ビジュアル並列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用性の向上と実行表示の効率化

A Visual Parallel Programming Environment KLIEG: Reuse of Program Components and Visualization of Program Execution by Process Network Pattern

志築 文太郎　豊田 正史　高橋 伸　柴山 悦哉*

Summary. We propose a new visual programming methodology based upon *process network pattern* which is a visual template that defines topologies and protocols of parallel programs. In our methodology, expert programmers define process network patterns and novice programmers instantiate them with concrete processes to construct their own programs. Our visual parallel programming environment KLIEG supports this methodology. Process network patterns can be defined and instantiated interactively. Particularly, its instantiation can be performed using its drag&drop interface. KLIEG also animates the dynamic behavior of program executions, preserving the process network topologies defined by process network patterns, automatically. Even if process networks to be displayed become so large, a programmer can check the arbitrary parts of the networks by its zooming interface.

1　はじめに

並列プログラミングの設計段階において、プロセス、およびプロセス間の通信路からなるデータフロー図を用いることが多い。しかしながら、このデータフロー図をテキスト言語で実装した場合、そのプログラムを解読し構造を把握することは困難である。また、その実装作業は煩雑である場合がある。現在我々が開発中のビジュアル並列プログラミング環境 KLIEG は、設計図と類似した図式を直接プログラムとして編集する機能、およびその図式表現に基づいたプログラム実行の可視化機能を提供する。さらに KLIEG 環境は以下に挙げる特徴を持つ。

- プロセスネットワークパターン [12] に基づくプログラミングにより、ネットワークトポロジー、およびプロトコルの再利用が可能である

---

[*] Buntaro Shizuki, Masashi Toyoda, Shin Takahashi, Etsuya Shibayama, 東京工業大学 情報理工学研究科 数理・計算科学専攻

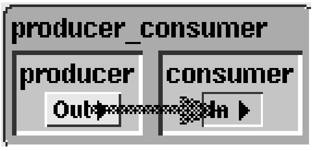

図 **1.** KLIEG による producer-consumer パターンの定義

- プログラム実行時に動的に生成されるプロセスネットワーク、プロセス内データ、およびプロセス間で通信されるデータを統一的な枠組みを用いて効率よく可視化する

KLIEG 環境は、データフロー図をプログラムとして編集するためのエディタ、および完成したプログラムの実行過程を可視化するトレーサから構成される。以下、第2節では我々が提案したプロセスネットワークパターンに基づいたプログラミングについて述べ、これを支援する KLIEG エディタのインタフェースについて述べる。第3節ではデータフローに基づくプログラミング言語の実行を統一的に可視化する手法について述べ、これを実装した KLIEG トレーサについて説明する。4節に関連研究を挙げる。

2　プロセスネットワークパターン

並列プログラムでは producer-consumer、divide-and-conquer など、さまざまな問題に共通したネットワークトポロジー、およびプロセス間のプロトコルを用いることが多い。しかし、新しくビジュアルプログラムを作成するたびに同じネットワークトポロジー、またはプロトコルを定義し直すのは煩雑である。これに対して我々はプロセスネットワークパターンを提案した [12]。プロセスネットワークパターン (以降、単にパターンと呼ぶこともある) はネットワークトポロジー、およびプロトコルをプログラミングにおける基本構成要素として扱うための機構で、これに基づいたプログラミングを行うことによって上記の問題を解決することができる。本節では、我々が提案したパターンに基づくプログラミングについて説明する。

2.1　パターンに基づくプログラミング

パターンは一部のプロセスが未定義のプロセスネットワークで、ネットワークの骨組みを表わすものである。プログラマは、(1) ライブラリに登録されているパターンをコピーする、既存のパターンを組み合わせる、または新たに定義することによってパターンを用意し、(2) パターン内の未定義プロセスに別に定義したプロセスを代入する、という 2 段階の操作によって、パターンで定義されたネットワークを利用した新たなプロセスを定義する。

図 1に KLIEG で定義できる最も単純なパターンの例として producer_consumer パターンを示す。ここで producer、および consumer が未定義プロセスを表わし、これをホールと呼ぶ。producer ホールに存在する Out、および consumer ホールの In は他のプロセスと通信するためのポートである。さらに producer の Out から consumer の In に張られているリンクは 2 つのプロセス間の通信路である。

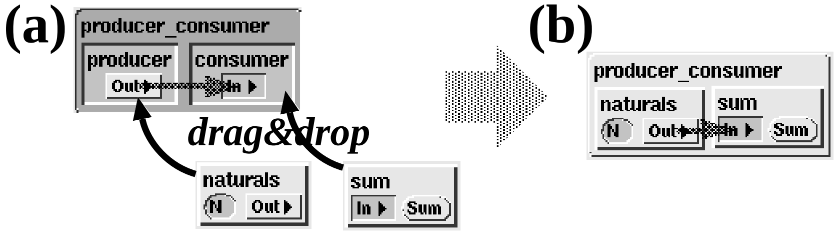

図 2. パターンを用いたプロセス定義

また、一般にホールはくぼんだ長方形で表現される。producer の Out ポート、および consumer の In ポートはデータの列を転送するためのストリームポートである。ポートには、データの列を転送するためのストリームポートの他に、一つのデータのみを受け渡しするシングルトンポートもある。ストリームポートは長方形、シングルトンポートは角の取れた長方形で表現される。さらにポートには入力専用のものと出力専用のものがあり、それらは凹凸で区別される。図 1中の Out は出力用、In は入力用ポートである。

プログラマは、ホールに他で定義したプロセスを代入することによって、パターンを利用することができる。この代入は KLIEG エディタ上で drag&drop により行える。ここでは producer_consumer パターンを利用して、自然数 N を受け取り$\sum_{n=1}^{N} n$ を計算するプロセス sum_of を定義する。あらかじめ図 2(a) 下に示されるような、入力シングルトンポート N から自然数を受け取り Out ポートに 1 から N までの整数列を出力する naturals プロセス、および入力ポート In から入力される数列の和を求め Sum ポートに出力する sum プロセスが別に定義されているとする。パターンの利用は図 2に示されるように、producer_consumer パターンの producer ホールに naturals プロセスを、consumer ホールに sum プロセスをそれぞれ drag&drop することで行える。結果として、それぞれのプロセスがホールに代入され、図 2(b) を得る[1]。この時、generator ホールの Out ポートに張られていたリンクが自動的に naturals プロセスの対応するポートに張られる[2]。この機構によって、パターンで定義されたネットワークトポロジーを保持した状態でプロセスを定義することができる。

ここで、ホールに定義されているポートがホールに代入できるプロセスを限定している。プロセスをホールに代入するためには、そのプロセスが少なくともホールに定義されているのと同種のポート (入出力、およびストリームポートかシングルトンポート) を持たなければならない。これによって型の合わない不適当なプロセスがホールに代入されることをある程度防ぐ。

図 2(b) で得られたプロセスネットワークを使った sum_of プロセスの定義を図 3 に示す。producer_consumer 部分にはパターン外部と通信するための N、および Out

---

[1] 同時に producer_consumer パターンの色が変化する。drag&drop する前は未定義状態であることを表わす暗い色であったのに対し、drag&drop 後は明るい色に変化しパターン内のすべてのホールがプロセスによって定義されたことを表現する。

[2] 自動的に張られたリンクがプログラマの意図したように張られていない場合、プログラマはそのリンクを正しく張り直す必要がある。

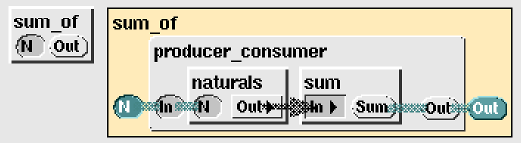

図 **3.** パターンを用いて定義した sum_of プロセス

ポートを追加してある。さらに図 3に示されている、色の濃い In、および Out ポートは sum_of プロセスの外部と通信するためのポートである。なお、図 3左に示されているアイコンは sum_of プロセスのインタフェースを定義するものである。このアイコンをホールに drag&drop して sum_of の定義を利用することができる。

以上のようにしてプログラマは、基本的にパターンを利用することでプロセスネットワークのトポロジーを決定し、エディタが提供する drag&drop のインタフェースによって、新しくプロセスを階層的に定義することができる。

2.2 パターンの階層的な定義

プログラマは、複雑なパターンを階層的に定義することができる。これは、ホールに、他で定義したパターンを drag&drop することによって行う。階層的なパターン定義の例として、ここでは master-worker パターンを扱う。

master-worker パターンは負荷分散を行いつつ問題を解くプロセスネットワークを定義する。このネットワークは master プロセス、および並列に動作する複数の worker プロセスから構成される。master プロセスが部分問題を生成し、計算待ち状態にある worker プロセスにその部分問題を割り当ててゆく。worker プロセスは受け取った部分問題の解を生成して master プロセスに送る。この master-worker パターンがあらかじめ定義されている場合、プログラマは部分問題の列を生成するプロセス、およびその部分問題を解くプロセスを master-worker パターンに与えるだけで負荷分散しつつ問題を解くプロセスを定義することができる。

図 4に master-worker パターンを階層的に定義する手順を示す。はじめに master_worker パターンを図 4(a) 右上のように定義する。パターン内の master ホールは問題を生成する部分に対応する。workers ホールはその問題を解くための worker プロセス群を生成する部分である。これらのホールに、図 4(a) 左のように定義しておいた master パターン、および workers パターンを drag&drop する。その結果、図 4(b) のように階層的に定義された master_worker パターンを得る。ここで master パターンは generator ホール、および dispatcher プロセス[3]を内部に持つ。generator が部分問題を生成し、その問題を dispatcher が worker プロセスに割り当てる。workers パターンは、同種のプロセスの列を動的に生成するためのシーケンスパターンと呼ばれるメタパターンを用いて定義されている。workers パターンのように、シーケンスパターンを用いて定義したパターンでは 3 つのホールが示される。これらは生成されるプロセス列の先頭、2 番目、および末尾のプロセスを表わす。この 3 つのいずれかに他で定義されたプロセスを drag&drop することによって他の 2 つにも同じプロ

---

[3] 図 4中の dispatcher プロセスは省略表示されている。プログラマは必要に応じて省略表示からポートが見えるような表示に切り替えることができる。

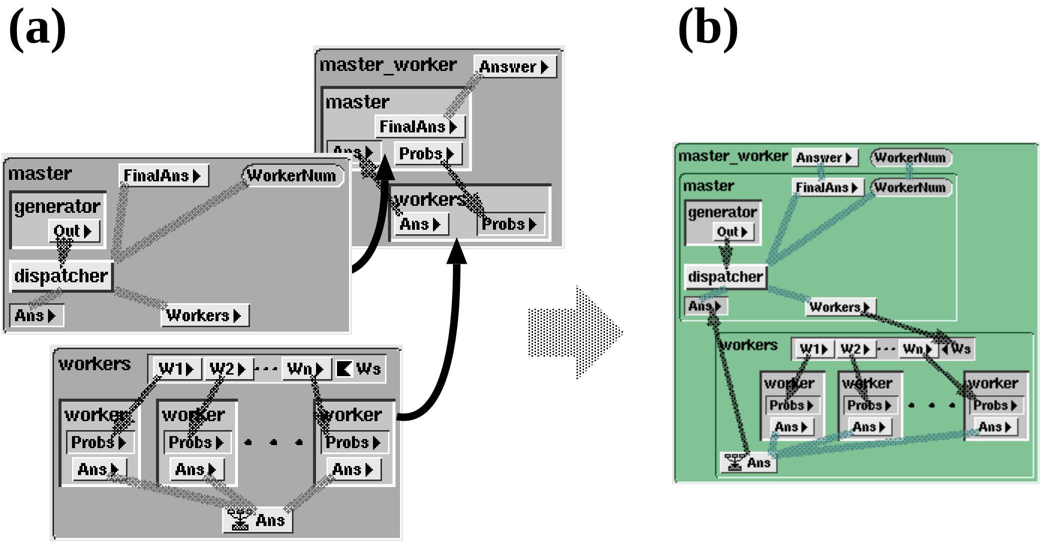

図 4. パターンの階層的な定義

セスが代入される。実行時に生成される worker プロセスの数は、入力用マップポート (図 4の場合、workers パターンの Ws ポート) に入力される要素 (この場合、W$_1$、W$_2$、... W$_n$) の個数で決定される。

2.3　パターンによるトポロジー、およびプロトコルの再利用

パターンを用いることにより、ネットワークトポロジー、およびプロトコルの再利用をはかることができる。例えば、図 4で定義した master_worker パターンは generator、および worker ホールそれぞれにさまざまな問題に応じたプロセスを代入することによって、その問題を負荷分散して解くプロセスを定義することができる。ここでは例として、master_worker パターンを用いて定義した、nqueens 問題を負荷分散して解くプロセス nqueensByMasterWorker を図 5に示す。このプロセスは master_worker パターン中の generator ホールに nqueensGenerator を、worker ホールに nqueensWorker を代入し、さらに幾つかのポートを追加して定義したものである。なお、図 5左上に示されているアイコンは nqueensByMasterWorker プロセスのインタフェース定義である。

また、パターンによってプロトコルもある程度再利用することができる。例えば、図 4中の dispatcher プロセスは master と workers の間のプロトコルの一部を実現するものと考えられる。したがってこの master_worker パターンのようにあらかじめ dispatcher を埋め込んだパターンを用意しておけば、そのパターンを利用することによってプロトコルも再利用できる[4]。

---

[4] さらに dispatcher プロセスを置き換えることによって、master_worker パターンの挙動を変えることができる。例えば、全解探索用の dispatcher を、1 つだけ解を求める dispatcher で置き換える、などが可能である。

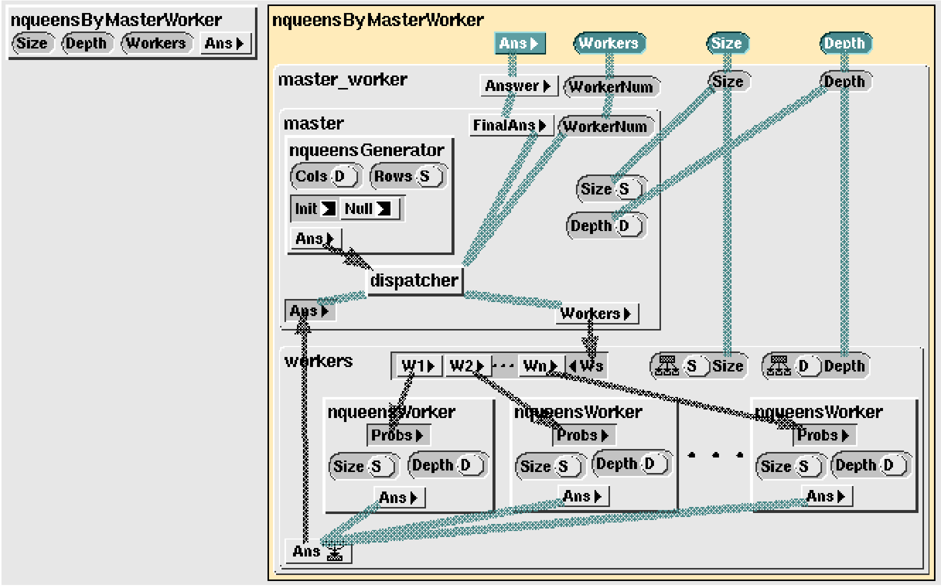

図 5. master_worker パターンを用いて定義した nqueens 問題を解くプロセス

3　プログラム実行の可視化

KLIEG 環境は KLIEG プログラムの実行時の挙動を可視化するトレーサを提供する。その可視化はビジュアルプログラムとして入力された図形の形状を元に行い、プログラマにとって把握しやすい表示を目指す。KLIEG ではネットワークトポロジーを基本的にパターンで定義するため、トレーサの表示は主にそのパターンの形状に基づく。

ここで、KLIEG のようなデータフロー型のプログラムの実行を可視化する場合、可視化の対象としてプロセスネットワーク、プロセス内データ、およびプロセス間で通信されるデータの 3 つが挙げられる。並列プログラムが扱うデータは膨大になる傾向があるため、これらを面積の限られた画面内に表示するにはスケーラビリティの問題を解決する必要がある。以下 3.1節では可視化の方針を示すとともに表示に関する問題を統一的に扱う枠組みを提案する。さらに 3.2節ではその枠組みを用いてネットワーク表示を行う KLIEG トレーサについて述べる。

3.1　プログラム実行の可視化方針

本節ではプログラムの実行とともに生成されるネットワーク、およびデータを、図形として入力されたプログラムの定義に基づいて統一的に構造化する方法について述べる。これにより、その構造を単位としたズーミングによる表示調整機能を提供することが可能になる。

3.1.1　パターン定義に基づく構造化

プロセスネットワークの構造化　KLIEG プログラムの実行とともに、プロセスネットワークが階層的に生成されてゆく。例えば図 6(a) のように、main プロセスが A、

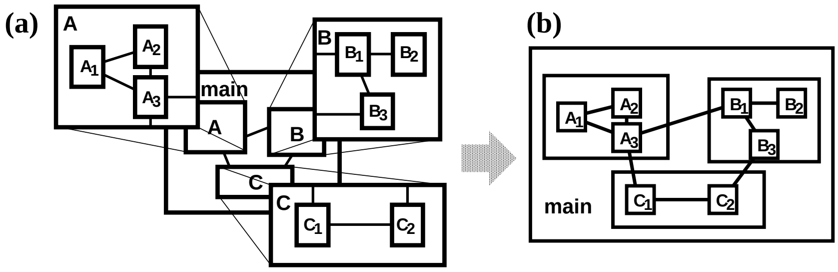

図 **6.** プロセスネットワークのパターン定義に基づく階層構造化

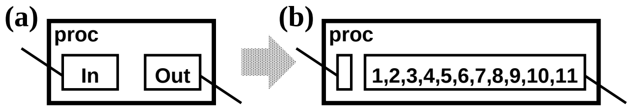

図 **7.** ズーミングによる通信路内のデータ表示

B、C の 3 つの子プロセスから構成されており、さらに A、B、C それぞれが内部にネットワークを持っている場合、プログラムが実行されると main プロセスからは図 6(b) のようなネットワークが生成される。この階層構造をそのままネットワーク表示でも用いる。兄弟プロセス同士の相対位置関係もプロセス定義時の関係を保つようにする。

ポート、およびポートに与えられるデータの構造化 ポート、およびポートに与えられるデータに対してもビジュアルプログラムに基づき構造を与える。KLIEG プログラムでは図 2の naturals に定義されている Out ポートのように、プロセスにポートが定義される。構造化の際にはこれらのポートをプロセスの子ノードと見なす。さらにプログラム実行時にポートに与えられたデータをポートの子ノードとする。

このようなビジュアルプログラムの形状に基づく構造化によって、得られるネットワークの形状はプログラマがプログラムに対して持つイメージに則したものになり把握しやすくなる。さらに KLIEG においてプロセス内のデータはそのプロセスに定義されたシングルトンポートに入出力されるデータとして表現される。またプロセス間通信路に流れるデータはストリームポートから入出力される。したがって、上のような構造化によって、プロセスネットワーク、プロセス内データ、およびプロセス間を流れるデータに統一的な構造を与えることができる。

3.1.2 ズーミングによる表示

以上のようにして構造化したプロセスネットワーク、およびデータを表示し、その構造を単位として一部を拡大・縮小する機能を提供するために、ズーミングに基づく表示アルゴリズムを用いる。これによってユーザはプロセスネットワーク中の任意のサブネットワークを拡大表示することができる。さらにプロセスに定義されているポートに対して拡大操作を与えることにより、プロセス内のデータ、通信路内のデータを参照することも可能になる。例えば図 7のように proc プロセスの Out ポー

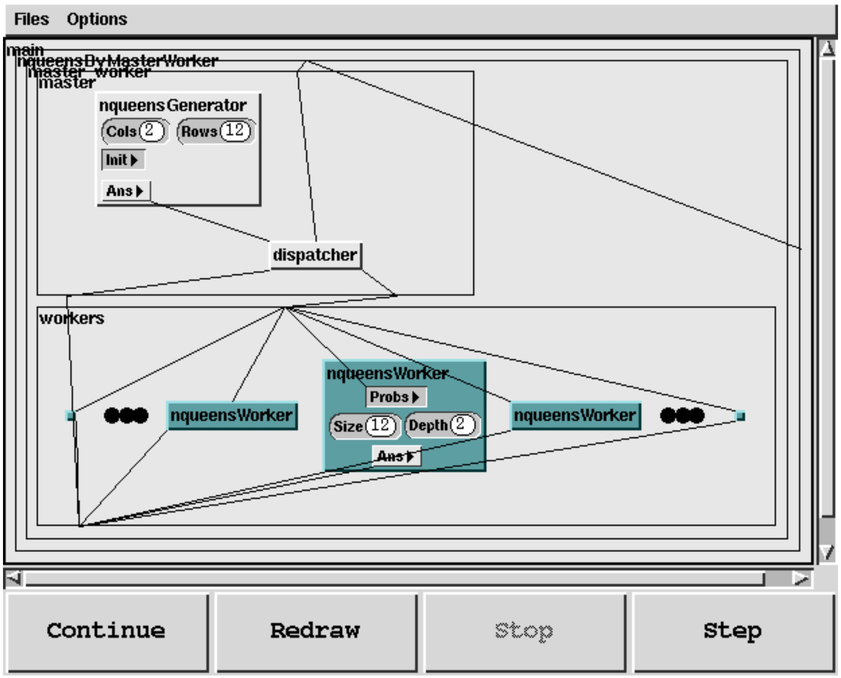

図 8. トレーサのプロセスネットワーク表示例

トから通信路に出力されたデータは、Out ポートを拡大してゆくことによって参照できる。

ここで、表示には以下の条件を満たす必要がある。

1. 詳細に表示すべき部分をユーザが容易に指定できる
2. 生成されたプロセス、データすべてを参照できる

fisheye view[5] と呼ばれる視覚化技術のひとつである Continuous Zoom アルゴリズム [1] はこの条件を満たし、図 6(b) に示されるような長方形のノードから階層的に構成されるネットワークを扱うアルゴリズムとして有効である。今回はこの Continuous Zoom アルゴリズムを用いて階層構造化されたプロセスネットワークの表示を行った。

3.2 トレーサの表示例

KLIEG トレーサの表示例を図 8に示す。これは、図 5で定義した nqueensByMasterWorker プロセスを含む KLIEG プログラムの実行過程を表示している。このように、トレーサはプログラムの実行とともに生成されるプロセスネットワークを、プログラムの形状 (図 5) を保持し、一画面内に収めて表示する。なお、ネットワークが生成され変化する過程はアニメーション表示される。

図 8のプロセスネットワーク表示では、図 4の workers パターンから生成されたプロセス列が省略表示されている。トレーサは、シーケンスパターンを用いることによって生成された同種プロセス列のうち一部のみを表示することによって、プロセス数が増加した場合でもプロセス列を画面内に収めて表示することが可能である。「…」

により省略されているプロセスはマウスにより変更可能である。これによって 3.1.2 節に挙げた 1,2 の条件を満たしつつ省略表示を実現した。

なお、図 8には nqueensWorker プロセスが 5 つ表示されている。中央のプロセスは拡大操作した結果十分な画面空間が割り当てられたため、プロセスに定義されているポート、およびそのポートに入力されたデータまでが表示されている。これに対して他のプロセスは画面空間が十分に割り当てられていないため、ポートが表示されずプロセス名付きの長方形、あるいは小さな正方形として表示されている。これは現在トレーサがプロセス表示に semantic zooming[4] の手法を用いていることよる。十分な表示面積が与えられた場合にはポートまで表示し、そうでない場合にはプロセス名のみの長方形、さらに面積が限られた場合には小さな正方形として表示する。

プロセス内のデータ表示に関しては現在実装中である。プロトタイプ版トレーサではシングルトンポートに入出力されるデータを、図 8の nqueensWorker プロセスの Size ポートのように、プロセス内のポート部分に表示する。ただしポートに対する拡大・縮小操作は実現していない。

4 関連研究

これまでにもデータフロー図を直接プログラムとして扱える並列ビジュアルプログラミングシステムとして HeNCE[2]、CODE 2.0[8]、および Meander[10] などが開発されてきた。しかし、これらのシステムではひとたび定義したプロセス、およびネットワークをそのまま再利用することは可能であるが、KLIEG のように未定義のネットワークをプログラムの基本構成要素として扱う機構は提供していない。

ネットワークトポロジーの再利用機構を提供する並列ビジュアルプログラミング環境としては VISTA[9]、および Software Architect's Assistant[7] が挙げられる。

VISTA[9] では processor を単位としてプログラムを作成する。processor には、内部に processor のネットワークを定義することができ、階層的な processor 定義が可能である。public processor と呼ばれる processor には別に定義した processor を実行時に代入することが可能である。これによってネットワークトポロジーを再利用できる。しかし、動的にトポロジーが定まるネットワークを図式表現することは難しい。これに対し KLIEG では、実行時にプロセスを取り替えることはできないが、シーケンスパターンによって動的なネットワークトポロジーを静的な図で表現し、そのトポロジーを簡単に再利用できるインタフェースを提供している。

Software Architect's Assistant[7] は Regis と呼ばれるテキスト言語に基づいたビジュアルプログラミング環境である。Regis では generic component を用いてネットワークトポロジーの再利用を行うことができるが、Software Architect's Assistant では GUI による再利用機構を提供していない。KLIEG ではエディタが提供する drag&drop のインタフェースによってこれをグラフィカルに行うことができる。

ビジュアルプログラムを元にプログラム実行の可視化を行うシステムとしては Pictorial Janus[6]、VIPR[3]、および PP[11] が挙げられる。これらのシステムではプログラムの挙動を、データの流れを含めて細部まで図式表現する。そのため規模の大きなプログラムの実行過程表示は表示が複雑になるという問題点がある。これに対し、KLIEG トレーサでは図 8のようにプロセスネットワークの概観を表示し、必要の応じてその任意の部分をズーム機能によって参照できるようなインタフェースを

提供することによって、大規模になり得る並列プログラムの実行を大局的に表示することを可能としている。

5 まとめと今後の課題

本稿ではプロセスネットワークパターンに基づく並列ビジュアルプログラミング環境 KLIEG について述べた。KLIEG ではパターンに基づいたプログラミングを支援するインタフェースを提供し、ネットワークトポロジー、およびプロトコルの再利用性を向上することができた。さらに、データフローに基づくビジュアル言語の実行を統一的に可視化する枠組みについて述べ、この枠組みに基づいて KLIEG プログラムの実行を可視化するトレーサについて述べた。

今後の課題としては、さまざまなメタパターンを提供し多くの形状のネットワークの設計を支援するとともに、木構造、メッシュ構造などさまざまな形状のネットワークを効率よく表示することが考えられる。

参考文献

[1] 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.

[2] A. Beguelin, J. J. Dongarra, G. A. Geist, R.Manchek, and V. S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. In *Proceedings of Supercomputing 91*, pp. 435–444, 1991.

[3] Wayne Citrin, Michael Doherty, and Benjamin Zorn. The Design of a Completely Visual OOP Language. In *Visual Object-Oriented Programming: Concepts and Environments*, chapter 4, pp. 67–93. Manning Publications Co., 1995.

[4] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. SemNet: Three-Dimensional Graphic Representation of Large Knowledge Bases. In R. Guindon, editor, *Cognitive Science And Its Applications For Human-Computer Interaction*. Lawrence Erlbaum Associates, 1988.

[5] 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.

[6] Kenneth M. Kahn and Vijay A. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *Proc. 1990 IEEE Workshop Visual Languages*, October 1990.

[7] Keng Ng, Jeff Kramer, Jeff Magee, and Naranker Dulay. A Visual Approach to Distributed Programming. In *Tools and Environments for Parallel and Distributed Systems*, chapter 1, pp. 7–31. Kluwer Academic Publishers, February 1996.

[8] P.Newton and J.C.Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[9] Stefan Schiffer and Joachim Hans Fröhlich. Visual Programing and Software Engineering with Vista. In *Visual Object-Oriented Programming: Concepts and Environments*, chapter 10, pp. 199–227. Manning Publications Co., 1995.

[10] Guido Wirtz. Modularization and Process Replication in a Visual Parallel Programming Language. In *Proc. 1994 IEEE Symposium Visual Languages*, pp. 72–79, 1994.

[11] 田中二郎. 並列論理型言語 GHC のビジュアル化の試み. 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, 近代科学社, 1994.

[12] 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 ・電子情報通信学会技術研究報告 SS95-46, March 1996.