるし、ユーザのかわりにタプルを送出したり前節のようにアニメーションを表示することにより、実際にアプリケーションの操作を行ないながら操作をアニメーションによりユーザに指示することができる。

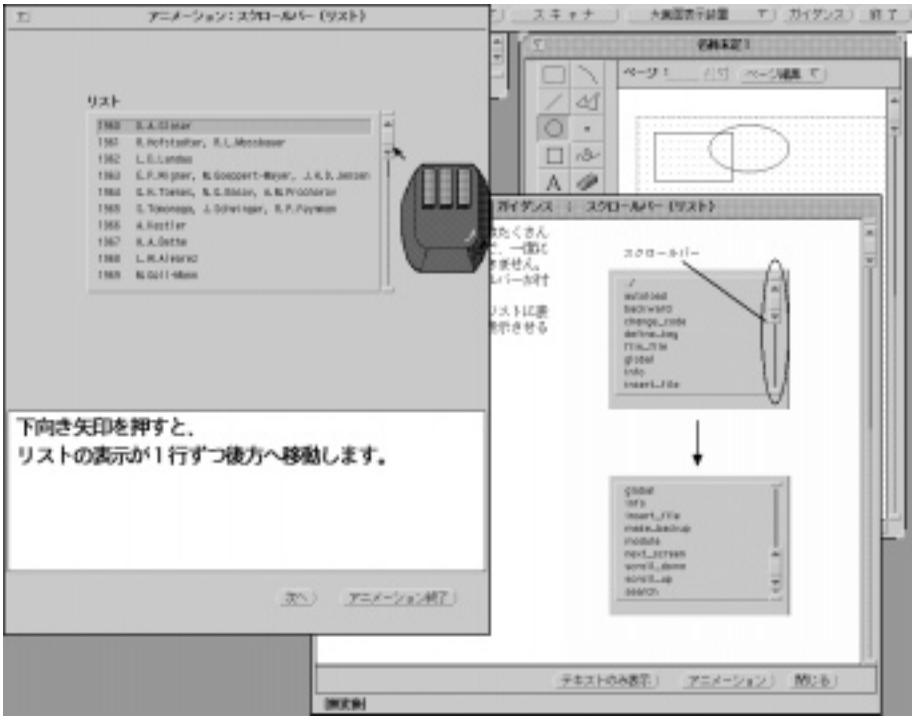


図 7.18: 図形エディタ上に実装されたアニメーションヘルプシステム

図形エディタ上に実装されたアニメーションヘルプシステムを図 7.18に示す。ユーザがシステムに操作の方法を質問すると、大きな「マウス」が出現してどの操作を声とアニメーションでユーザに教える。ヘルプシステムは独立した Linda プロセスとして動作しており、動作は特別なヘルプ記述言語で記述されている。このように共有空間通信を使用することにより簡単にアニメーションヘルプシステムを構築することができる。

対話的シミュレーションシステム　一般にシミュレーションシステムは、複雑な計算アルゴリズムと単純なインタラクションを持つオーサリングシステムと考えることが可能である。以下のようなプログラムで簡単なシミュレーションプログラムを作ることが可能であるが、これではシステムが**calculate()** を実行中はユーザはシステムに何も指示することができない。

```
while(1){
     calculate();          // シミュレーション計算を行なう
     check_user_input(); // ユーザの介入によりパラメタ修正、
                           // 実行 / 停止などを行なう
}
```

計算の実行中にもユーザがシステムを操作可能にするためには、ユーザインタフェース部は計算部と独立している必要がある。そのようにすれば、計算部はどのようなタイミングでもユーザの要求に従ってパラメタ値の変更などの処理を行なうことができる。

図 6.14の GALAPAGOS システムは NEXTSTEP のライブラリだけを用いて作成されているため、上記のようなプログラム構造になっており計算終了のタイミングでしかパラメタ変更や計算の停止 / 再開を指示できないが、遺伝的アルゴリズム計算部を独立した Linda プロセスとして実装することにより任意の時点においてパラメタを修正することが可能になる。

議論

以上のような例で示したように、独立した部品を結合させることにより複雑な対話的システムを構築することができる。アプリケーション部とインタフェース部の分離が重要であるのと同様に、ビジュアライザ、シナリオインタプリタや他の部品は独立して並列に実行されて、タプル空間を介してのみ通信することが望ましい。

実装

以上に紹介したシステムは全て UNIX 上に構築されている。タプル空間は Linda サーバとしてひとつのプロセスとして実装され、各プロセスは TCP/IP でサーバと通信を行なう。ひとつの UNIX プロセスはコルーチンとして動作する複数の Linda プロセスで構成されてもよい。 Linda プリミティブは C, C++, Perl[84]のライブラリとして実装されており、 Linda サーバは Perl で実装されている [105]。

7.5 予測 / 例示インタフェースの統合的アーキテクチャ

7.5.1 共有空間通信による予測例示インタフェースの実現

7.2.3節で述べたように、予測 / 例示インタフェースを実現するアーキテクチャには以下のような機能が必要である。

- ユーザの操作履歴の記録
- システムの状態遷移の記録
- システムの出力の記録
- 付帯情報の記録
- 各種記録情報の解析
- 予測結果の提示 / 実行
- 解析モジュールの交換

これらの要件は 7.2.2節で述べた各種の要件のサブセットと考えられる。普通の手続き型言語でこれらの要件を満たすことは難しいが、 Linda を用いた共有空間通信方式を用いればこれらは容易に実現することができる。変更される可能性のあるモジュールはひとつのプロセスとしておけば動的に交換することができるし、予測や推論のためのデータ収集や解析にも専用のプロセスを割り当ててアプリケーション本体と通信させることにすれば非同期的なデータ収集 / 解析が可能になる。

履歴に応じて動作を変えることのできる適応型インタフェースシステムでは上記の要件を満たすために専用のアーキテクチャを工夫しているものが多い。例えば Flexcelシステム [82] では、ユーザ操作を解析するために Lisp を使った推論システムが並列に動作しており、 Excelと推論システムの間で適宜通信を行なうことにより適応を行なわせている。 Flexcel のアーキテクチャでは推論システムが独立しているため高度な適応を行なわせることも可能であるが、 Excel 本体の機能は変更することができないので Excel が持つマクロ機能で可能になる適応動作しか指示することはできない。また UIDEシステム [78] では、適応やヘルプ機能を実現するためのアプリケーションモデルの構成法が決められており、これに基づいてアプリケーションを作成する限りある程度の適応動作が可能になっているが、ソフトウェア構成に厳しい制限があるため、既存の資源を活用したり、全く独立したモジュールを適応に使用することはできない。共有空間通信を使用する方式では、既存のアプリケーションをほとんどそのままの形で予測 / 適応モジュールと組み合わせて使用することが可能である。

7.5.2 予測 / 例示インタフェースシステムの統合的実現

本論文で述べた各種の予測 / 例示インタフェース手法を共有空間通信によって統合して扱うためのモジュール構成法を図 7.19に示す。

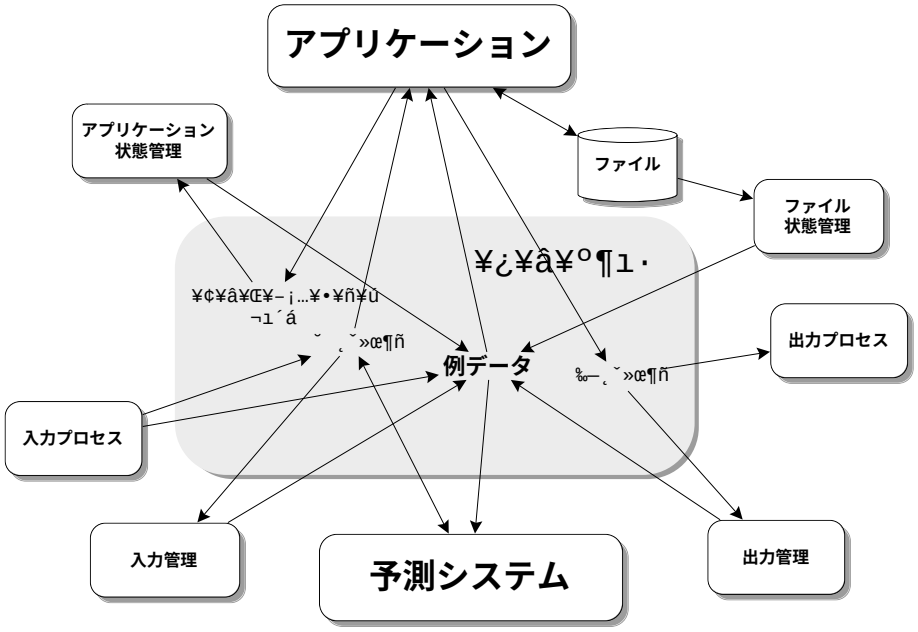


図 7.19: 共有空間通信で実現した統合的予測 / 例示インタフェースアーキテクチャ

アプリケーションは常にタプル空間内の入力字句、内部状態、出力字句を示すタプルを使いながら動作する。アプリケーションの入力には入力字句タプルを使い、出力には出力字句タプルを使用する。

予測 / 例示インタフェースシステム全体は、入力管理 / 出力管理 / アプリケーション状態管理 / ファイル状態管理の各モジュール及び手法毎に異なる予測システムから構成される。各モジュールは常にタプル空間を監視し、入力履歴、アプリケーション状態、出力履歴、ファイルの参照 / 更新履歴などの情報から例データを生成してタプル空間内に格納する。予測システムは例データを使用して各種の予測を行ない、その結果をアプリケーションへの入力としてタプル空間に返す。

4章〜 6章で述べた各種の予測 / 例示インタフェースをこのアーキテクチャで実現する手法を以下に述べる。

Dynamic Macro の実現

図 7.19のアーキテクチャで 4章の Dynamic Macro を実現した様子を図 7.20に示す。

```
#!/usr/local/bin/perl
require 'cbreak2.pl;                                    #
require 'linda.pl';                                     # Linda ライブラリ

&lindaopen;                                             # Linda サーバとの接続
&cbreak;                                                # 端末を CBREAK モードに設定

while(1){
        $s = getc;
        $c = ord($s);
        if($c == 0x14){                                 # 繰返しキーの場合
                $ks = '';
                for($i=$n-1;$i>=0;$i--){        # タプル空間から履歴を取得
                        &rd('KEYSTROKE',$i,'?a');
                        $ks .= $a;
                }
                # 繰返しの抽出
                $rep = '';
                if($ks =~ /^(.+)\1/){                   # 規則 1 の適用
                        $rep = $1;
                }
                elsif($ks =~ /^(.+)(.+)\1/){            # 規則 2 の適用
                        $rep = "$1$2";
                }
                if($rep){ # 繰返しが検出された場合は処理を繰り返す
                        @rep = split(//,$rep);
                        for($i=$#rep;$i>=0;$i--){
                                &func(substr($rep,$i,1)); # キー操作処理
                        }
                }
        }
        else {          # 通常キー
                &out('KEYSTROKE',$n++,$s);      # タプル空間にキーストローク情報を送付
                &func($s);                              # キー操作処理
        }
}

sub func { # キー操作処理関数
...
```

図 7.21: Perl と Linda サーバによる Dynamic Macro の実現

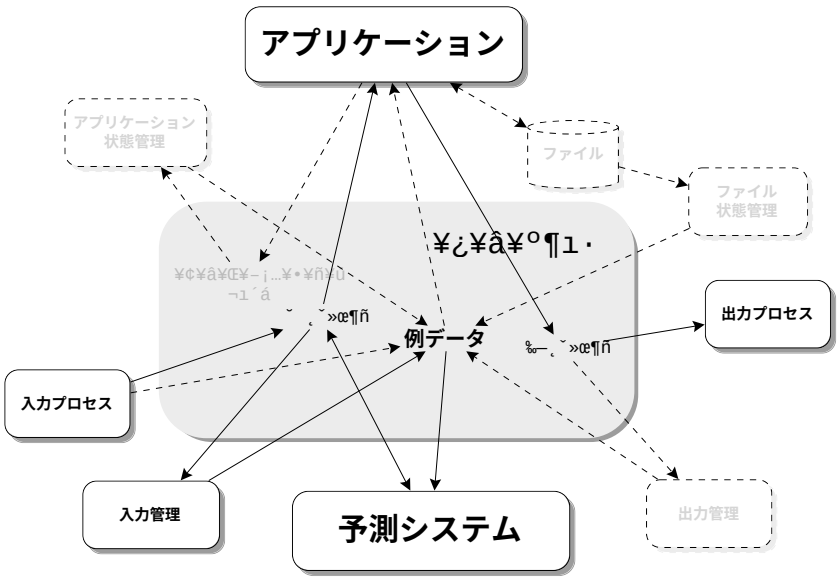


図 7.20: 共有空間通信による Dynamic Macro の実現

入力管理モジュールは入力を監視して操作履歴の保存を行ない、予測のための例データとして提供する。予測システムは、ユーザからの指示があったとき、例データ中の繰返しを抽出してアプ

リケーションへ返す。

Linda サーバを用いて Perl プログラムで Dynamic Macro を実現した例を図 7.21に示す。この例では、入力字句のリストをそのまま例データとして使用している。

Smart Make の実現

図 7.19のアーキテクチャで 5章の Smart Make を実現した様子を図 7.22に示す。

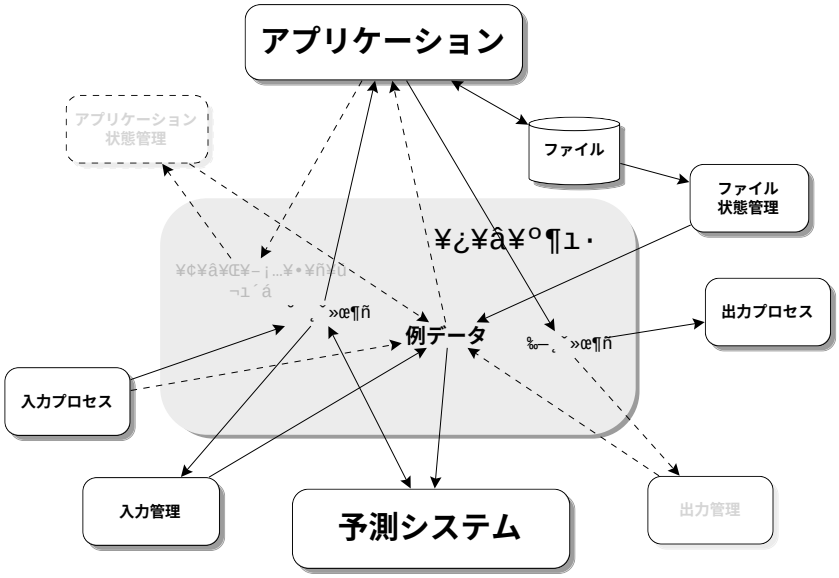


図 7.22: 共有空間通信による Smart Make の実現

ファイル状態管理モジュールはファイルの参照 / 更新を監視し、依存関係を利用した予測のための例データとして提供する。入力管理モジュールは、コマンド履歴を監視して例データとして提供する。予測システムは、ユーザからの指示があったとき、例データ中のファイルとコマンドの依存関係を抽出する。

図形配置評価関数獲得システムの実現

図 7.19のアーキテクチャにより 6章の図形配置の評価関数獲得システムを実現した様子を図 7.23に示す。

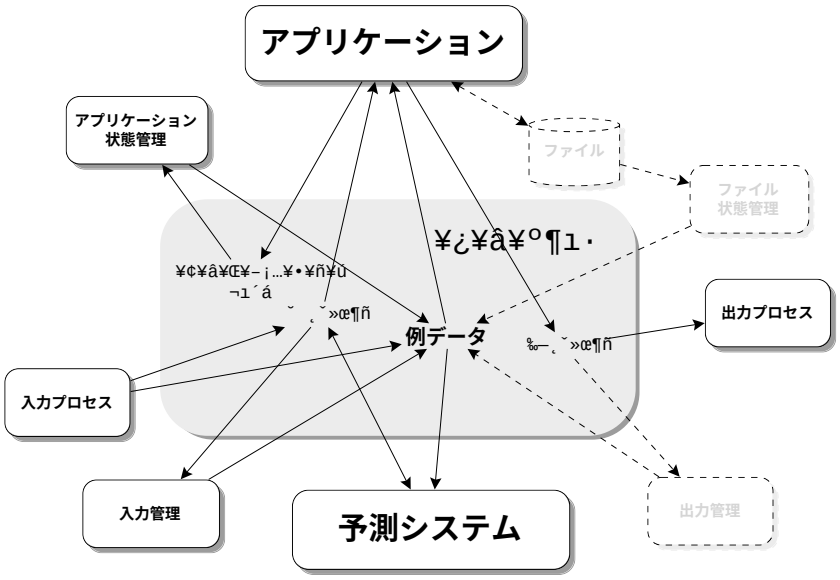


図 7.23: 図形配置評価関数獲得システムの実現

グラフ配置の例データは最初はユーザにより与えられ、評価関数は最初ランダムに与えられる。評価関数のよしあしはアプリケーション部で判断され、その結果が例として予測システムにフィードバックされる。予測システムはこれらの配置例、評価関数、評価結果から次世代の評価関数の候補を生成する。

Triggers の実現

図 7.19のアーキテクチャでは、アプリケーションの出力も例示データとして使用することができるため、画面出力を例示データとして使用する Triggers[73]のようなシステムも図 7.24のようにして実現することができる。

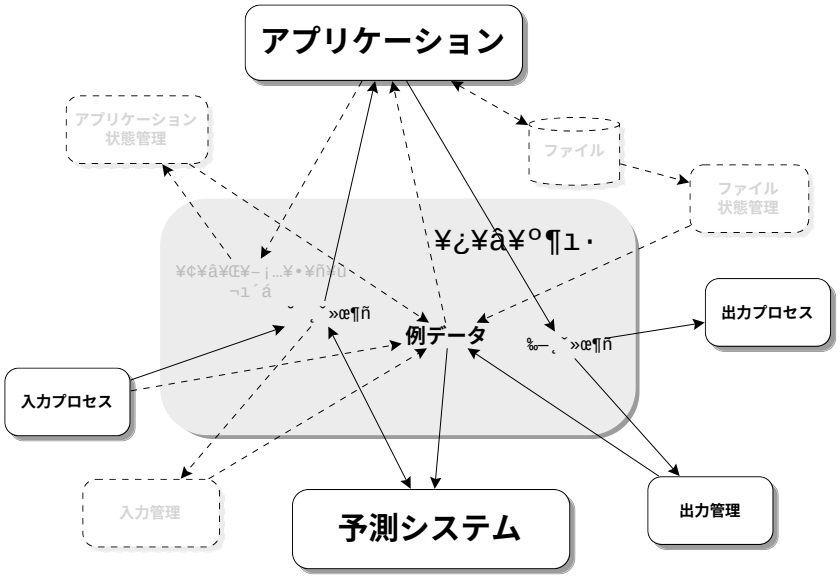


図 7.24: 共有空間通信による Triggers システムの実現

アプリケーションの出力は出力管理モジュールにより加工された後、例データとしてタプル空間に返される。予測システムは常にこれを監視し、指定したパタンが見つかった場合は対応するアクションを入力として返す。

7.5.3 統合アーキテクチャの評価

前節で述べたように、図 7.19のアーキテクチャにより、本論文で述べた各種の予測 / 例示インタフェース手法及びさらに広い範囲の予測 / 例示インタフェースシステムが実現できる。図 7.20〜図 7.24ではアプリケーションをひとつのプロセスのように表現しているが、共有空間通信を利用する場合は 7.3.2節で述べたようにアプリケーションの構築手法は柔軟であり拡張性も高い。予測システムは他のモジュールから独立しているため、予測手法に応じて交換することが容易である。

7.6 まとめ

本章では共有空間通信方式を基盤とするインタフェースアーキテクチャについて述べ、その予測 / 例示インタフェースへの適用について述べた。本章で述べたアーキテクチャは予測 / 例示インタフェースだけでなく CSCW システム、オーサリングシステム、適応型インタフェースなど広い範囲のアプリケーションの構築に適している。

第 8 章　予測 / 例示インタフェースの展望

8.1　はじめに

これまでの章では、 3 種類の予測 / 例示インタフェースシステム及びそれを構築するためのアーキテクチャについて述べてきた。本論文で述べた各種の予測 / 例示インタフェースシステムは、単純な繰返し的作業の効率化に関して非常に効果的であるが、計算機を使った知的な作業全般について効果があるわけではない。例えば、文章の編集作業に関しては 4章で述べた Dynamic Macro は非常に効果的に機能するが、新しい文章の作成や発想の支援には役にたたない。また、6章で述べた配置システムは比較的単純な図形のレイアウトには有用であるが、芸術的なレイアウトの作成ができるわけではない。現状の計算機インタフェースにおいては、単純な仕事さえ簡単に実行できないということがまず解決しなければならない問題であったため、単純な問題に的を絞って予測 / 例示インタフェースの適用を試みたわけである。

しかしこれは予測 / 例示インタフェース手法が単純な問題にしか適応できないということを示しているわけではない。予測 / 例示インタフェース手法は、高度に知的な作業に対しても適用可能な幅広い応用を持っている。本章では、さらに広い視点から見た予測 / 例示インタフェースの展望について述べる。

8.2　例にもとづくインタフェース

近年は大規模記憶装置やインターネットの普及により、あらゆる種類のデータの入手が容易になりつつある。本論文で述べた予測 / 例示インタフェースでは例データとしてユーザ自身の操作履歴を重視していたが、世の中に存在する各種の大量のデータを例として活用することを考えると、予測 / 例示インタフェースの適用範囲は非常に広がる。

芸術活動のように高度に知的な作業ですら従来の作品の存在抜きに考えることはできない。芸術作品は何も無いところから生まれるわけではなく、従来の作品や伝統の上に新たな創造を追加したものがその時代における優れた芸術作品となるからである。このとき、従来の作品を「例」と考えると、芸術の創造は例にもとづいた活動であるということができる。日常的な文章作成のように、芸術活動ほど創造的でなくてもよい場合は、世の中に存在する大量の前例を計算機でうまく利用することにより作業が大幅に効率化されることが考えられる。

このように既存のデータを活用する各種の手法を総称して**例にもとづくインタフェース**と呼ぶことにする。例にもとづくインタフェースのいくつかの例について以下に述べる。

8.2.1　例示による日本語入力システム

2.4節において、簡単な仮名漢字変換は予測 / 例示インタフェースの一種と考えられることについて述べた。仮名漢字変換では、単語や文章の完全な読みをシステムに与えた後で、その読みを持つ候補の中から必要なものを選択するのが普通であるが、完全な読みを与えるよりも前の段階からリアルタイムに単語予測を行なわせて、目的の単語が候補として出現した段階でそれを選

択することにより、通常の仮名漢字変換よりも効率的に単語を入力することができる可能性がある。特に、ペン計算機のように読みの入力に時間がかかる機械ではこのような手法が効果的と考えられる。

このような考えにもとづいた日本語入力システム POBoxを試作した [118]。以下に本手法を用いた文章入力例を示す。「以下に本手法を用いた...」という文を入力例として使用する。ペンの位置は矢印カーソルで示している。

図 8.1に初期画面を示す。ひらがなと若干の編集コマンドを含むソフトキーボードが表示されている。

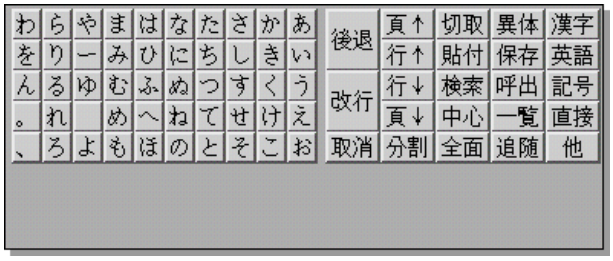


図 8.1: POBox 初期画面

“い” の上でペンを押すと、読み “い” が検索条件となり、 “い” で始まる単語のうち出現頻度が高いものから順に 10 個程度プルダウンメニューとして表示される。 (図 8.2)

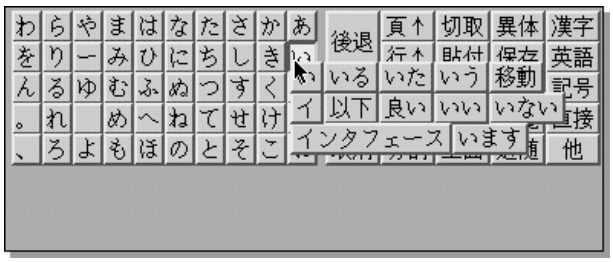


図 8.2: 読み “い” を指定

ここでペンを離すと、プルダウンメニューで表示されていた候補単語が図 8.3のように画面の下部に表示されるようになる。ペンを画面に触れてこれらの候補単語を選択することによりその単語を入力することができる。

図 8.3: 読み “い” を指定

図 8.3の状態からペンを画面に触れたまま移動させて “か” の上に持ってくると図 8.4の状態になる。この状態では検索条件が “いか” となっているため、 “いか” で始まる全単語が表示されている。ペンを別の文字上に移動させると、それにともなって検索条件と候補単語が動的に変化する。この状態からペンをメニュー上で移動して画面から離すことにより図 8.5のように候補中の “以下に” を選択することができる。

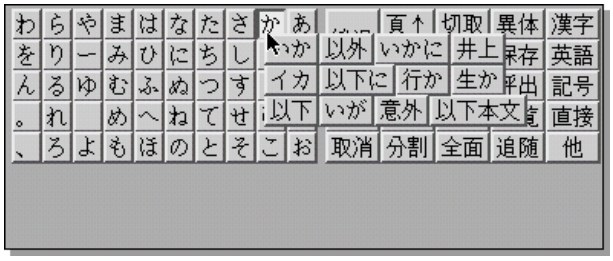


図 8.4: 読み “か” を指定

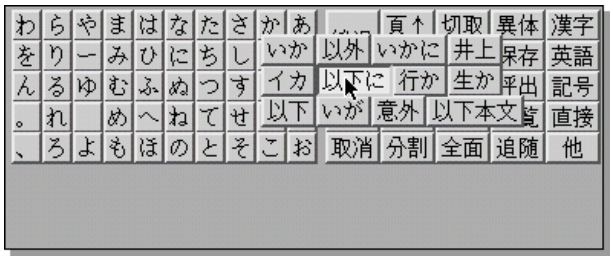


図 8.5: 候補から “以下に” を選択

同様の手順により読み “ほん” を指定すると図 8.6の状態になり、候補中の “本手法” を選択できるようになる。

図 8.6: 読み “ほん” を指定

“本手法” を選んだ時点で確定文字列は “以下に本手法” となっており、その直後に出現する頻度が高い単語が順に候補として画面下部に提示されている (図 8.7)。 “を” は “本手法” の後での出現頻度が高い単語のひとつなので、読みの条件を指定しなくてもリストされた候補の中からペンで直接選択することができる。
“を” の選択の結果、確定文字列は “以下に本手法を” となり、その後に出現する頻度の高い単語中から “用いた” を直接選択することができる。 (図 8.8)
ペンで画面に触れてから離すまでの操作を 1 操作と数えると、 “以下に本手法を用いた” という文字列を 6 操作で入力できたことがわかる。文字認識を使用する場合約 40 操作、ソフトキーボードと変換ボタンを使用する場合は最低 20 操作が必要となる。このように、自然言語辞書を例データとして活用することにより、ペン計算機でも高速に文章を入力することができるようになる。これは例にもとづくインタフェースの有望な応用例と考えられる。

8.2.2 例文からの文章作成システム

前節の文章作成システムでは単語 / 例文辞書を例として使用していたが、既存の文章そのものを例として活用することも考えられる。以前に作成したあらゆる文章を例データとして活用するシステムを図 8.9に示す。

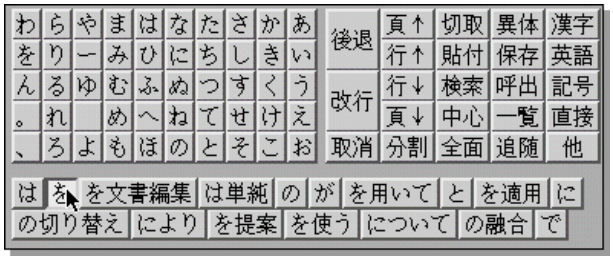


図 8.7: “を” を選択

図 8.8: “用いた” を選択

画面最下部において検索キーワード “**paper predic**” を入力することにより、キーワードマッチする文字列を含むファイル名のリストが画面上側に表示され、各ファイルの内容が画面下側に表示されている。 Emacs の**isearch** (incremental search) に類似した動的検索を行なうことにより、キーワードを変更するとすぐにそれにマッチするファイルがリストされるようになっている。このように、以前作成したファイルや例文を簡単な手間で検索することができるようになっていれば、それらを編集することにより似た文書を簡単に作成することができる。

このような操作は一般に予測 / 例示インタフェースと呼ばれることは少ないが、前節で述べた入力システムの拡張であり、例にもとづくインタフェースの有望な例のひとつと考えることができる。

8.3 移動環境におけるコンテクストの利用

予測 / 例示インタフェースシステムでは一般に現在のコンテクストを活用することが重要であるが、計算機内のコンテクストだけでなくユーザの置かれたコンテクストをも例として活用することが有用であると考えられる。

計算機の小型化 / 低価格化及び無線ネットワークの整備により常に身のまわりに計算機が存在して生活のあらゆる局面で計算機を利用する世界が目前となっており、このような状況で計算機を活用するための “実世界指向インタフェース”[92][123] が注目を集めている。 従来の計算機は部屋の中の机の前で使うことがほぼ前提とされていたため、ユーザの位置や状況を考慮する意味はあまりなかったが、移動環境におけるの実世界指向インタフェースでは、ユーザの現在位置や置かれた状況などのコンテクストが動的に変化するという大きな特徴があるため、それらを計算機で利用することが有用であると考えられる。

ユーザの位置 / 時間 / 状況などのコンテクストは非常に多くの組みあわせがあるので、すべての可能性を網羅してあらかじめプログラムしておくことはむずかしいが、例を活用することにより、状況に応じた適切な動作を予測することができるようになる可能性がある。例えば夜の駅で携帯端末で時刻表を調べたとことがあるとき、別の日の夜に携帯端末を使おうとすると、前の例から判断して自動的に時刻表を表示することができるかもしれない。通常のプログラミング言語を用いてこのような機能を実現しようとすると、センサから得られた緯度 / 経度情報、電界

図 8.9: 例文の検索

強度、時刻などの数値を判断し、条件を満たした場合に時刻表を起動するようなプログラムを書く必要があるかもしれない。しかし端末が事例にもとづいて適切な動作をするようになっていれば、一度夜の駅で時刻表を使ったことがあるという例情報にもとづいて、それに近い別の状況においても時刻表を表示することが可能になるであろう。このように、移動環境においては、ユーザの操作履歴だけでなく、位置 / 時間 / 周囲状況などのコンテクストも例として保存して活用することが有用であると考えられる。

8.4 適応型インタフェースとの融合

計算機を使いやすくするための別のアプローチとして、ユーザの特性を何らかの方法で検出してそれに応じて挙動を変える適応型 (adaptive) インタフェース [75] が研究されている。

真の適応を行なうためにはユーザの特性の深い部分をシステムが知る必要があるが、限られたバンド幅のインタラクションから充分な情報を得ることはむずかしいため、深いレベルまでユーザに適応可能なシステムは数少ないのが現状である一方、表層的な情報を利用するだけでも効果的な適応を行なわせることが可能である。例えば SKKなどの仮名漢字変換システムでは、直前に確定した単語が次回は第一候補として提示されるという単純な適応により変換効率が向上しているし、プルダウンメニュー中の項目のうちよく使われるものを他の項目と分離してメニューの最上部に表示されるようにした SplitMenu[76] や、階層的電話帳メニューにおいてよく選択さ

れる名前がメニューの階層の上の方に出現するようにしたシステム [22] において、通常のメニュー構成をとるのに比べ操作効率が上がったことが報告されている。

これらのシステムに共通しているのは、検索や予測を含むインタフェースにおいては単純な適応手段が効果的に働くということである [114]。各種の予測 / 例示インタフェース手法を適応的に使用することにより、より効果的な予測インタフェースを構築できる可能性がある。

8.5 まとめ

予測 / 例示インタフェースシステムは、非定型的な繰返し作業を効率化することに有用であるだけでなく、全く新しいインタフェースの構築や、新しい形態の計算機活用にも応用できる可能性が開けており、さらに広い応用をめざした研究が必要と考えられる。

第 9 章　結言

本論文では、予測 / 例示インタフェースシステムの必要性及び成功の条件についての考察を行ない、その考えに基づいた新しい予測 / 例示インタフェースシステムを提案し、実際のシステム構築による実証を行なった。

第 2章では現在までに提案されている各種の予測 / 例示インタフェースについて述べた。数多くの予測 / 例示インタフェース手法が提案されているにもかかわらず実用的に使われているものが少ない理由について第 3章で考察し、従来の予測 / 例示インタフェースの問題点を克服する新しい予測 / 例示インタフェースシステムでは、信頼性の高い単純な手法によりユーザの操作履歴などからユーザの暗黙的な意図をプログラムとして自動的に抽出し、その結果を単純な操作で再利用できることが必要であることを示した。

このようなシステムの構築が可能であることを示す実例として 3 種類の予測 / 例示インタフェースシステムを提案した。第 4章では、テキストエディタの操作履歴中から繰り返し操作を自動抽出して再利用することのできるシステム “Dynamic Macro” について述べた。 Dynamic Macro は動作原理が単純であるにもかかわらず、広い範囲の編集操作の繰り返し操作に適用可能であり、キーボードマクロなどの既存の機能よりも簡単な操作で使うことができる。第 5章では、操作履歴情報から操作間の依存関係を自動抽出し再利用を可能にするシステム “Smart Make” 及びその関連システムについて述べ、ユーザの意図を暗黙的に依存関係として操作履歴から抽出することのできる場合には、そのような依存関係の自動抽出及び再利用により操作の手間を大きく減らすことができることを示した。また第 6章では、遺伝的プログラミングの手法を用いることによりユーザの好みをプログラムの形として抽出し後で利用する手法について述べ、以前の配置例をもとにして図形の配置におけるユーザ個有の評価関数を自動抽出し、確率的配置手法によりその評価関数を別の例に適用可能であることを示した。これら 3 種類のシステムは全て、単純な手法によりユーザの暗黙的な意図をプログラムとして自動的に抽出し再利用ができるようになっており、実用的な予測 / 例示インタフェースシステムを構築可能であることが示された。ユーザの意図は、 Dynamic Macro では繰返しプログラムとして表現され、 Smart Make では依存関係にもとづく規則の集合として表現され、図形配置システムでは評価関数として表現される。

第 7章では、予測 / 例示インタフェースシステムを実装するための、共有空間通信に基づくユーザインタフェースアーキテクチャについて述べた。本アーキテクチャは予測 / 例示インタフェースシステムの構築に有効であるだけでなく、単独のアプリケーションから CSCW アプリケーションまで幅広い用途に有効であることを示した。

第 8章では、予測 / 例示インタフェース手法の新しいインタフェースへの応用について述べ、今後の展望について述べた。

本論文ではまずあらゆる有効な予測 / 例示インタフェースに必要な条件を明らかにし、新しい考え方にもとづいた 3 種類の予測 / 例示インタフェースを示すことによりそれを実証した。これらの各手法はそれぞれが新しく有益なものであり、また本論文で示した条件はさらに異なる予測 / 例示インタフェースを構築するときの基本要件の確認にも使用することができる。

参考文献

[1] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, Vol. 15, No. 3, pp. 237–269, 1983.

[2] AT&T. *Augmented Version of Make*, UNIX System V - release 2.0 support tools guide edition, April 1984.

[3] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pp. 14–21, Hillsdale, NJ, July 1987. Lawrence Erlbaum Associates, Publishers.

[4] D. Baldwin. Consul: a parallel constraint language. *IEEE Software*, Vol. 6, No. 4, pp. 62–69, July 1989.

[5] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[6] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, pp. 353–387, July 1989.

[7] Edwin Bos. Some virtues and limitations of action inferring interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pp. 79–88. ACM Press, November 1992.

[8] D. Browne, P. Totterdell, and M. Norman, editors. *Adaptive User Interfaces*. Academic Press, London, 1990.

[9] Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *Proceedings of SIGGRAPH*, Vol. 19, No. 3, pp. 199–204, July 1985.

[10] Nicholas Carriero and David Galernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, Vol. 21, No. 3, pp. 323–357, September 1989.

[11] Nicholas Carriero and David Gelernter. *How To Write Parallel Programs*. The MIT Press, Cambridge, MA, 1990.

[12] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. Mmconf: An infrastructure for building shared applications. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, October 7-10 1990. ACM Press.

[13] Allen Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 33–39. Addison-Wesley, April 1991. also in [14].

[14] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. The MIT Press, Cambridge, MA 02142, 1993.

[15] Allen Cypher and David Canfield Smith. KIDSIM: End user programming of simulations. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, pp. 27–34. Addison-Wesley, May 1995.

[16] John J. Darragh, Ian H. Witten, and Mark L. James. The Reactive Keyboard: A predictive typing aid. *IEEE Computer*, Vol. 23, No. 11, pp. 41–49, November 1990.

[17] Richard Dawkins. *The Blind Watchmaker*. W. W. Norton & Company, 1985.

[18] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, Vol. 13, No. 4, pp. 424–437, 1990.

[19] Emden R. Gansner and John H. Reppy. *A Foundation for User Interface Construction*, chapter 14, pp. 239–260. In Myers [63], 1992.

[20] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, pp. 80–112, January 1985.

[21] Saul Greenberg. *The Computer User as Toolsmith*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, March 1993.

[22] Saul Greenberg and Ian H. Witten. Adaptive personalized interfaces - a question of viability. *Behaviour and Information Technology*, Vol. 4, No. 1, pp. 31–35, 1984.

[23] John J. Grefenstette and Nicol N. Schraudolph. *A User's Guide to GENESIS 1.1ucsd*. Computer Science & Engineering Department, University of California, San Diego, La Jolla, CA, August 1990.

[24] L. J. Groves, Z. Michalewicz, P. V. Elia, and C. Z. Janikow. Genetic algorithms for drawing directed graphs. In Z. W. Ras, M. Zemankova, and M. L. Emrich, editors, *Methodologies for Intelligent Systems 5, Proceedings of the Fifth International Symposium*, pp. 268–276. North-Holland, October 1990.

[25] Daniel. C. Halbert. Programming by example. Technical Report OSD-T8402, Xerox Office Systems Division, December 1984.

[26] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, Vol. 8, pp. 231–274, 1987.

[27] R. D. Hill. *A 2-D Graphics System for Multi-User Interactive Graphics Base on Objects and Constraints*. Springer-Verlag, Berlin, 1990.

[28] Ralph D. Hill. Supporting concurrency, communication, and syncronization in human-computer interaction: The sassafras uims. *ACM Transactions on Graphics*, pp. 179–210, July 1986.

[29] Ralph D. Hill. Some important features and issues in user interface management systems. *Proceedings of SIGGRAPH*, Vol. 21, No. 2, pp. 116–120, April 1987.

[30] C. A. R. Hoare. Communicating sequential process. *Communications of the ACM*, Vol. 21, pp. 666–677, 1978.

[31] Scott E. Hudson. Graphical specification of flexible user interface displays. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'89)*, pp. 105–114. ACM Press, November 1989.

[32] Scott E. Hudson and Chen-Ning Hsi. A synergistic approach to specifying simple number independent layouts by example. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems (CHI'93)*, pp. 285–292. Addison-Wesley, April 1993.

[33] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, Vol. 10, No. 1, pp. 1–39, January 1991.

[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, No. 220, pp. 671–680, 1983.

[35] Corey Kosak, Joe Marks, and Stuart Shieber. New approaches to automating network-diagram layout. Technical Report TR-22-91, Center for Research in Computing Technology, Harvard University, Cambridge, MA 02138, 1991.

[36] Corey Kosak, Joe Marks, and Stuart Shieber. A parallel genetic algorithm for network-diagram layout. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 458–465, UCSD, California, August 1991. Morgan Kaufmann Publishers.

[37] John R. Koza. *Genetic Programming*. The MIT Press, Cambridge, MA, 1992.

[38] Robyn Kozierok and Pattie Maes. A learning interface agent for scheduling meetings. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, pp. 81–88. ACM Press, January 1993.

[39] David Kurlander. *Chimera: Example-Based Graphical Editing*, chapter 12, pp. 270–290. In Cypher [14], May 1993.

[40] David Kurlander and Steven Feiner. Editable graphical histries: The video. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 451–452. Addison-Wesley, April 1991.

[41] David Kurlander and Steven Feiner. A history-based macro by example system. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pp. 99–106. ACM Press, November 1992.

[42] Henry Lieberman. *Mondrian: A Teachable Graphical Editor*, chapter 16, pp. 340–358. In Cypher [14], May 1993.

[43] Jonas Löwgren. An architecture for expert systems user interface design and management. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'89)*, pp. 43–52. ACM Press, November 1989.

[44] Pattie Maes. Learning interface agents. In *Proceedings of the 1994 Friend21 International Symposium on Next Generation Human Interface*, February 1994.

[45] A. Markus. Experiments with genetic algorithms for displaying graphs. In *Proceedings of 1991 IEEE Workshop on Visual Languages (VLWS'91)*, pp. 62–67, Kobe, Japan, October 1991. IEEE Computer Society, IEEE Computer Society Press.

[46] Toshiyuki Masui. User interface construction based on parallel and sequential execution specification. *IEICE Transactions*, Vol. E 74, No. 10, pp. 3168–3179, October 1991.

[47] Toshiyuki Masui. User interface specifacation based on parallel and sequential execution specification. In *USENIX'91 Conference Proceedings*, pp. 117–125. USENIX' Conference Proceedings, January 1991.

[48] Toshiyuki Masui. Graphic object layout with interactive genetic algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pp. 74–80. IEEE Computer Society Press, September 1992.

[49] Toshiyuki Masui. *User Interface Programming with Cooperative Processes*, chapter 15, pp. 261–277. In Myers [63], 1992.

[50] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94)*, pp. 103–108. ACM Press, November 1994.

[51] Toshiyuki Masui and Ken Nakayama. Repeat and predict – two keys to efficient text editing. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pp. 118–123. Addison-Wesley, April 1994.

[52] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of SIGGRAPH'89*, Vol. 23, pp. 127–136, Boston, MA, July 1989.

[53] Micro Logic Corp., POB 70, Hackensack, NJ 07602. *KeyWatch*, 1990.

[54] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi, and Akinori Yonezawa. Interactive generation of graphical user interfaces by multiple visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94)*, pp. 85–94. ACM Press, November 1994.

[55] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi, Akinori Yonezawa, and Tomihisa Kamada. Declarative programming of graphical interfaces by visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pp. 107–116. ACM Press, November 1992.

[56] Dan H. Mo and Ian H. Witten. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology*, Vol. 11, No. 1, pp. 32–45, 1992. also in [14].

[57] Francesmary Modugno. *Extending End-User Programming in a Visual Shell With Programming by Demonstration and Graphical Language Techniques*. PhD thesis, Carnegie Mellon University, 1995.

[58] Francesmary Modugno and Brad A. Myers. A state-based visual language for a demonstrational visual shell. In *Proceedings of 1994 IEEE Symposium on Visual Languages (VL'94)*, 1994.

[59] Sven Moen. Drawing dynamic trees. *IEEE Software*, Vol. 7, No. 4, pp. 21–28, July 1990.

[60] Brad A. Myers. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, pp. 51–60, September 1987.

[61] Brad A. Myers. Text formatting by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 251–256. Addison-Wesley, April 1991.

[62] Brad A. Myers. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Computer*, Vol. 25, No. 8, pp. 61–73, August 1992.

[63] Brad A. Myers, editor. *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, MA, 1992.

[64] Brad A. Myers, Jade Goldstein, and Matthew A. Goldberg. Creating charts by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pp. 106–111. Addison-Wesley, April 1994.

[65] Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems (CHI'93)*, pp. 293–300. Addison-Wesley, April 1993.

[66] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'92)*, pp. 195–202. Addison-Wesley, May 1992.

[67] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating graphical interactive application objects by demonstration. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'89)*, pp. 95–104. ACM Press, November 1989.

[68] Robert P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 600–621, October 1985.

[69] Hajime Nonogaki and Hirotada Ueda. FRIEND21 project: A construction of 21st century human interface. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 407–414. Addison-Wesley, April 1991.

[70] Dan R. Olsen and Kirk Allan. Creating interactive techniques by symbolically solving geometric constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'90)*, pp. 102–107. ACM Press, October 1990.

[71] Rob Pike. Newsqueak: a language for communicating with mice. Computing Science Technical Report 143, AT&T Bell Laboratories, Murray Hill, New Jersey, 1989.

[72] Richard Potter. *Just-in-Time Programming*, chapter 27, pp. 513–526. In Cypher [14], May 1993.

[73] Richard Potter. *TRIGGERS: Guiding Automation with Pixels to Achieve Data Access*, chapter 17, pp. 361–380. In Cypher [14], May 1993.

[74] Steven F. Roth, John Kolojejchick, Joe Mattis, and Jade Goldstein. Interactive graphic design using automatic presentation knowledge. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pp. 112–117. Addison-Wesley, April 1994.

[75] M. Schneider-Hufschmidt, T. Kühme, and U. Malinowski, editors. *Adaptive User Interface – Principles and Practice*. North-Holland, Amsterdam, 1993.

[76] Andrew Sears and Ben Shneiderman. Split menus: Effectively using selection frequency to organize menus. *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 1, pp. 27–51, March 1994.

[77] David Canfield Smith, Allen Cypher, and Jim Spohrer. KIDSIM: Programming agents without a programming language. *Communications of the ACM*, Vol. 37, No. 7, pp. 55–67, July 1994.

[78] Piyawadee Sukaviriya and James D. Foley. Supporting adaptive interfaces in a knowledge-based user interface environment. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, pp. 107–113. ACM Press, January 1993.

[79] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA88)*, pp. 36–45, August 1988.

[80] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *Proceedings of the ACM SIGGRAPH and SIGCHI Symposium on User Interface Software and Technology (UIST'91)*, pp. 165–174. ACM Press, November 1991.

[81] Roberto Tamassia and Peter Eades. Algorithms for drawing graphs : an annotated bibliography. Technical Report CS-89-09, Brown University, Department of Computer Science, October 1989.

[82] Chritoph G. Thomas and Mete Krogsaeter. An adaptive environment for the user interface of Excel. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, pp. 123–130. ACM Press, January 1993.

[83] John Q. Walker. A node-positioning algorithm for general trees. *Software – Practice & Experience*, Vol. 20, No. 7, pp. 685–705, July 1990.

[84] Larry Wall and Randal L. Schwartz. *Programming Perl*. A Nutshell Handbook. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[85] Anthony I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, Vol. 11, No. 8, pp. 699–713, August 1985.

[86] Ian H. Witten and Dan H. Mo. *TELS: Learning Text Editing Tasks from Examples*, chapter 8, pp. 182–203. In Cypher [14], May 1993.

[87] David Wolber. Pavlov: Programming by stimulus-response demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'96)*, pp. 252–259. Addison-Wesley, April 1996.

[88] Bradley T. Vander Zanden. Constraint grammars - a new model for specifying graphical application. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'89)*, pp. 325–330. Addison-Wesley, May 1989.

[89] 畝見達夫. 人工生命と情報処理 (4) – 品種改良で CG 画像を作る模擬育種システム. *Computer Today*, Vol. 11, No. 6, pp. 76–82, November 1994.

[90] 北野宏明（編）. 遺伝的アルゴリズム. 産業図書, June 1993.

[91] 北野宏明（編）. 遺伝的アルゴリズム 2. 産業図書, May 1995.

[92] 小島啓二. ビジュアルインタフェースの研究動向と応用, 第 2.9 章, pp. 168–175. 平川, 安村 [100], February 1996.

[93] 佐藤雅彦. かな漢字変換システム SKK. *bit*, Vol. 23, No. 5, pp. 793–802, April 1991.

[94] 杉浦淳, 古関義幸. 例示プログラミングにおけるマクロ定義の簡略化. 田中二郎（編）, インタラクティブシステムとソフトウェア IV: 日本ソフトウェア科学会 WISS'96, pp. 101–110. 近代科学社, December 1996.

[95] 高橋伸, 松岡聡, 米澤明憲, 鎌田富久. 抽象的データと絵データ間の双方向変換に関する 一般的枠組. 第 7 回ヒューマンインタフェースシンポジウム 論文集, pp. 291–300. 計測自動制御学会: ヒューマン・インタフェース部会, October 1991.

[96] 塚本高史, 中島修, 山田雅高, 三宅芳雄. 個人適応型のユーザインターフェイス環境. 情報処理学会ヒューマンインタフェース研究会研究報告 94-HI-53, Vol. 94, No. 23, pp. 117–124, March 1994.

[97] 中山健, 宮本健司, 川合慧. コマンド履歴からの動的スクリプト生成. 竹内彰一（編）, インタラクティブシステムとソフトウェア II: 日本ソフトウェア科学会 WISS'94, pp. 155–164. 近代科学社, 1994.

[98] 萩谷昌己. ビジュアルプログラミングと自動プログラミング. コンピュータソフトウェア, Vol. 8, No. 2, pp. 27–39, March 1991.

[99] 浜田喬, 増井俊之, 茅野昌明. 自己増殖型コンパイラコンパイラの開発. 電子情報通信学会論文誌, Vol. J69-D, No. 1, pp. 50–60, January 1986.

[100] 平川正人, 安村通晃（編）. ビジュアルインタフェース – ポスト GUI を目指して. bit 別冊. 共立出版, February 1996.

[101] 増井俊之, 川原健児. オフィスワークステーションの開発 - ハードウエアとウィンドウ管理. 電気情報通信学会春期全国大会予稿集, No. 7, p. 62, 1987.

[102] 増井俊之. **`keisen.el`** プログラム, July 1991. Mule 配付パッケージに含まれる.

[103] 増井俊之. Perl 書法 – perl で書くプリティプリンタ. *Super ASCII*, Vol. 3, No. 4, pp. 146–155, April 1992.

[104] 増井俊之. 遺伝子アルゴリズムを用いた対話的図形配置. 情報処理学会ヒューマンインタフェース研究会研究報告 92-HI-41, Vol. 92, No. 15, pp. 41–48, March 1992.

[105] 増井俊之. Perl 書法. アスキー出版局, July 1993.

[106] 増井俊之. 遺伝的アルゴリズムの図形配置問題への適用, 第 6 章, pp. 165–183. 北野 [90], June 1993.

[107] 増井俊之, 花田恵太郎, 音川英之. 共有空間通信を利用したグループワークシステムの構築. 情報処理学会プログラミング – 言語・基礎・実践 – 研究会研究報告 92-PRG-10, Vol. 93, No. 11, pp. 49–56, January 1993.

[108] 増井俊之, 太和田誠. 操作の繰返しによるマクロの自動生成. 情報処理学会ヒューマンインタフェース研究会研究報告 93-HI-48, Vol. 93, pp. 65–72, May 1993.

[109] 増井俊之. インタフェースビルダの動向, 第 2.4 章, pp. 107–117. No. 05-R003. 日本情報処理開発協会, March 1994.

[110] 増井俊之. 効率の良いトライ / 状態遷移機械の構成方式. 情報処理学会 プログラミング – 言語・基礎・実践 – 研究会研究報告 94-PRG-15, Vol. 94, No. 7, pp. 73–80, January 1994.

[111] 増井俊之. 進化的学習機構を用いたグラフ配置制約の自動抽出. 竹内彰一（編）, インタラクティブシステムとソフトウェア II: 日本ソフトウェア科学会 WISS’94, pp. 195–204. 近代科学社, 1994.

[112] 増井俊之, 中山健. 操作の繰返しを用いた予測インタフェースの統合. コンピュータソフトウェア, Vol. 11, No. 6, pp. 484–492, November 1994.

[113] 増井俊之, 中山健. 操作の繰返しを用いた予測インタフェースの統合. 竹内彰一（編）, インタラクティブシステムとソフトウェア I: 日本ソフトウェア科学会 WISS’93, pp. 225–232. 近代科学社, 1994.

[114] 増井俊之. 適応 / 予測型テキスト編集システム. 竹内彰一（編）, インタラクティブシステムとソフトウェア II: 日本ソフトウェア科学会 WISS’94, pp. 145–154. 近代科学社, 1994.

[115] 増井俊之. グラフ配置評価関数の進化的獲得, 第 5 章, pp. 127–144. 北野 [91], May 1995.

[116] 増井俊之. 例を用いた視覚化. 「情報の視覚化とその関連技術」チュートリアル資料, pp. 71–87. 日本ソフトウェア科学会, July 1995.

[117] 増井俊之. GUI ベースのプログラミング, 第 2.2 章, pp. 45–64. 平川, 安村 [100], February 1996.

[118] 増井俊之. ペンを用いた高速文章入力手法. 田中二郎（編）, インタラクティブシステムとソフトウェア IV: 日本ソフトウェア科学会 WISS’96, pp. 51–60. 近代科学社, December 1996.

[119] 松岡聡. 例示による GUI プログラミング, 第 1.4 章, pp. 41–78. No. 06-R003. 日本情報処理開発協会, March 1995.

[120] 松岡聡, 宮下健. 例示による GUI プログラミング, 第 2.4 章, pp. 79–97. 平川, 安村 [100], February 1996.

[121] 宮下健, 松岡聡, 高橋伸, 米澤明憲. 複数の視覚的例による直接インターフェイスの対話的実現. 竹内彰一（編）, インタラクティブシステムとソフトウェア I: 日本ソフトウェア科学会 WISS’93, pp. 241–248. 近代科学社, 1994.

[122] 吉田紀彦, 楢崎修二. 場と一体化したプロセスの概念に基づく並列協調処理モデル Cellula. 情報処理学会論文誌, Vol. 31, No. 7, pp. 1071–1079, July 1990.

[123] 暦本純一, 長尾確. ポスト GUI: 今後の展望, 第 3 章, pp. 178–198. 平川, 安村 [100], February 1996.

[124] 暦本純一, 垂水浩幸. ユーザインタフェースとオブジェクト指向, 第 6 章, pp. 213–244. 岩波書店, 1993.

謝辞

本論文をまとめるにあたり、修士論文以来一環して御指導をいただいている東京大学浜田喬教授に適切な御序言と御指導をいただきました。また東京大学高木幹雄教授、田中英彦教授、原島博教授、坂内正夫教授、石塚満教授には、本論文の完成にあたり多大なる御教示、御助言をいただきました。心より感謝いたします。

本論文中の第 4章から第 7章までの研究は、 1986 年 3 月から 1995 年 12 月まで筆者が所属したシャープ株式会社技術本部における研究業務の一貫として行ないました。その期間における研究の機会を与えて下さったシャープ株式会社 河田亨取締役、大崎幹雄情報技術研究所長、小松純一ソフトウェア研究所長、千葉徹部長に感謝いたします。また、その間本研究に関する有益な討論、御助言をいただいた、シャープ株式会社 舟渡信彦氏、花田恵太郎氏、音川英之氏、今井明氏、市川雄二氏に感謝いたします。

第 4, 5章の研究について多大な議論をいただいた東京大学の中山健氏に感謝いたします。

第 8章の研究および本論文の作成は、筆者が 1996 年 1 月より所属する、株式会社ソニーコンピュータサイエンス研究所における研究業務の一貫として行ないました。その期間における研究の機会を与えて下さったソニーコンピュータサイエンス研究所 土井利忠所長、所真理雄副所長に感謝いたします。また、第 6章の研究の基礎となる遺伝的アルゴリズムについて御指導いただいた北野宏明氏及び、インタフェースシステム全般における幅広い議論をしていただいた暦本純一氏に感謝いたします。

著者による本論文に関連した発表

1 著書

[1] 増井俊之. Perl 書法. アスキー出版局, July 1993.

2 論文

[1] 浜田喬, 増井俊之, 茅野昌明. 自己増殖型コンパイラコンパイラの開発. 電子情報通信学会論文誌, Vol. J69-D, No. 1, pp. 50–60, January 1986.

[2] Toshiyuki Masui. User interface specifacation based on parallel and sequential execution specification. In *USENIX'91 Conference Proceedings*, pp. 117–125. USENIX' Conference Proceedings, January 1991.

[3] Toshiyuki Masui. User interface construction based on parallel and sequential execution specification. *IEICE Transactions*, Vol. E 74, No. 10, pp. 3168–3179, October 1991.

[4] Toshiyuki Masui. Graphic object layout with interactive genetic algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pp. 74–80. IEEE Computer Society Press, September 1992.

[5] Toshiyuki Masui. *User Interface Programming with Cooperative Processes*, chapter 15, pp. 261–277. In Brad A. Myers, editor, *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, MA, 1992.

[6] 増井俊之. 遺伝的アルゴリズムの図形配置問題への適用, 第 6 章, pp. 165–183. In 北野宏明（編）, 遺伝的アルゴリズム, 産業図書, June 1993.

[7] Toshiyuki Masui and Ken Nakayama. Repeat and predict – two keys to efficient text editing. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pp. 118–123. Addison-Wesley, April 1994.

[8] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94)*, pp. 103–108. ACM Press, November 1994.

[9] 増井俊之, 中山健. 操作の繰返しを用いた予測インタフェースの統合. コンピュータソフトウェア, Vol. 11, No. 6, pp. 484–492, November 1994.

[10] 増井俊之. グラフ配置評価関数の進化的獲得, 第 5 章, pp. 127–144. In 北野宏明（編）, 遺伝的アルゴリズム 2, 産業図書, May 1995.

3 解説 / チュートリアル

[1] 増井俊之. 例を用いた視覚化. 「情報の視覚化とその関連技術」チュートリアル資料, pp. 71–87. 日本ソフトウェア科学会, July 1995.

[2] 増井俊之. 例示 / 予測インタフェースの研究動向 コンピュータソフトウェア, Vol. 14, 1997, to appear.

4 予稿集他

[1] 増井俊之, 川原健児. オフィスワークステーションの開発 - ハードウエアとウィンドウ管理. 電気情報通信学会春期全国大会予稿集, No. 7, p. 62, 1987.

[2] 増井俊之. 遺伝子アルゴリズムを用いた対話的図形配置. 情報処理学会ヒューマンインタフェース研究会研究報告 92-HI-41, Vol. 92, No. 15, pp. 41–48, March 1992.

[3] 増井俊之, 花田恵太郎, 音川英之. 共有空間通信を利用したグループワークシステムの構築. 情報処理学会プログラミング – 言語・基礎・実践 – 研究会研究報告 92-PRG-10, Vol. 93, No. 11, pp. 49–56, January 1993.

[4] 増井俊之, 太和田誠. 操作の繰返しによるマクロの自動生成. 情報処理学会ヒューマンインタフェース研究会研究報告 93-HI-48, Vol. 93, pp. 65–72, May 1993.

[5] 増井俊之. 効率の良いトライ / 状態遷移機械の構成方式. 情報処理学会 プログラミング – 言語・基礎・実践 – 研究会研究報告 94-PRG-15, Vol. 94, No. 7, pp. 73–80, January 1994.

[6] 増井俊之, 中山健. 操作の繰返しを用いた予測インタフェースの統合. 竹内彰一（編）, インタラクティブシステムとソフトウェア I: 日本ソフトウェア科学会 WISS'93, pp. 225–232. 近代科学社, 1994.

[7] 増井俊之. 進化的学習機構を用いたグラフ配置制約の自動抽出. 竹内彰一（編）, インタラクティブシステムとソフトウェア II: 日本ソフトウェア科学会 WISS'94, pp. 195–204. 近代科学社, 1994.

[8] 増井俊之. インタフェースビルダの動向, 第 2.4 章, pp. 107–117. No. 05-R003. 日本情報処理開発協会, March 1994.

[9] 増井俊之. GUI ベースのプログラミング. 平川, 安村（編）, ビジュアルインタフェース – ポスト GUI を目指して, 第 2.2 章, pp. 45–64. 共立出版, February 1996.

[10] 増井俊之. ペンを用いた高速文章入力手法. 田中二郎（編）, インタラクティブシステムとソフトウェア IV: 日本ソフトウェア科学会 WISS'96, pp.51–60. 近代科学社, December 1996.

5 著者によるその他の論文

[1] Toshiyuki Masui. Keyword dictionary compression using efficient trie implementation. In *Proceedings of Data Compression Conference*, Salt Lake City, Utah, April 1991.

[2] Toshiyuki Masui, Kouichi Kashiwagi, and George R. Borden. Elastic graphical interfaces for precise data manipulation. In *CHI'95 Conference Companion*, pp. 143–144. Addison-Wesley, May 1995.

[3] Toshiyuki Masui, Mitsuru Minakuchi, George R. Borden IV, and Kouichi Kashiwagi. Multiple-view approach for smooth information retrieval. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'95)*, pp. 199–206. ACM Press, November 1995.

索引