プログラミング—言語・基礎・実践— 10−1
（1993. 1. 28）

抽象的データに対する直接操作インターフェイスの例による実現

宮下 健　松岡 聡　高橋 伸　米澤 明憲
東京大学 理学部 情報科学科

グラフィカルユーザーインターフェイス (GUI) の製作において，アプリケーション中のデータを視覚化することが必要だが，そのためのプログラミングは煩雑である．この問題を解決するため，本研究では視覚的例を用いた GUI のプログラミング環境を提案する．プロトタイプシステム TRIP3 は，アプリケーション中のデータとそれを視覚化したイメージとの間の双方向変換モデルに基づいており，プログラマーがアプリケーション中のデータとそれに対応した絵の具体例を直接操作を通して描きさえすれば，その例を汎化してデータと絵の間の双方向変換を実現する GUI を生成する．

Programming Graphical Interfaces by Visual Examples

Ken Miyashita　Satoshi Matsuoka　Shin Takahashi　Akinori Yonezawa
Dept. of Information Science, Faculty of Science, University of Tokyo

The “semantic gap” between the textual application data and its visual representation makes the construction of graphical user interfaces (GUI) more difficult than that of text-based interfaces. To solve the problem, we propose a programming environment based on the *programming by visual example (PBVE)* scheme, which allows the GUI designers to “program” visual interfaces for their applications by “drawing” the example visualization of application data with a direct manipulation interface. Our system, TRIP3, realizes this with (1) the *bi-directional translation model* between the application data and the pictorial data of the GUI, and (2) the ability to generate mapping rules for the translation by generalizing example application data and its corresponding example visualization.

1 始めに

Graphical user interface が広く使われるようになってきているが，その製作は従来のテキストベースのインターフェイスの製作に比べて，以下のような理由により複雑である．(1) 本質的にグラフィカルなものをテキストベースのプログラム言語で扱わなければならない．(2) アプリケーション中の抽象的なデータをどのように視覚化するか，プログラマーが管理しなければならない．

本研究の前身 TRIP2[10] では，(2) の問題を解決するために抽象的データとそれを視覚化した絵の間の双方向変換モデルを提案した．しかし，TRIP2 では双方向変換のためのルールをテキストベースの Prolog のプログラムとして表現したため，まだ (1) の問題点が残っていた．本研究では，この問題を解決するため直接操作と「視覚的例によるプログラミング Programming by Visual Example (PBVE)」を用いた GUI プログラミング環境を提案する．プロトタイプシステム TRIP3[8] は，GUI デザイナーが与えた抽象的データとそれに対する視覚化した絵の例を汎化して，抽象的データに対する直接操作インターフェイスを生成する．

2 双方向変換モデル — TRIP2

我々の以前の研究 TRIP[5], TRIP2[10] では，抽象的データとそれを視覚化した絵の間の双方向変換を一般的にモデル化するために，任意の抽象的データ，およびそれを視覚化した絵という一般的なデータ表現形式の層の間に，統一された 2 つのデータ表現形式の層を考えた (図 1)．ここで，Application's Data Representation (AR) は任意のアプリケーション中のデータ表現，今の例で言えば自然言語の文章である．また，Pictorial Representation (PR) は抽象的データを視覚化した絵そのものを指し，今の場合，上司 1 人と部下 3 人から成る組織図がそれに当たる．一方，Abstract Structure Representation (ASR) は抽象的データを「Prolog の節」という一定の形で表現し直したものであり，同様に Visual Structure Representation (VSR) は絵の構造 (この例で言えば，「部下は縦に 1 列に並べて描く」などの配置に関する規則) を Prolog の節で表したものである．

このように，ASR, VSR という統一された表現法を定めることにより，さまざまな形の抽象的データとそれを視覚化した絵の間の双方向変換に対するルールを統一された記法で記述することができる．例えば，先の図 1 中の組織図に対するルールは図 2 のようになる．ここで，大文字 (X, H など) は変数を表し，[H|L] はリスト構造を表す．つまり，リスト構造 [H|L] は任意個の要素を含むことができるので，このルールは一般的に「1 人の上司の下に任意数の部下がいる」という組織図を記述することができる．図 3 はこのルールを適用した時の情報の流れを示している．このルールを ASR から VSR への変換に適用すると，まず 1 行目により ASR データとルール中の変数，X, [H|L] との間で単一化が起こり，X は hogan, [H|L] は, [heinlein,clarke,herbert] に対応させられる．この変数と値の対応を用いて，2 から 4 行目までで VSR データが生成される．例えば，2 行目からは horizontal([hogan,heinlein],20) が生成される．逆に，VSR から ASR への変換を行なう時は，まず最初に 2 から 4 行目までで変数と値の対応が得られ，それを用いて ASR が生成される．このようにルール自体が宣言的であるため，双方向変換が簡潔に記述できる．

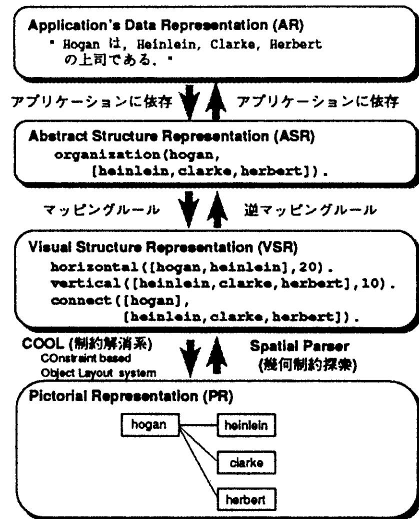

図 1: 双方向変換モデルの 4 つのデータタイプとその例

```
organization_rule :-
    % (1) X は上司, [H|L] は部下のリスト.
    asr(organization(X,[H|L])),
    % (2) X, H を間隔 20 で水平に並べる.
    vsr(horizontal([X,H],20)),
    % (3) 部下は, 垂直に並べる.
    vsr(vertical([H|L],10)),
    % (4) 上司と, 部下は線で結ぶ.
    vsr(connect([X],[H|L])).
```

図 2: 組織図に対するマッピングルール

ルール自体の簡潔さにもかかわらず十分な視覚的表現力を得るため，TRIP2 システムで使用している制約解消系 COOL は従来の GUI 用制約解消系に比べて，非常に高機能なものになっている．COOL の特徴は以下の通り：(1) 制約過多の場合も最小二乗法によって誤差を分散して近似解を得ることが出来る．この機能によって，horizontal([a,b]), vertical([a,b]) のような矛盾する幾何制約からも近似解を用いることによって絵を生成することが出来る．(2) ER ダイアグラムのようなトポロジーにのみ注目したグラフの配置を決定できる．(3) 木構造のような再帰構造を持った絵を視覚化で

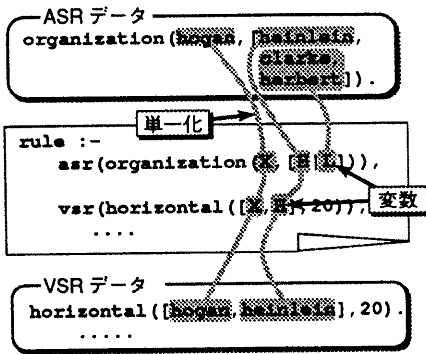

図 3: ルール適用の様子

きる．詳しくは，[5] を参照いただきたい．

このような一般的な枠組みを提供することで，任意の抽象的データとそれに対する視覚化との間の双方向変換を実現することができたが，プロトタイプシステム TRIP2 を使っていく中で，いくつかの根本的な問題点が明らかになってきた．

1. プログラマーは意図した絵を構成するように VSR を記述しなければならないが，ルールを記述している段階では実際の絵がまだ見えないので，細かいパラメータ (例えば，長方形の大きさなど) の設定が試行錯誤になりがちである．
2. プログラマーは，様々なデータに対して常に適切な幾何制約 (例えば，horizontal など) を過不足なく与えるような VSR を記述しなければならないが，これは分かりづらい．

3 視覚的例によるプログラミング

これらの問題点を解決するため，本研究では「視覚的例によるプログラミング Programming by Visual Example (PBVE)」を提案し，そのプロトタイプシステム TRIP3 を示す．PBVE とは，ここでは抽象的データとそれを視覚化した絵との具体例を汎化することによって一般的な双方向変換ルールを推論することを意味している．

図 1の例を用いて，PBVE によってルールを生成する過程を図 4に示す．左の 2 つの囲みが，それぞれデザイナーの描いた GUI の絵とそれに対応する ASR データの例である．人間のプログラマーがこれらの例から一般的な GUI を作るように頼まれたらどのような汎化を行なうであろうか．そのプログラマーは多分以下のような一般的な「常識」に基づいてこれらの絵を汎化するであろう．

- 絵の中の長方形についているラベルと，ASR データに含まれている文字列を比較すれば，絵の各要素と ASR データとの対応が分かる．
- 絵の中で左に 1 つだけある長方形が上司で，右の方に列を成しているのが部下であろう．
- 一般的な組織図では部下の人数は任意なので，生成するルールは任意人数の部下の長方形を並べることが出来なければならない．よって，部下を表す長方形は等間隔で並べるべきであろう．

TRIP3 も，PBVE を用いることによってほぼ上記のような人間と同じ汎化の方針で汎化を行なう．まず，デザイナーは MacDraw 風の簡単な作画ツールを用いて PR (視覚化した絵) の例を描く (図 4中の左上の絵)．この描画に際して，システムはレイアウトに関するデザイナーの意図を推論し適切な VSR，つまり horizontal などの幾何制約，を生成していく (図 4のステップ 1)．実際は，デザイナーが 1 つ図形を描くごとにシステムはその図形に関する VSR の推論を行い，次の 2 つの方法でデザイナーにその推論を示す．(1) 描画パネル中では，色付きの点線で推論された制約を表す．例えば図 5では，点線が，2 つの長方形が水平に並べられているらしいとシステムが推論したことを表している．(2) 確認用パネル (図 6) には確信度の高い推論から順に表示される．デザイナーは，各項目の横のチェックボックスを用いて，どの推論が正しいか示す．ただし最も確信度の高い推論はあらかじめチェックされているので，普通デザイナーはチェックボックスの内容を変更する必要はない．[1]

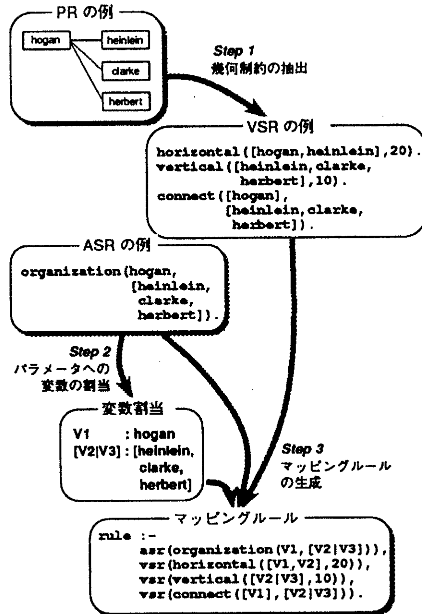

図 4: PBVE によるルール生成の過程

一方，デザイナーは図 7にあるように，対応する ASR

[1] Peridot[9] では 1 つ 1 つの推論に対して順に YES/NO の質問に答えなければならない．それに比較して，この確認方法は一度に複数の推論に対するデザイナーの意図を示すことができる点で優っている．

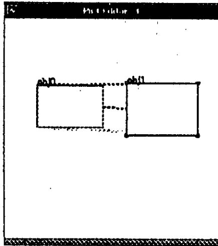

図 5: 描画パネル

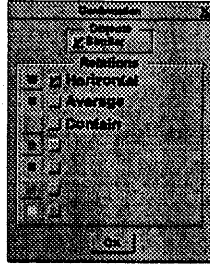

図 6: 確認用パネル

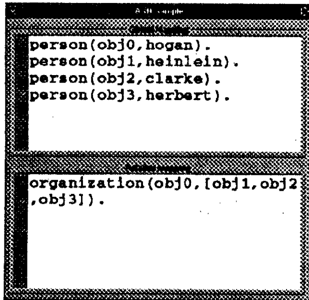

図 7: ASR 例入力パネル

も同時に入力する. これらの例の入力が終ると, デザイナーはメニューから "ルール生成" コマンドを選択し, システムは例を汎化して一般的な双方向変換ルールの生成を始める. まず, ASR の各節中のパラメーターに対して変数が割り当てられ, それに際して汎化が行なわれる (図 4のステップ 2). つまり, 図 4では,「上司は 1 人, 部下は任意数」という事実を反映して, 上司 hogan に対しては単純な変数 V1 が, 部下のリスト [heinlein, clarke, herbert] に対してはリスト変数 [V2|V3] が割り当てられている. この例から分かるように, 汎化を行なうには, デザイナーが例を与える際に, どこを汎化すべきかという意図を ASR 例の中でリスト構造を用いることによって示す必要がある. 最後に, 推論した VSR, 割り当てた変数を用いてルールを生成する (図 4のステップ 3).

4 TRIP3 システム

4.1 システム構成

上記のようにルールを生成し, また出来たルールを適用するため, 我々はプロトタイプシステム TRIP3 を NeXT Station Color 上で Objective-C と Prolog を用いて実装した. システム構成は図 8のようになっている. 全体はルール生成段階とルール適用段階に分かれており, ルール適用段階は基本的には TRIP2 の拡張である. ルール生成段階においては,

1. デザイナーは, ASR の例を入力し, また絵の例を MacDraw 風の簡単な直接操作描画ツールで描く.
2. Spatial Parser は, デザイナーが描いた絵の例から VSR を抽出する.
3. Inference Engine は, Spatial Parser が抽出した VSR とデザイナーが入力した ASR の例を汎化して, 双方向変換ルールを生成する.

ルール適用段階においては,

1. ユーザーが入力した ASR は, ルールによって対応する VSR に変換される. そして制約解消系 COOL (COnstraint-based Object Layout system)[5] が, VSR によって表現された幾何制約を解いて絵 (PR) を構成する.
2. ユーザーが絵を入力すると, Spatial Parser がそこから必要な幾何制約を抽出して VSR を生成し, その VSR はルールによって ASR に変換される.

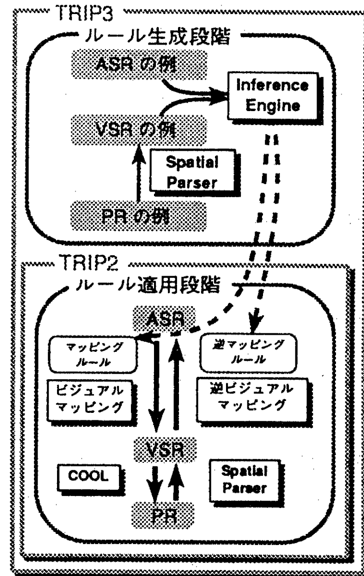

図 8: TRIP3 システムの構成

4.2 制約の十分性の検査

デザイナーの描いた絵から VSR を抽出する時, システムは幾何制約の十分性を保証しなければならない. 例えば, object[0].y=object[1].y=...=object[N].y という幾何制約だけでは, これらのオブジェクトの Y 座標は定まるが X 座標はまだ定まらない. よって COOL が幾何制約を解いてオブジェクトの配置を決めるためには, これらのオブジェクトの X 座標を定める他の何らかの幾何制約が必要である.[2] ここである幾何制約の集合を仮定したとき, X, Y, または両方の座標の定まった

[2] 前述の通り, COOL は系が制約過多の場合でも最小二乗法によって解くことが出来る.

オブジェクトを それぞれ “X 座標確定”, “Y 座標確定”, “完全座標確定” と呼ぶことにする.

```
select startObject;
unstableObjects = allObjects - {startObject};
/* '{}' denotes a set. */
xStableObjects = yStableObjects = {};
xyStableObjects = {startObject};
do{
  loop = NO;
  for each anObject in stableObjects do
    for each aRelation that includes anObject do
      if([aRelation checkStableObjects:anObject])
        loop = YES;
}while(loop);
```

図 9: 制約の十分性確認のアルゴリズム

図9に, 抽出した VSR (つまりそれが示す幾何制約) によって全てのオブジェクトが完全座標確定になるかどうか調べるアルゴリズムを Objective-C の擬似コードで示す. このアルゴリズムでポイントとなるのは, `[aRelation checkStableObjects:anObject]` というメッセージ送信である. ここで **aRelation** は抽出された VSR の1つ, また anObject はその VSR に関与しているオブジェクトの1つである. **checkStableObjects** というメソッドは, そのレシーバの VSR (aRelation) によってオブジェクト (anObject) の座標がどのような制限を受けるのかを調べ, その結果, オブジェクトは集合 **unstableObjects**, **xStableObjects**, **yStableObjects**, **xyStableObjects** の間を移動する. メソッド **checkStableObjects** での処理は各 VSR の種類によって異なる. 例えば, **horizontal** は関与しているオブジェクトの Y 座標を固定するので, `[horizontal checkStableObjects:anObject]` はおおまかには以下のような処理を行なっている.

- もしオブジェクト anObject が集合 unstableObjects に含まれていたら, そのオブジェクトを集合 yStableObjects に移す.
- もしオブジェクト anObject が集合 xStableObjects に含まれていたら, そのオブジェクトを集合 xyStableObjects に移す.

別の例として **x-center** を考えてみると, この幾何制約はある1つのオブジェクトの X 座標を他のいくつかのオブジェクトの X 座標の平均の位置に定める. よってこの場合, あるオブジェクトの X 座標を確定するには, 関与している他の全てのオブジェクトの X 座標が確定している必要がある. したがって, `[x-center checkStableObjects:anObject]` は以下のような処理を行なう.

- もし x-center に関与している anObject 以外のオブジェクトが全て集合 xStableObjects または xyStableObjects に含まれているなら:
  - もし anObject が集合 unstableObjects に含まれているのなら, それを集合 xStableObjects に移す.
  - もし anObject が集合 yStableObjects に含まれているのなら, それを集合 xyStableObjects に移す.

このようにして各 VSR によってオブジェクトの座標がどのような制約を受けるのか調べながらループが回り, 集合間でのオブジェクトの移動がなくなった時点でループが終了する.

ループが終了すると, システムは集合 **unstableObjects**, **xStableObjects**, **yStableObjects** が空集合であるか調べる. もしこれらのどれかが空でないときは完全座標確定でないオブジェクトがあることになるので, システムはそれらのオブジェクトの座標を確定するために VSR の追加, または変更を行なう. 組織図の例 (図4) では, システムが例の絵から VSR を抽出した段階では, **horizontal**, **vertical** はそれぞれ Y 座標, X 座標しか確定しない (つまり, パラメータ 20, 10 をまだ持っていない). よって, 長方形 **heinlein** は VSR **horizontal**, **vertical** によって完全座標確定だが, **clarke**, **herbert** は VSR **vertical** しか関与していないため X 座標確定である.

ここで人間が図4の例の絵を見たとき, どのようにして例に込められた意図を読み取り, この絵を汎化するか考えてみる. 人間なら, 部下を表す長方形は等間隔で一列に配置されていると考えるであろう. なぜならそれ以外に任意人数の部下を配置する方法がないからである. 同様に, TRIP3 でも対応する ASR の例に含まれているリスト構造 (つまり, 任意個の要素が含まれ得る) の存在を考慮して, VSR を変更する. 今の例では, VSR `vertical([heinlein, clarke, herbert], 10)` (3つの長方形は10 ピクセル間隔で垂直に配置されている) を新しく生成する.

このように ASR データのリスト構造から分かる VSR データを追加した後でまだ完全座標確定でないオブジェクトが残っていた場合は, システムはデザイナーにその旨を伝え, デザイナーに明示的に幾何関係ブラウザー (図10) を用いて幾何関係を示してもらう.

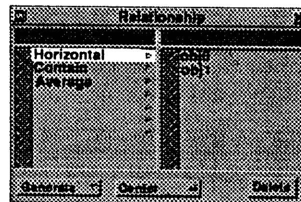

図 10: 幾何関係ブラウザ

5　再帰構造を持つデータに対するルール

ここまでで，組織図のような単純な構造を持ったデータに対するルール生成を見てきたが，TRIP3 は木構造のような複雑な再帰構造を持つものに対する視覚化にも対応している．一般に，再帰構造を持ったデータの例からその再帰性を見付け出すことは困難だが，TRIP3 ではデザイナーが例の中で再帰構造を明示的に示すことによって，この問題を解決している．つまり，デザイナーは絵がどのような規則で再帰構造を持っているのかという考えを持っているので，この手法は実際的であるばかりでなく，デザイナーにとっても例を示し易い手法であると言える．

例として，木構造を視覚化するための変換ルールの生成過程を簡単に示す．まず，デザイナーは図 12, 図 13の絵を描く．図 12は，再帰的な構造を示している．つまり，点線で描かれた長方形はこのルールで定義する木構造が再帰的に現れることを示している．図 13は，この再帰構造がどのように終るのか示している．この場合，ノードが現れることで再帰構造が終る．次にデザイナーは対応する ASR の例を入力する (図 14)．ここで rec0, rec1 は図中の再帰的出現を表す点線の長方形に相当する．実際にこのルールを適用する時には，図 15のようにこの rec0, re1 のあった場所には tree() という再帰出現がくる．

```
%% 再帰的出現に対するルール
visualize(tree(X,[H|L])) :-
  recursive([H|L]),               %(1)
  vsr(box(X)),                    %(2)
  vsr(horizontal([H|L],10)),      %(3)
  vsr(x-center(X,[H|L])),         %(4)
  vsr(y-order([X,H],10)).         %(5)

%% 末端に対するルール
visualize(tree(X)) :-
  vsr(box(X)),                    %(6)
```

図 11: 木構造ダイアグラムに対するルール

デザイナーがこれらの例を入力すると，システムは図 11のようなマッピングルールを生成する．このようにマッピングルールは再帰的な場合に対するルールと，末端に対するルールとの 2 つの部分からなる．以下，各行の意味を見ていく．1 行目は，このルールの再帰的適用を行なっている．2 行目と 6 行目は ASR データ X に対応する長方形を生成している．3 行目から 5 行目は幾何制約を宣言している．

このルールを用いて 図 15の ASR データを視覚化すると，図 16のような木構造ダイアグラムが得られる．

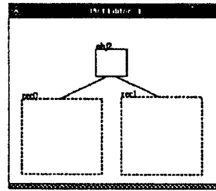

図 12: 木構造 PR 例 (再帰)

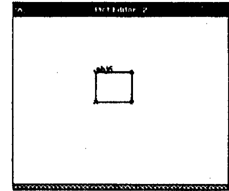

図 13: 木構造 PR 例 (末端)

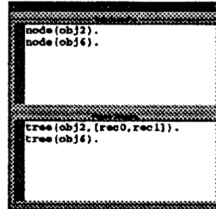

図 14: 木構造 ASR 例

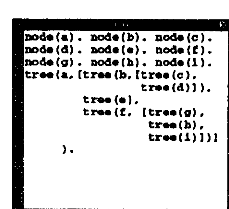

図 15: 木構造 ASR データ

6　その他のインターフェイスの実現例

組織図，木構造ダイアグラム以外にも，TRIP3 を用いて以下のような GUI の生成に成功している．(1) 標準的データベーススキーマである ER ダイアグラムの GUI, (2) cons-cell ダイアグラムの視覚化．

図 17, 18 は，ER ダイアグラムのための GUI の例である．この絵からルールを生成する時，システムは horizontal などの幾何制約ではなく，トポロジカルな制約である隣接 adjacent を探す必要がある．[3] 図 19, 20 は複雑な ER ダイアグラムをこのルールによって視覚化したものである．図 20 の各要素の配置は COOL 中のグラフレイアウトモジュールによって自動的に決定される [4]．

図 21, 22 は cons-cell ダイアグラム視覚化のためのルール生成時の，例の ASR データと絵である．木構造ダイアグラムの時と同様，再帰的な出現は点線の長方形によって明示的にデザイナーが描く必要がある．セルを表す長方形と左側の再帰的出現を覆っている灰色の長方形はこれらの 2 つの要素をデザイナーがグルーピングした

[3] 現在のインプリメンテーションでは，幾何制約の方がトポロジカル制約より高い確信度を持つようになっている．よって，デザイナーは確認用パネルを用いて明示的にトポロジカル制約を選択する必要がある．

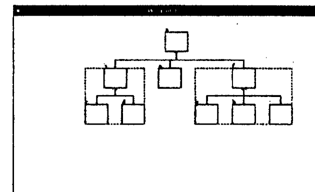

図 16: 木構造 PR データ

ことを表している．このグルーピングを表す長方形と右側の再帰的出現の間に一定の間隔があることをシステムに推論させることによって，デザイナーは生成したルールによって2つの再帰的出現が重なるような絵が生成されることを防いでいる．このルールによって，図23のASRデータは図24のように視覚化される．

```
entity(obj0,entity0).
entity(obj1,entity1).
attr(obj3,attr0).

con(obj0,obj1).
con(obj0,obj3).
```

図 17: データベーススキーマの例

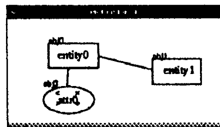

図 18: ER ダイアグラムの例

```
entity(a,alice).
entity(b,bob).
entity(c,cindy).
attr(a1,tokyo).
attr(a2,22).
attr(b1,newyork).
attr(c1,30).
con(a,b).    con(a,c).
con(a,a1).   con(a,a2).
con(b,b1).
con(c,c1).
```

図 19: データベーススキーマ

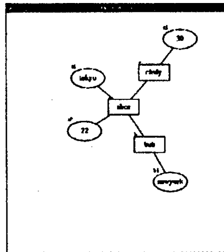

図 20: ER ダイアグラム

```
cell(obj0).
cell(obj6).
cell(obj9).

cons(obj0,rec1,rec4).
cons(obj6,rec7).
cons(obj9).
```

図 21: Cons-Cell データの例

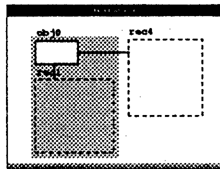

図 22: Cons-Cell ダイアグラムの例

7 関連研究

ここでは我々の研究に関連する分野の中でも，特に「例によるプログラミング (PBE)」に注目してみる．GUIの分野での PBE の活用には，大きく分けて2つの流れが見られる．1つはGUI環境を使うエンドユーザーがマクロを生成するのを PBE によって支援しようという流れ，もう1つは複雑な GUI の設計をするプログラマーを PBE によって支援しようというものである．我々の研究はこのうち後者にはいる．

```
cell(a). cell(b). cell(c).
cell(d). cell(e). cell(f).
cell(g). cell(h). cell(i).
cell(j).
cons(a,cons(b,cons(c),
          cons(d)),
     cons(e,
       cons(f,cons(g)),
       cons(h,cons(i),
            cons(j))))).
```

図 23: Cons-Cell データ

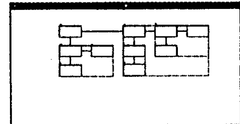

図 24: Cons-Cell ダイアグラム

前者の範疇に入る研究としては，以下のようなものが挙げられる．Eager[1] は Macintosh の Hypercard のユーザーの操作を監視して，そこから繰り返し行なわれる動作を検出してマクロを自動生成するシステムである．Metamouse[7] システムにおいては，お絵かきツールのユーザーが操作の例を与えるとシステムがその例を汎化して操作手続きを自動生成してくれる．Macro by example[6] では，ユーザーがお絵かきツールの操作記録の中からマクロにしたい部分を切り出してくると，システムが適切な汎化を行いマクロを生成する．

後者に含まれる研究には以下のようなものがある．Peridot[9] は，GUI プログラマーがメニューなどの GUI の部品を作るのを支援するシステムである．プログラマーが部品の外見を直接操作お絵かきツールで描きその動きの例を示すだけで，システムが汎化を行い部品を生成する．また DEMO II[2] も同様のシステムで，プログラマーはユーザーの動作とそれに対するシステムの反応をデモンストレーションすることで GUI のプログラミングをすることが出来る．

これらの従来の研究と我々の TRIP3 の間には，以下のような根本的な相違がある．すなわち，従来の研究での PBE の使い方の中心は，ユーザーまたはプログラマーの低レベルな動作 (例えばマウスボタンのクリックなど) の記録列の汎化にあった．それに対して，TRIP3 においては，PBE はアプリケーションのセマンティックスとそれに対応する視覚的表現との間の対応関係の汎化に用いられている．よって，我々の研究における汎化の考え方は，固有のシステムの固有の動作に依存しないという点で，より一般性の高いものである．

8 まとめと今後の課題

本研究では，抽象的データとそれを視覚化した絵との間の双方向変換を，例を通して実現する手法を提案した．プロトタイプシステム TRIP3 においては，デザイナーが絵の例を直接操作によって描き，それに対応する抽象的データの例を入力するだけで，システムがそれらの例を汎化して一般的な双方向変換ルールを生成する．また，絵を描く際には，システムがデザイナーの意図を反映した絵の構成要素間の幾何制約を推論するので，デザイナーは細かい指定をする必要がない．

今後の課題としては，以下のようなものを考えている:

(1) 現在用いている双方向変換ルールの記述法は，もともとTRIPにおいて抽象的データの視覚化という向きの変換を実現するために作られた．よって絵から抽象的データへの変換に際しては，直接操作を考慮していないなどいくつかの不都合な点がある．よって現在，新しいルールの記述法を検討中である．(2) 現在のシステムは絵からアプリケーションデータへの変換(ビジュアルパージング)の効率に問題がある．よって，アプリケーションデータ，およびマッピングルールの記法を改良し，効率的なビジュアルパージングを実現する．図25 に，(1)(2) の改良を実現するため現在検討している新しいアプリケーションデータとその視覚化のためのルールの記法を示す．これは今までの例でも用いてきた長方形とラベルで出来たオブジェクトを表している．従来の記法との大きな違いは:

- アプリケーションデータの定義に型と内部構造が取り込まれている．この例では worker という型は，文字列型の name という要素からなる．
- アプリケーションデータの各型の間でどのような依存関係があるのか分かるので，ボトムアップにビジュアルパージングを行なうことが出来る．よって無駄なバックトラックを回避し効率的なパージングが可能となる．

```
define-asr-type{
  name : string;
} worker;

define-mapping{
  object :
    box(bid, 20, 10);
    word(lid, name);
  relation :
    contain(bid, lid);
} worker;
```

図 25: 型を採り入れた新しいルール記法

(3) 現在は一組の例からルールを生成しているが，これではデザイナーの意図が完全に伝わるとは限らない．よって複数の例を汎化することによってルールを生成するように，システムを改良する．(4) 汎化機構の強化，およびどのようにルールを生成するかを記述するメタルールの採用などによって，より複雑なルールを生成できるようにする．(5) 現在の TRIP3 システムは，絵を描くモジュールとルールを適用するモジュールに分かれており，その間の通信がスピードのネックとなっている．結果として，プログラマーの指定した幾何制約を実時間で解いて絵に反映することが出来ていない．通信のオーバーヘッドの削減，および DeltaBlue アルゴリズム [3] などのインクリメンタルな制約解消系の採用などにより，実時間での絵の更新を達成する．

参考文献

[1] Cypher, A., "Eager : Programming Repetitive Tasks by Example," in *ACM Human Factors in Computing Systems*, 1991, pp. 33–39.

[2] Fisher, G. L., D. E. Busse, and D. A. Wolber, "Adding Rule-Based Reasoning to a Demonstrational Interface Builder," in *Proc. of ACM User Interface Software and Technology (UIST)*, Nov. 1992, pp. 89–97.

[3] Freeman-Benson, B. N., J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Comm. ACM*, vol. 33, no. 1, Jan. 1990, pp. 54–63.

[4] Kamada, T. and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, vol. 31, no. 1, Apr. 1989, pp. 7–15.

[5] Kamada, T. and S. Kawai, "A General Framework for Visualizing Abstract Objects and Relations," *ACM Trans. Graphics*, vol. 10, no. 1, Jan. 1991, pp. 1–39.

[6] Kurlander, D. and S. Feiner, "A History-Based Macro By Example System," in *Proc. of ACM User Interface Software and Technology (UIST)*, 1992, pp. 99–106.

[7] Maulsby, D. L. and I. H. Witten, "Inducing Programs in a Direct-Manipulation Environment," in *ACM Human Factors in Computing Systems*, 1989, pp. 57–62.

[8] Miyashita, K., S. Matsuoka, S. Takahashi, A. Yonezawa, and T. Kamada, "Declarative Programming of Graphical Interfaces by Visual Examples," in *Proc. of ACM User Interface Software and Technology (UIST)*, 1992, pp. 107–116.

[9] Myers, B. A., *Creating User Interfaces by Demonstration*. San Diego: Academic Press, 1988.

[10] Takahashi, S., S. Matsuoka, A. Yonezawa, and T. Kamada, "A General Framework for Bi-Directional Translation between Abstract and Pictorial Data," in *Proc. of ACM User Interface Software and Technology (UIST)*, 1991, pp. 165–174.