Mochi Sheet: ズーミングとレイアウト編集機能の統合

豊田正史 志築文太郎 高橋伸 柴山悦哉
東京工業大学 情報理工学研究科 数理・計算科学専攻

ビジュアルプログラムのエディタに適したズーミングおよびレイアウト編集のインタフェースを提案する．Mochi Sheet はこれを実装した汎用ライブラリであり，なめらかなズーミング，図形の重なりの回避，およびズーム操作の選択的なアンドゥ機能を提供する．この結果，Mochi Sheet を用いて実装したエディタでは大規模な階層構造の編集を，編集部分の拡大・編集・ズームのアンドゥという手順の繰り返しによって効率良く行うことができる．

Mochi Sheet: Integration of Zooming and Layout Editing

We propose an integrated interface for zooming and layout editing which is appropriate for visual program editors, and implement the interface as a library for the AMULET Toolkit. Mochi Sheet provides continuous distorted zooming, and a selective undo function of zooming operations, and avoids overlapping of graphical objects during editing. Using graphical editors implemented with Mochi Sheet, we can edit large hierarchical data structures efficiently by repeating following processes: zooming a part, editing the part, and undoing zooming of the part.

1 はじめに

データフローグラフ，コントロールフローグラフなどのグラフ表現を扱うビジュアルプログラムのエディタにおいては，大規模なプログラムの扱いが問題となる．大規模なビジュアルプログラムの編集に特徴的な点を以下に示す．

- プログラムは階層構造をなす．通常プログラムは複数のモジュールから構成され，モジュールは複数のグラフから構成され，グラフは複数のサブグラフから構成される．この結果，プログラム全体が巨大な階層構造をなすことになる．
- グラフのノードは重なってはいけない．ノードにはプログラムの意味を表す情報が含まれるため，ノードが重なるとプログラムの意味が理解できなくなる．
- あるモジュールで定義されている部品を他のモジュールで利用する際には，モジュール・グラフ間で部品をドラッグ&ドロップする操作が一般的である．このためモジュール同士，グラフ同士を重ならないように同一画面に表示する必要がある．

実際には，以上の特徴を持つ図形構造を狭いスクリーン上で編集しなくてはならない．したがって，どのように大規模な図形の階層構造を重なりを避けながら狭いスクリーンに表示し，そのレイアウトを効率良く編集できるようにするかという点が問題となる．

このうち表示に関してはズーミングの手法が有効であり，すでに Pad++ [3] といった汎用のライブラリが発表されてきている．数あるズーミング手法のうちビジュアルプログ

ラムの表示に有効なのが fisheye view [4] である．Fisheye view ではデータ全体の概観表示および一部の詳細表示を同じビューで行うため，プログラム全体の構造を見失うことなく一部分の詳細を見ることができる．この手法から派生した Rubber Sheet [5]，Continuous Zoom [2] は，2 次元平面上の入れ子状に構成された図形の階層構造に対して，(a) 図形の相対位置を保ち，(b) 図形の重なりを避けながら，(c) 複数のフォーカスを指定したズームを行うという点で我々の目的に適合している．特に [2] はズームによる変化をなめらかにアニメーション表示するため，視点の変化を把握しやすいという特徴を持っている．

一方，レイアウトの編集作業にはズームに加えて，図形の追加，削除，および移動が最低限必要になる[1]．しかし上で述べたズーミング手法では，大規模な図形のレイアウトを編集するインタフェースがどうあるべきかという点についてはあまり考慮されていない．[5] では拡大部分の編集が可能であるとされているが，編集操作後の図形の重なりの回避については触れられておらず，[2] では初期配置は重なりの無いものとしており，レイアウトの編集については考慮されていない．

我々の目的は，[2] と同様のなめらかなズーミングインタフェースにレイアウト編集の機能を統合したビジュアルプログラムのエディタを構築することである．現在我々はビジュアルプログラミング環境 (以下 VPE)KLIEG [8] [7] におけるエディタをこの目的に沿って開発している．KLIEG はプロセスとその間のデータフローに基づく VPE である．エディタでは，プロセスのネットワーク構造，および関連するプロセス定義をまとめたモジュールの階層構造を表示し，編集を可能にする．

エディタでは，1 つのウインドウの中で編集したい部分を適宜拡大して編集を行うことができる．図 1では 3 つのモジュールが表示されており，そのうちの qsort モジュールを拡大し (図 2)，次に qsort モジュール中のネットワーク図を拡大している様子 (図 3) を示している．拡大したネットワーク図は詳細が見えるようになり，周りの図はそれに応じて縮小され，詳細が見えなくなっている様子がわかる．また拡大後も図形の相対位置は保たれている．また，2 つのモジュールを拡大しておいて，その間でプロセスをドラッグアンドドロップするという編集操作も容易に行うことができる．

このエディタの設計に際しては以下のような問題点が生じた．

編集による図形の重なりの回避 図形の追加，削除，および移動を行った後も，図形が重ならないよう自動的に再配置することが必要となる[2]．この問題に関しては力学モデルに基づいて頂点の大きさの異なるグラフを配置するアルゴリズムが [6] に述べられている．しかし，再配置の計算を高速に行うことが難しいため，なめらかなズームを行うために必要な応答速度が得られない．

ズームによって歪んだレイアウトを元に戻す方法 大規模な図形レイアウトの編集作業では，(1) 編集部分の拡大，(2) 拡大した部分内の編集，(3) 拡大した部分を元に戻す，という手順を繰り返し行うことになる．(3) の元に戻す手順では，ズーム操作で戻す方法があるが，これは元の大きさに正確に戻すことが難しい．また普通のアンドゥ操作を導入しても，拡大した部分に行った編集操作を無視してアンドゥを行うことはできない．

[1] リサイズはズームと区別が付きにくい のでズームで代用する．
[2] グラフにおける辺の重なりの回避は扱わない

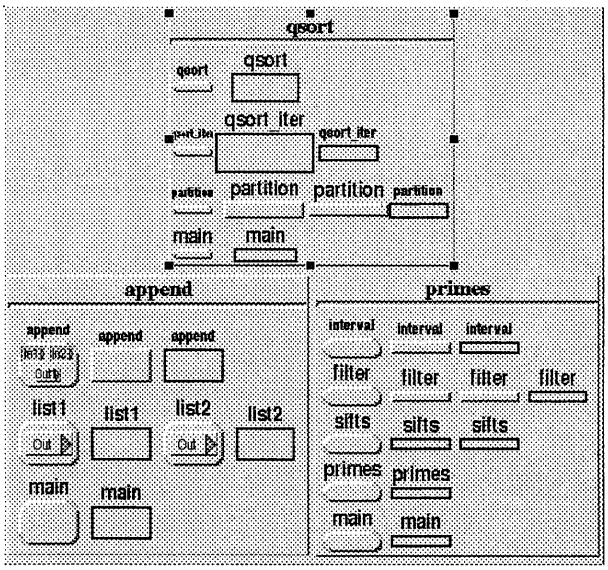

図 1: KLIEG のエディタ

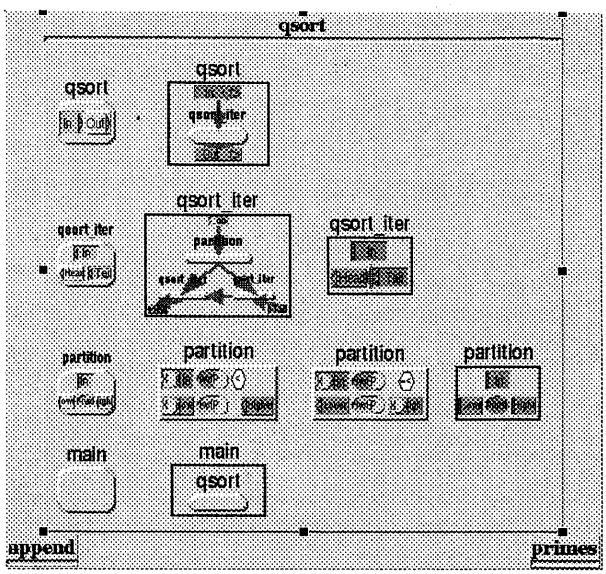

図 2: qsort モジュールを拡大 (ズームイン)

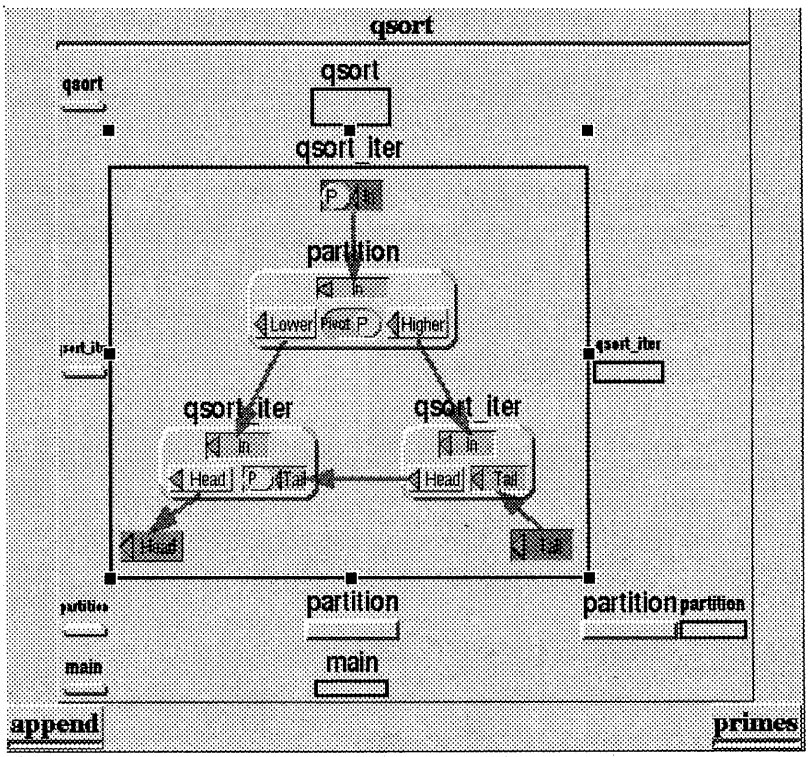

図 3: ネットワーク図を拡大

以上の問題点を踏まえて，我々はまず汎用のズームおよび編集のためのライブラリ Mochi Sheet を作成し，これを用いて KLIEG のエディタを実装した．Mochi Sheet は他にもディレクトリツリーのエディタなど様々なアプリケーションに応用できる．Mochi Sheet は，以下のような特徴を持っている．

- 編集操作後の図形のレイアウトを基にグリッドを動的に生成して，その上に図形を並べる制約を導入した．グリッドは比較的きつい制約であるが，細かいレイアウトを必要としない VPE のようなアプリケーションには十分である．これにより編集操作後に図形の重なりを回避するための再配置を単純な計算で高速に行うことができる．また後述するように空いている空間を探しやすいという利点がある．これを利用して [2] では利用できない空きスペースを図形に割り振ることが可能になった．
- ある図形におけるズーム操作のみを選択的にアンドゥするインタフェースを提供する．この機能により，拡大した編集部分を簡単に元の大きさに戻すことができる．また図形を編集履歴中における好みの大きさに一回の操作で戻すことが可能になる．
- [2] と同様にズームをなめらかに行う．
- 様々なズームアルゴリズムを階層構造中に混在可能にした．例えば，小さくなったときに詳細を表示せず，名前だけは表示する図形を階層の中間に置くことが容易である．したがって [3] で述べられている semantic zoom を容易に実現できる．

2 Mochi Sheet

この節では，Mochi Sheet のインタフェース，およびズーム・レイアウトのアルゴリズムに関して説明する．Mochi Sheet は，矩形の領域を持つ図形[3]を扱う．矩形の中に子供にあたる矩形を入れ子状に配置することにより，階層的な木構造を構成できる．以後，木構造の節および葉にあたる矩形をノードと呼ぶ．また，内部に子ノードを持つノードをクラスタノードと呼んで区別する場合がある．

2.1 インタフェースと挙動

Mochi Sheet における編集作業は，通常以下のような手順の繰り返しによって行われる．但し図形のレイアウトに影響をあたえない編集操作についてはここでは扱わない．

1. 編集したいノードを拡大 (ズームイン) する，
2. 拡大したノードの内部についてノードの追加，削除，移動による編集を行う，
3. 拡大したノードを元の大きさに戻す (ズームのアンドゥ)．

この手順によって基本的なレイアウト編集を効率良く行うことができる．ここではそれぞれの手順における Mochi Sheet のインタフェースについて述べる．

[3]輪郭が矩形である必要はない．

ズーム

ズームは，選択したノードの8方向につけられるハンドルで行う．ハンドルをつかんで動かしている間，ノードとその周辺のノードが重ならないようになめらかに変化する．

Mochi Sheet はレイアウトの編集が主な目的であるため，個々の図形の大きさを細かく指定できるハンドルインタフェースを採用している．目的がブラウズのみならばこのインタフェースは煩わしいものとなるため，Continuous Zoom ではマウスのボタンを押している間，ズームイン・アウトするインタフェースを採用している．しかし，このインタフェースでは縦横比を変更するズームができない．また Rubber Sheet では，図形とは独立した領域を指定しそれをラバーバンドとしてズームを行うため，個々の図形の大きさを細かく指定するのが難しい．

クラスタノードを小さくした結果，子ノードが元の大きさで表示できなくなったときに，その内部に対してズームアウト (縮小) が行われる．逆に大きくした場合，プログラマはズームイン (拡大) を行うか，行わないかをクラスタノードに対して指定することができる．ズームインを行わない場合，子ノードは元の大きさのままで，各ノードに均等に空間が分配される．これらは用途によって使い分けられる．

ノードの追加，削除，移動

ノードの追加，削除，移動については，通常の図形エディタと同様の操作で行うことができる．これらの操作が行われるとすべての図形が重ならないように再配置が行われる．移動に関してはクラスタノード間にまたがる移動も行うことができる．

ズームのアンドゥ

ズームのアンドゥは，対象とするノードへのズーム操作のみのアンドゥを行う．例えば，ある図形をズームした後移動して，アンドゥを行った場合，図形は元の大きさには戻るが，元の場所には戻らない．このアンドゥ機能を用いることで編集のために拡大したノードを簡単に元の大きさに戻すことができる．通常のアンドゥ機能では拡大したノードの内部への編集操作を無視してアンドゥを行えない．このためノードの大きさを元に戻すにはズームアウトの操作を行うしかない．しかしズームアウト操作で正確に元の大きさに戻すのは難しい．

アンドゥはノードのポップアップメニューから行う．メニューは一般的なアンドゥと同様に，アンドゥ，リドゥの項目を持つ．さらに次に述べる選択的アンドゥのためのダイアログを呼び出す項目も持っている．

選択的なアンドゥは，直前のズームだけではなく，それ以前のズーム操作を選択的にアンドゥできる機能である．これにより，ノードをある時点での大きさに一度の操作で戻すことが可能になる．選択的アンドゥはズーム操作の履歴を示すダイアログボックスから，大きさを選択することで行う．

アンドゥの代替案としては，図形のスケールをある時点で 1 に変更するというインタフェースも考えられる．しかしスケールを 1 に変更した時点での大きさにしか戻すことができないため，アンドゥインタフェースを採用した．

2.2 ズーム・レイアウトアルゴリズム

Mochi Sheet では，ノードの追加・移動の後もノードが重ならないようにするため，子ノードを動的に生成した格子の上に並べる制約を導入している．この格子を用いて各ノードが重ならないようにズームを行う．また格子制約を利用して，空いているスペースを探しノードに割り振ることも行う．以下では，まず基本的なズーム・レイアウトのアルゴリズムを述べ，続いて空きスペースを探し有効利用する方法について述べる．

2.2.1 基本アルゴリズム

基本アルゴリズムは，1つのクラスタノードの子ノードのレイアウトを決定する．このアルゴリズムは対象とするクラスタノードの寸法，および，各子ノードの座標および寸法を入力とし，各子ノードを実際に表示する際の座標および寸法を出力する．階層構造に対しては，このアルゴリズムをルートのクラスタノードから再帰的に適用することで全体のレイアウトを定められる．以下に実行手順を示す．

1. **グリッドの生成** x, y 各軸についてノードの中心の座標を比較し，しきい値以下の距離のものは同じ格子線上に置くようにグリッドを生成する[4](図 4).
2. **要求幅の計算** ノードが重ならないために必要な各区間 (x1～x3, y1～y4) の要求幅を計算する．ある x 軸上の区間 xk の要求幅 R_{xk}は以下の式で表される．

$$R_{xk} = \max(\frac{W_{(k-1,j)}}{2}) + \max(\frac{W_{(k,j)}}{2})$$

   ただし$W_{(i,j)}$は，図 4の右図における (i, j) の示す格子点の要求する幅を表す．格子点の要求する幅は，格子点に矩形がのっているときはその矩形の幅，のっていないときは 0 となる．例えば，x2 の要求幅は r2 と r3 の幅を$\frac{1}{2}$したものの和のとなる．y 軸上の区間の要求幅は矩形の高さについて同様の計算を行うことで求める．
3. **ノードの配置** x,y 各軸について要求幅の和 R_x, R_yを計算する．R_x, R_yがクラスタノードの幅 C_x, C_yより大きい場合，x,y 各軸について$\frac{C_x}{R_x}, \frac{C_y}{R_y}$というスケールで各子ノードの大きさを縮小する (図 5)．要求幅の和がクラスタノードの幅より小さく，拡大を行う指定の場合，同じスケールで拡大を行う．拡大を行わない指定の場合，格子の幅をできるかぎり平均化するように余りのスペースを割り振る (図 4 右).

ノードの追加・削除・移動が行われた場合，1 のグリッドの生成からやり直す．ズームが行われた場合には 2 からやり直すだけで済むため，なめらかにズームを行うことができる．

2.2.2 空きスペースの割り振り

グリッドにノードをのせるという制約を導入したことで，空きスペースを有効にノードに割り当てることが可能である．スペースの割り振りは要求幅がクラスタノードの幅より大きいときに行う．例えば図 6に示した 2 つの例では斜線で示した格子の幅を狭めてスペースを節約し，ノードの元の大きさを保つことができる．ただしノードの中心点の相対位置は保存するようにする．Continuous Zoom では，ノードの中心点ではなく頂点の相対位置を保つためこのような空きスペースを利用することができない．

[4] ノードが同じ格子点にのらないように前処理が行われていることを仮定する

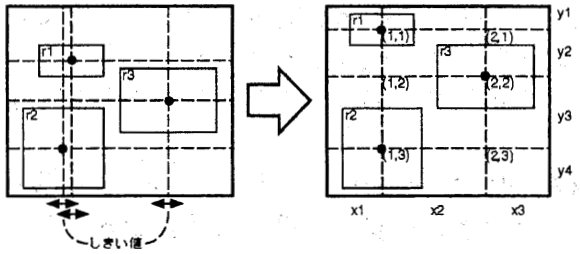

図 4: グリッド生成とノードの配置

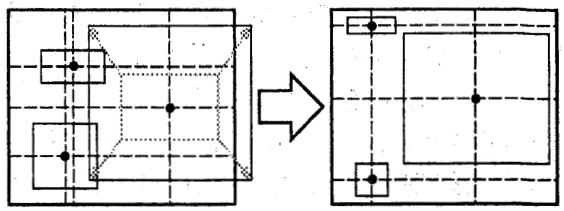

図 5: 要求幅が大きくなった場合の配置

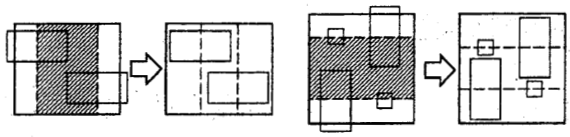

図 6: 空きスペースの割り振り

スペースを割り振るときには，x,y 軸それぞれに対して各区間を可能な限り狭めた幅 (狭め幅) を計算する．ある x 軸上の区間 xk の狭め幅 S_{xk} は以下の式で表される．

$$S_{xk} = \max(\frac{W_{(k-1,j)}}{2} + \frac{W_{(k,j)}}{2})$$

要求幅とこの狭め幅との差 (xk の場合 $R_{xk} - S_{xk}$) が大きいものから格子を狭めていく．このとき x 軸，y 軸の区別はせずに順番を決める．ただし x,y 軸にまたがる区間 (x2 と y3 など) を比較する際には，スケール (2.2.1 節の手順 3 で計算した値) のより小さい軸の方を優先する．すなわち，より縮小されている軸の幅を優先的に狭める．例えば図 4 の右の図では，x2, y3, y2 の順に狭めていく．ある格子を狭めると他の格子が狭められなくなる場合があるが，このときは狭められなくなった格子を無視して残りの格子を狭める．

3 実装

Mochi Sheet は Amulet Version 2.0 [1] 上に実装した．Amulet はプロトタイプオブジェクトシステムと制約に基づく GUI 開発環境である．Amulet では，図形オブジェクトの集約を行う group オブジェクトが提供されている．Mochi Sheet のクラスタノードは，この group オブジェクトを継承して実装したため，Amulet の group と同様に扱うことができる．

ズームアルゴリズムは，group 自身の寸法，および group 中の図形オブジェクトの位置および寸法に対する制約として実現している．このためズームを行った場合，Amulet の制約解消系によって，位置・寸法の必要な部分についてのみ再計算が行われる．例えば，階層構造の中ほどの group のズームを行うと，その group に含まれる図形，および 1 階層上の group に含まれる図形の再計算が行われる．

ズームアルゴリズムでは group に直接含まれる (1 階層下の) 図形のみを変更するため，階層構造中に異なるズームアルゴリズムを持つ group を加えることが可能である．これ

により [3] において述べられている semantic zoom と同等の機能を実現しやすくなっている．例えば，第1節で示したエディタではモジュール，プロセスなどは，どのような大きさになっても名前だけは表示するという group で実装されており，その内部にまた Mochi Sheet の group を含んでいる．

現在の実装は，Pentium 90MHz の PC 互換機 (OS は Linux) 上で行っている．50 個程度の図形が画面上に表示されていて，そのすべてが大きさを変える場合には秒間 5 フレーム程度で動く．数が 100 個程度になると秒間 2 フレーム程度である．なお，Amulet では 100 個の図形を表示するのに約 0.2 秒かかるため，秒間 5 フレーム程度が限界である．

4 まとめと今後の課題

ズーミングのインタフェースにレイアウト編集機能を統合する手法を提案し，これを汎用ライブラリとして実装した．ユーザは大規模な図形構造の編集を，拡大・編集・アンドゥ操作の繰り返しにより効率良く行うことができる．

今後の課題としてはより一般的な幾何制約のもとでの編集を可能にすることがあげられる．またユーザテストを行ってインタフェースの有効性を調べることも必要である．

参考文献

[1] Amulet Project Home Page. http://www.cs.cmu.edu/afs/cs/project/amulet/www/amulet-home.html.

[2] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Space. In *Proceedings of UIST '95*, pp. 207–215, November 1995.

[3] Benjamin B. Bederson and James D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of UIST '94*, pp. 17–26, November 1994.

[4] George W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, pp. 16–23. Association for Computing Machinery, 1986.

[5] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss. Streching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *Proceedings of UIST '93*, pp. 81–91, November 1993.

[6] Xiaobo Wang and Isao Miyamoto. Generating customized layouts. In *Proceedings of Graph Drawing '95*, pp. 504–515, September 1995.

[7] 志築文太郎, 豊田正史, 高橋伸, 柴山悦哉. 並列ビジュアルプログラミング環境 KLIEG: プロセスネットワークパターンを利用した、再利性の向上と実行表示の 効率化. インタラクティブシステムとソフトウェア IV, 1996.

[8] 豊田正史, 志築文太郎, 高橋伸, 柴山悦哉. 並列ビジュアルプログラミング環境 KLIEG: プロセスネットワークパターンによる柔軟な再利用機構の導入. 情報処理学会研究報告 96-PRO-6(1996 年 3 月 26 日・27 日), 電子情報通信学会技術研究報告 SS95-46(1996-03), March 1996.