マルチウインドウデバッガ HyperDEBU における細粒度高並列プログラムの実行のデータフローの視覚化

舘村純一[†] 小池汎平[†] 田中英彦[†]

細粒度高並列プログラムには実行の流れが多数存在する．これをデバッグするにはまず実行状況の把握が重要であり，実行の視覚化手法が問題となる．視覚化をデバッグに応用する場合には，状況に応じたユーザの観点を反映させる必要がある．我々は，並列論理型言語 Fleng を対象とするデバッガ HyperDEBU を開発している．HyperDEBU は，ユーザの意図に応じたコントロール／データフローの視覚化機能を持ち，プログラムの視覚的な観察・操作による効果的なデバッギングを可能にする．HyperDEBU では，ブレークポイントをユーザが自分の意図・観点を伝えるものとして位置付けており，ここから得られる情報を実行の視覚化に用いる．HyperDEBU の機能のうち，本論文では主にデータフローの視覚化の方式と機能を述べる．データフローをグローバルに観察するため，HyperDEBU ではストリーム通信に着目して実行の視覚化を行う．我々は，このような視覚化を実現するのに必要な，(1)ストリーム通信を表現する視覚化要素，(2)ユーザの意図を採り入れた視覚化対象の選択手法，(3)視覚化データの画面上での配置手法について考察し，これに基づいてデータフローの視覚化機能を HyperDEBU に実装した．また，この機能の有効性を示すため適用例を提示し，バグを効果的に発見することが可能であることを明らかにした．

Visualizing Data Flows of Fine-grained Highly Parallel Programs on a Multi-window Debugger HyperDEBU

Jun'ichi Tatemura,[†] Hanpei Koike[†] and Hidehiko Tanaka[†]

A fine-grained highly parallel program has many threads of execution. The first task to debug it is comprehending the situation of the execution. For this task, it is important to visualize the execution. Our debugger HyperDEBU for a parallel logic programming language Fleng visualizes control/data flows of execution of a Fleng program according as a user's intention. Breakpoints are introduced as information which represents a user's intention or points of view. HyperDEBU uses this information to visualize execution of a program. HyperDEBU enables efficient debugging by its visual examining/manipulating facilities. In this paper, we describe mainly the methods and facilities of its visualization of data flow. HyperDEBU visualizes stream-based communication which makes global dataflow of the program. To realize this function, we introduced (1) a set of visualized objects to represent streams, (2) a method to decide visualized objects from program code and user's intention, and (3) a method to arrange visualized objects on a display. Then we installed this function for dataflow visualization on HyperDEBU. Applying the debugger to examples, we demonstrate that this function is useful to find a bug efficiently.

1. はじめに

予期せぬ動作をするプログラムをデバッグする場合には，まず実行の様子を把握することが必要である．特に細粒度で高並列なプログラムにおいては実行の流れが多数あるので，どのでどのようなことが起きているのか，プログラムの実行の巨視的な状態をまず理解することがより重要な課題になる．

† 東京大学工学部電気工学科
Department of Electrical Engineering, Faculty of Engineering, The University of Tokyo

このためには，実行情報を抽象化したグローバルな視野が必要となり，デバッガが実行情報をユーザにいかに見せるかといった，プログラム実行の視覚化手法の問題を解決しなければならない．

プログラムの実行の視覚化として研究されているものには，(1)ビジュアルデバッガ，(2)アルゴリズム・アニメーションがある．従来の逐次言語用のビジュアルデバッガとしては，Prolog のデバッガ PROEDIT 2[3] などが挙げられる．これらのビジュアルデバッガはプログラム言語レベルの抽象度の図形を用いるが，細粒度で高並列なプログラムでは大規模で複

雑になり，理解が困難である．一方アルゴリズム・アニメーションとしては，Balsa[4]，ESP のプログラム可視化システム[5]などがあげられる．アルゴリズムアニメーションは，アルゴリズムの理解，プログラミングの教育，仕様／設計の確認などに用いられ，プログラムの仕様を描画用プログラムとして与え対象プログラムにプローブを埋め込んで動作させるなどの方法により，設計・アルゴリズムレベルの抽象度の図形でプログラムの動作を表現する．しかし，これをデバッグに利用するには，(1)仕様記述の手間がかかり，仕様中にもバグが存在しうる，(2)予期しない動作の視覚化が必要である，(3)バグを絞り込んでその位置に到達するにはより低レベルなビューも必要であるなどの理由で適さない．

アルゴリズム・アニメーションのような高レベルの視覚化はソース以外の情報なしでは実現が難しい．しかもデバッグ時には，同じプログラムでもどの部分をどのように見たいかというユーザの意図が状況によって変化するので，ユーザの主観を反映させて「見たいものを見たいように見せる」ことが重要となる．我々のとった視覚化の方針は，ユーザの意図を反映した「付加的な知識」を与え，これを利用して視覚化を行い，プログラム全部について完全な知識を与えなくても，知識を与えない部分は低レベルな視覚化でサポートし，知識の与え方に応じて高レベルなデバッグを可能にする．

我々は，並列論理型言語の一種である Committed-Choice 型言語 (CCL) を対象とするデバッガ Hyper-DEBU を開発している[2]．HyperDEBU は，実行状況の把握を支援するために CCL プログラムの実行の視覚化を行う．ユーザの意図に応じてコントロールフローとデータフローを視覚化する機能を実装し，視覚化されたプログラムの観察・操作を行うことによる効果的なデバッギングを可能にした．

本論文では，HyperDEBU のこのような視覚化機能について述べる．データフロー，コントロールフローの視覚化のうち，ここでは主に，文献 2) の版に対して新たに導入されたデータフローの視覚化について，その方式と実装された機能を述べる．

2. Committed-Choice 型言語 Fleng

Concurrent Prolog や GHC などの Committed-Choice 型言語 (CCL) は論理型プログラミングを並列に実行するためガードの概念を導入して通信・同期を記述できるように制御機能を強化した並列論理型言語である．Fleng[1] も CCL の一つであり，他の CCL に較べてその言語仕様が簡潔になっていることが特徴である．Fleng はガードゴールを持たず，ヘッドのみがガードの働きをする．よって，ヘッドユニフィケーションだけで定義筋がコミットされる．

Fleng プログラムは次のような定義節の集合である．

$H :- B_1, \cdots, B_n.\ (n \geq 0)$

:- の左側をヘッド部，右側をボディ部と呼び，B_i をボディゴールと呼ぶ．

Fleng プログラムの実行は，ゴールの集合の書き換えによって進められる．各ゴールについてパターンマッチが成功するようなヘッド部を持つ定義節が一つ選ばれ，これに基づいて新しいゴールに書き直される．この書き換え操作をリダクション，マッチング操作をユニフィケーションと呼ぶ．CCL の場合，ユニフィケーションは，二つの種類に分けられる．一つはヘッド部で行われるガードつきユニフィケーションで，ゴール側からのデータを待つ時に用いられる．もう一つはボディ部で行われるアクティブユニフィケーションで，ゴールの持つデータに値を代入する．このように，Fleng プログラムは，ゴールリダクションによる制御依存関係と，ガードつき／アクティブユニフィケーションによるデータ依存関係を定義節という形で記述するものである．

Fleng などの CCL は，個々のゴールが並列実行される細粒度並列言語であり，いくつかのプロセスが静的に存在してデータを介して相互に作用するのではなく，ゴールが動的に生成・消滅していく．このために，ユーザがゴールの実行を観察したり操作したりするのは困難である．

3. マルチウインドウデバッガ HyperDEBU

我々は，Committed-Choice 型言語 Fleng のデバッガとして，多次元的インタフェースを用いたマルチウインドウデバッガ HyperDEBU を開発した[2]．このデバッガは，制御・データの流れが形成する複雑なグラフ構造を観察・操作するための多様な視野としてウインドウを提供する．ユーザは，このウインドウ上に表現されたプログラムの実行情報を構成するリングをたどることによってさらに希望するウインドウを開くことが可能である．

CCL において一つのゴールまたはシーケンシャル

なゴールの列をプロセスと見る見方があるが，HyperDEBU では，ある一つのゴールから生成されたゴールの集合を一つのプロセスとして表現する．これによって，プログラムの実行を任意の抽象度で階層的にとらえられ，高並列で大規模なプログラムにも対応できる．

HyperDEBU は，(1)プログラムの実行をグローバルに観察操作するトップレベルウインドウ，(2)任意のプロセスに割り当てられるプロセスウインドウ，(3)構造データを観察するストラクチャウインドウなどから構成されており，以下にあげる各機能が協調してユーザのバグ探索を支援する．

1. **多様な視野を用いたバグの絞り込み：**
   HyperDEBU は，グローバルな視野からよりローカルな視野までをユーザに提供する．プロセスウインドウは，トップレベルウインドウに表示されている各プロセスから開くことができ，多様な側面からプロセスを観察・操作する．ここからさらにサブプロセスを別のウインドウとして開くことで，効果的なバグの絞り込みを行える．
2. **プログラム実行の視覚化：**
   トップレベルウインドウ上でグローバルなビューとして実現され，実行状況の把握を支援することで，バグの絞り込みを効率化している．
3. **ブレークポイント：**
   HyperDEBU では，「ブレークポイント」を拡張して考え，デバッガがユーザから実行前にあらかじめ与えられた知識ととらえる．この情報はプログラムの実行制御・実行の視覚化などに活用される．
4. **プログラム・コードのブラウジング：**
   ブレークポイントの設定時などにおいて静的情報の把握を支援する．

4. HyperDEBU における視覚化手法

4.1 視覚化の基本的アプローチ

高並列プログラムの視覚化は，多数の制御の流れとデータの流れを扱う必要がある．CCL においては，前者にゴールリダクションが，後者にガードとユニフィケーションが対応する．HyperDEBU は，それぞれの履歴情報をプロセスウインドウで詳しく観察できるが，これだけでは実行状況の巨視的な把握には繁雑過ぎる．そこで，それぞれについてユーザの主観を反映させた抽象化を行いプログラムの挙動を視覚化する．

このとき，コントロールフロー，データフローそれぞれについて，

- 何を視覚化するか
- どのように指定するか
- どのように表示するか

といった点が課題となる．本論文では，これらの点について，データフローに関する手法を中心に述べていく．

また，表示に関する CCL 特有の問題として，表示すべきデータ・ゴールが動的に生成される点がある．このために実行時に動的に画面上の配置を行う手法が必要となる．

4.2 ブレークポイントによる指定

多数のコントロールフローとデータフローからなる並列プログラムのデバッグを行う場合，逐次プログラムのようにブレークポイントで停止して実行状態を見るといった手法は適用できない．HyperDEBU では，デバッグにおける「ブレークポイント」を拡張して考え，「プログラム実行前にユーザがデバッガに与えた知識」ととらえる．デバッガは，この情報をプログラムの実行制御などに活用することができる．実行の視覚化も，このブレークポイントによって指定される．

ブレークポイントは「場所」と「処理」の組で指定される．ブレークポイントの場所の指定には，(1)述語名による指定，(2)述語の各定義節ごとの指定，(3)定義節中のボディゴールごとの指定，(4)ゴール中の引数レベルの指定といった各レベルがある．

ブレークポイントによって指定できる「処理」としては，現在以下のものがあり，各「場所」についてそれぞれ複数指定することができる．

- ゴール実行の停止 (pause)：
  該当するゴールのみが実行を停止する．
- 実行履歴の制御 (notree)：
  該当するゴールから先の実行履歴歴を記録しない．
- プロセスの切り分け (process)：
  該当するゴールがプロセスとして視覚化される．
- データレベルの視覚化 (stream)：
  該当するデータをストリーム通信として視覚化する．

このうちの “process” と “stream” がプログラムの視覚化に関するブレークポイントである．

4.3 コントロールフロー

HyperDEBU では，ある一つのゴールから生成されたゴールの集合を一つのプロセスとして表現する．

トップレベルウインドウは，特定のゴールに関するプロセスのみを表示することで制御の流れに関するグローバルな視野を提供する．**図1**は HyperDEBU におけるコントロールフローの表示例である．画面中央の矩形の入れ子構造がコントロールフローを表現している．

各プロセスは，ウインドウの中に表示された矩形で表現される．そのプロセスの内部のゴールに関するプロセス（サブプロセス）は，矩形の入れ子によって表現され，プロセスの状態は矩形の色によって表現されている．マウスカーソルが矩形の中に入ると，その矩形に対応するゴールがゴール表示ウインドウに表示されるので，各プロセスの識別と，プロセス間の大まかな関係がわかる．これらの表示は，実行状態を反映して動的に変更され，プロセスの生成や状態変化，引数のデータの変化が把握できる．また，矩形からプロセスウインドウをとり出して，より詳しい観察・操作ができる．

矩形として表示されるプロセスは，ユーザがブレークポイントで “process” と指定したゴールに関するプロセスのみであり，他は内部のゴールとして抽象化される．これによって制御の流れの概略が観察できる．

4.4 データフロー

データの流れをグローバルに観察するため，Hyper-DEBU では視覚化されたプロセス間に存在するストリーム通信に着目して実行の視覚化を行う．

ストリーム通信とは，CCLにおける重要なプログラミング手法であり，プロセス間の継続的な通信を可能にする．このストリーム通信は共有変数を用いて次のように行われる．

1. 一つのゴールが何らかの構造データを変数に代入する（ストリーム出力）．
2. 他のゴールはその値の確定すると，それを読んで処理を行う（ストリーム入力）．
3. 構造データの中には新たな変数が含まれており，これを介して次の通信が行われる．

プログラム中にはこのようなデータフローが多数存在するが，HyperDEBU では，ユーザがまず注目すべき大域的なデータフローをなすようなストリームのみを視覚化することにより，プログラムの実行状況の把握を支援する．このとき，ユーザが何に注目したいかをデータフローのためのブレークポイント “stream” で指定する．

次章では，このストリーム通信をどのように視覚化してデバッグに用いるかを述べる．

5. データフローの視覚化

この章では，(1)ストリーム通信の様子を表現するのに何をどのような形で視覚化するか，(2)視覚化すべきものをユーザの意図を採り入れながらどのように選択するか，(3)視覚化されたデータを画面上にどのように配置するかについて述べ，これに基づき実現された視覚化機能の説明を行う．

5.1 視覚化対象

ストリーム通信の様子を表現するために，ストリームが生成される様子，それが分配される様子，データの入出力が行われる様子を**図2**のように視覚化する．

- 図 2-(1) はストリームが生成された時に画面に視覚化される図形の様子を表したものである．スト

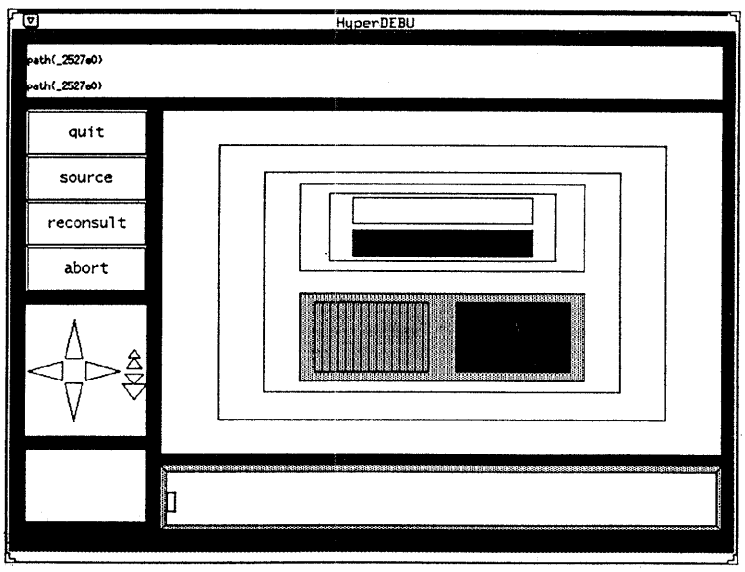


図 1 コントロールフローの視覚化
Fig. 1 Visualizing control flows.

リームが生成されるのは，定義節のボディゴールに新たな共有変数が生まれた時の状態である．

- 図 2-(2) はストリームが分配される様子を示している．これは，図にあるような定義節によってゴールcがリダクションされ，変数Sを参照するゴールが増えた時の状態である．
- 図 2-(3) はストリームに出力が行われた様子を示している．これは，変数Sに構造データが代入された時である．このデータがゴール間のストリームにつながれて表現されている．
- 図 2-(4) はストリームからの入力が行われた様子を示している．定義節のヘッド部に書かれたような構造データを待ってリダクションが行われた時の状態である．このデータがガードとして表現されている．

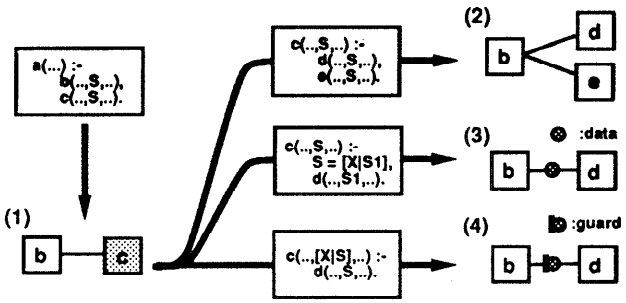


図 2 ストリームの生成・分配・入出力
Fig. 2 Creation, distribution, input and output of stream.

5.2 視覚化対象の選択

着目するストリームを視覚化するには，ストリームに用いられる変数について，図2にあるような各動作にあたる部分をプログラム中で指定する必要があるが，このような各部分を指定するのは繁雑である．ユーザは着目点を示すのに必要最低限の情報のみを与え，デバッガはこれに基づき必要十分な実行情報を提示することが望まれる．このためには次の点を明確にしなければならない．

1. ユーザの見たいものは何か．
2. これを伝えるのにユーザがどのように指定するか．
3. デバッガが何をどのように見せるか．

ユーザの着目点 ユーザがあるストリームに着目する場合，それをどの観点から見るかが問題である．ユーザは，あるプロセスに着目し，そのストリームを通じた他との通信において

- ストリーム通信の相手，
- ストリームに対する操作

に関する情報を要望する．

ユーザの指定 どのストリームをどのプロセスから見るかを伝えるかには，プロセスとその引数を指定すればよい．ユーザは，着目するプロセスを定義している述語の引数をブレークポイントで指定して，これを着目するストリームとしてデバッガに知らせる．

デバッガの視覚化対象 ストリーム通信の様子を表すには，引数として指定したストリーム変数に与えられるデータだけでなく，その中のストリーム変数に関しても視覚化を行うことで，ストリームが生成してから，着目するプロセスのストリーム動作までの一連のストリームを視覚化する必要がある．ただし，

- 変数を共有した相手
- ストリーム動作を行うためのサブルーチン

などの視覚化するストリームを共有するゴールに関して，そのゴールからリダクションを繰り返して生まれるサブゴールがどれもストリームのブレークポイントを指定されていない場合，リダクションされてもサブゴールの中に着目するストリーム動作がないので，リダクション時のストリーム動作を見せずにゴールのまま表示をしてその内部を抽象化する．その内部が見たい場合には，その部分にもブレークポイントを指定すればよい．

これによって着目するプロセスについてはストリーム通信の相手とストリームに対する操作を知る必要十分な情報を視覚化できる．

ユーザの指定したブレークポイントからこれらの視覚化対象を特定するため，デバッガ側はプログラムを解析して各述語について次の情報を得る．

- プログラムコード中に現れる変数のうち，どれが視覚化されるストリームに該当するのか．
- リダクションによるデータ操作を視覚化するか，あるいはゴールの内部を抽象化するか．

これらに基づきデバッガはストリームの各動作を視覚化する．

5.3 ユーザの指定方法

5.3.1 何を指定するか

継続して行われるストリーム動作を視覚化するには，変数に与えられた構造データの中でどの部分が次の通信に用いられる新しいストリーム変数であるかが

問題になる．ユーザは述語の引数としてストリーム変数の位置を指定する．この変数に与えられるデータ中の新しいストリーム変数の位置は，ユーザがどのストリームの流れに着目するかに依存するので，ストリームにどのようなデータが与えられ，そのどこにストリーム変数があるかの情報が必要である．

これを指示するため，ストリーム変数に与えられるデータの集合をストリームの型として以下のように表す．

$$S = s_1(S) + s_2(S) + \cdots + s_m(S) + t_1 + \cdots + t_n$$

$s_i(S)$ は，ストリーム変数 S と任意の変数を含むデータ(項)で，変数部分に任意のデータを代入したデータの集合を表す．ストリーム変数の部分には，その型のストリームのデータが代入でき，ストリーム通信の継続に用いられる．t_i は，ストリーム変数 S を含まずストリームの終端に用いられる．演算子＋は集合の和を表す．

多くのプログラムの場合，ストリームとして用いられるのは一列のリストである．これは次のように表せる．

$$S = [X|S] + [\,]$$

これは，ストリーム変数で表されるデータがリスト [_|_] か，[ ] かであり，リストの場合その CDR 部が同じ型のストリームであることを表す．

この情報により，ストリームに与えられたデータの中での次のストリーム変数を知ることができる．ストリーム変数 S のデータの中に変数 $S1$ が存在して，これが次のストリーム変数であるということを next(S, $S1$) と表すと，上記のストリームの型の定義では，例えば次のことがいえる．

next([X|S], S),
next([X, Y|S], S), …

このようなストリームの型指定を導入すれば，例えば以下のような木構造をしたデータをストリームとみなすなどの一般的なストリームにも対応できる．

$$S = \mathrm{node}(S, X, S) + \mathrm{leaf}(X)$$

5.3.2 デバッガでの指定操作

HyperDEBU にはブレークポイントを設定するためのインタフェースが用意されている．ユーザは，これを通して，着目したい部分をブレークポイントとして指定する．

現在のインタフェースの実装では，指定できるストリームの型として，通常用いられるリスト型のストリームのみをサポートしている．多くの場合はこれで十分であるが，より一般的なストリームの型の指定を行うためには，ウインドウ上でインタラクティブかつ容易な型指定ができるようなインタフェースを開発する必要がある．

図 3 は，デバッガで “stream” のブレークポイントを設定している様子である．ソースコードを参照しながら，画面の右半分のウインドウに表示された述語について引数をクリックするとブレークポイントが指定される．ここでは，filter という述語の第 2 引数と第 3 引数に “stream” のブレークポイントをつけている．画面上ではクリックした引数が黒く表示されている．この設定により，視覚化の際にこれらの引数が表示すべきストリームとみなされる．

このように HyperDEBU ではブレークポイントを設定することによって視覚化が行われる．ブレークポイントはユーザがプログラムからの静的な情報をもとに自分の意図を示すものであるから，設定作業の負担を軽減するには，ユーザが静的情報を把握しやすいように支援する機能が必要である．そこで，ソースコー

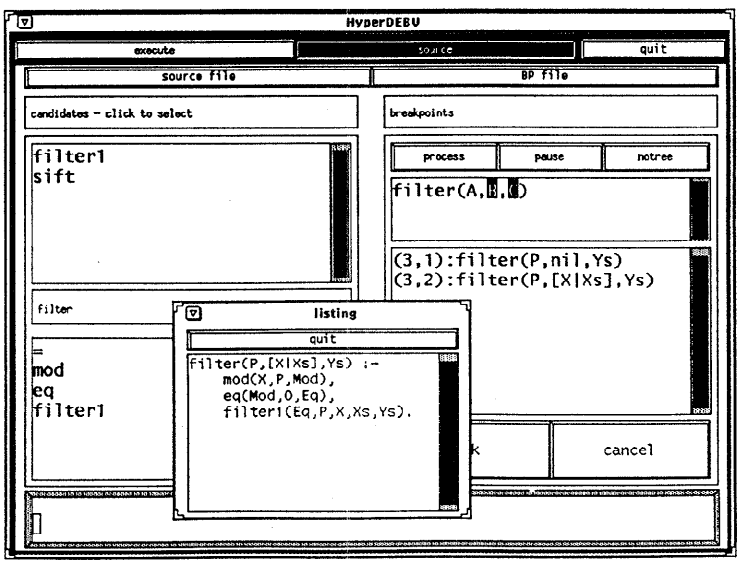


図 3 データフローのブレークポイントの設定
Fig. 3 Setting breakpoints to visualize dataflow.

ドのブラウジング機能を用意することにより，ブレークポイントの設定を支援した．図3で表示されているウインドウはこの機能の一部である．現在実装されている機能は，述語の呼びだし関係のトレースや，述語名の検索・補完機能などであり，機能をより強力にした改良版の開発も進めている．

5.4 視覚化データの解析手法

デバッガに実装されたプログラムの静的解析機能は，ブレークポイントとして指定されたユーザからの情報と，ソースコードから得られる情報をもとに，何をどのように視覚化するかを決定し，実行時にコードのどの部分でデバッガが何をするかを示すデータを生成する．これは，次の段階により行われる．

1. ブレークポイントをもとに，プログラム中にストリーム変数を見つけ出す．
2. ストリーム変数を持つゴールのうちどれを視覚化するかを決定する．
3. 視覚化される各ストリーム操作に対応した視覚化命令の生成．

視覚化部分の決定規則 まず，プログラム中のどの変数が視覚化すべきストリーム変数であるかを知るには，プログラムに関する知識と，ユーザから与えられる知識から推論を行う．

解析手法の概略説明のため，ここでは以下の術語を用いる．

- arg(G, i)：ゴール G の i 番目の引数．
- var(T, k)：データ T の中に含まれる k 番目の変数．
- breakpoint(A, T)：述語の引数 A に型 T のストリームのブレークポイントが設定されている．
- stream(A, T)：述語の引数 A は型 T のストリームである
- equiv(X, Y)：変数 X と Y が同一である．
- next(S_1, T_1, S_2, T_2)：型 T_1 のストリーム変数 S_1 に与えられたデータの中に，次に用いられるストリーム変数として型 T_2 の S_2 が存在する．

また，定義節のヘッドを H，ボディを B_i とする．

ストリーム変数を発見するための推論規則は次のようになる．

1. 定義節中でストリームと同じ変数として現れる部分はストリームである．

   breakpoint(arg(B_m, i), T)
   → stream(arg(B_m, i), T).

   stream(arg(B_m, i), T), equiv(arg(B_k, j), arg(B_m, i))
   → stream(arg(H, j), T).

   stream(arg(B_m, i), T), equiv(arg(B_k, j), arg(B_m, i))
   → stream(arg(B_k, j), T).

   stream(arg(H, j), T), equiv(arg(H, j), arg(B_m, i))
   → stream(arg(B_m, i), T).

2. ストリーム入力が行われ，次のストリーム変数が存在する．

   stream(arg(H, i), T_1),
   next(arg(H, i), T_1, var(arg(H, i), k), T_2),
   equiv(var(arg(H, i), k), X)
   → stream(X, T_2).

3. ストリーム変数にデータが代入され，その中に次のストリーム変数が存在する．ここで，$U(X, Y)$ は X と Y をユニファイする述語とする．

   stream(arg(U, 1), T_1),
   next(arg(U, 2), T_1, var(arg(U, 2), k), T_2),
   equiv(var(arg(U, 2), k), X)
   → stream(X, T_2).

ゴールGのリダクション時のデータ操作が視覚化されることを visible(G) で表すと，これを推論する規則は次のようになる．

breakpoint(arg(B_m, i), T)
→ visible(B_m).

stream(arg(B_m, i), T), visible(B_m), stream(arg(H, j))
→ visible(H).

以上により視覚化すべきストリーム変数を持つ定義節が明らかになったが，次にこの定義節にそれぞれに関して解析が行われ，図2に示されるようなストリーム変数に対する生成・分配・入出力操作を抽出する．この定義節が選択実行された時にデバッガが実行する視覚化操作を示す命令を，ここで抽出されたストリーム操作に応じて生成する．

静的デバッグとの融合 また，ユーザからの情報は，プログラムの静的デバッグにも役立つと考えられるので，ここで得られる解析結果を用いた静的デバッグ機能を開発している．この機能を用いれば，着目するストリームに関してユーザの意図とソースコードとの矛盾を指摘することでプログラムのバグを発見することができる．視覚化部分の解析にともなうデータフロー解析により，次のようなバグが発見しうる．

- ストリーム変数の型のエラー
  変数への代入や，データの入力待ちにユーザが指定した型のデータ以外のものが用いられている．
- ストリーム変数の参照エラー
  ストリーマ変数が単一参照となっている複数のゴールに共有されていない．
- 入出力モードのエラー
  一つのストリーム変数について入力モードと出力モードが競合している．

現在試作版が実装されており，ユーザがブレークポイントを設定したあとで，このようなバグが発見されるとウインドウが開いて報告する．

解析機能の実装方式 この Fleng プログラム解析機能は，Fleng 自身によって並列実装されている．各述語に対応するプロセスを割り当て，述語の呼びだし関係に基づいてネットワークを構成する．これらが推論で得られた知識をメッセージとして通信し合いながら並列に解析を行う．各述語ごとの知識を各プロセスが分散管理することで，並列度の高い処理が行われる．

5.5 表示手法

データフローとして画面に表示すべきものは，データ，ガード，ゴールといったノードと，これらをつなぐストリームからなるグラフである．このようなグラフを表示する場合，画面上の配置が問題となる．この時にデバッグのための表示として重要な点は以下の2点である．

- ユーザが配置情報を与えなくても自動的に行われる必要がある．視覚化のための手間がかかりすぎてはユーザの負担になるばかりでなく，視覚化を指定するための記述側にもバグが存在する可能性がでてくる．ユーザは何を見たいかという自分の意図だけを簡潔に示せることが望ましい．
- プログラムの実行が進むにつれて動的に配置が変化する．配置が変化するたびに計算の手間がかかりすぎないよう，インクリメンタルな配置手法が望まれる．また，変化前と後とでグラフの自然な対応がとれなければアニメーションには適さない．

このグラフの配置は以下の手法により行われる．

1. **ノードのグループ化**：互いにストリームでつながれたノードを一つのグループとし，これに矩形領域を割り当て，制御の流れの視覚化と同様にして配置する．
2. **グループ内の配置**：グループに割り当てられた矩形領域は，ノードの数だけの矩形に分割され，それぞれの中央に各ノードが配置される．矩形はノードが増えるたびに分割され，各矩形の面積が同じになるように再配置が行われる．あるノードが二つのノードに分割されたときは，そのノードのおかれた矩形領域が生成された時の分割面と垂直にその領域を分割する．**図4**はノードが生成されていく様子を示したものである．図中の a,…,

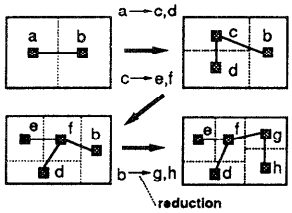


図 4 ゴール・データ・ガードの配置手法
Fig. 4 Method to arrange visualized objects.

```
primes(Max, Ps) :-
        gen(2,Max,Ns), sift(Ns,Ps).

gen(N,Max,Ns) :-
        less_eq(N,Max,LE), gen1(LE,N,Max,Ns).

gen1(true,N,Max,Ns) :-
        Ns = [N|Ns1], add1(N,N1),
        gen(N1,Max,Ns1).          %correct
%       gen(N1,Max,Ns).           %erroneous
gen1(false,N,Max,Ns) :- Ns = [].

sift([P|Xs],Zs) :-
        Zs = [P|Zs1], filter(P,Xs,Ys),
        sift(Ys, Zs1).
sift([], Zs) :- Zs = [].

filter(P,[X|Xs],Ys) :-
        mod(X,P,Mod), eq(Mod,0,Eq),
        filter1(Eq,P,X,Xs,Ys).
filter(P, [], Ys) :- Ys =[].

filter1(true,P,X,Xs,Ys) :-
        filter(P,Xs,Ys).
filter1(false,P,X,Xs,Ys) :-
        Ys = [X|Ys1], filter(P,Xs,Ys1).
```

図 5 視覚化するプログラムの例
Fig. 5 Example program to be visualized.

h はゴール・データ・ガードといった表示すべきノードである．例えば，上二つのグラフは，aがリダクションによってcとdに分割された様子を表している．

データフローとコントロールフローのブレークポイントを合わせて用いることにより，それぞれを融合した視覚化が行われる．プロセスの矩形は，ゴールやデータと同様に配置されストリームの直線で他と結ばれる．プロセス内部に位置付けられるデータフローは矩形内に表示され，矩形を境界にして外部と連結される．

これにより，ストリーム通信をする矩形同士は近くに配置され直線で連結されるので，コントロールフローのみの視覚化よりもプロセス間の関係が把握しやすくなる．また，プロセスとして抽象化された矩形内部のデータフローを別のレイヤとして表示することで，データフローのみの視覚化に比べて表示するグラフの繁雑化を避けられる．

5.6 デバッガへの実装

ここでは，以上で述べた手法を導入して実装されたHyperDEBU のデータフローの視覚化の実際の機能について，具体例と実画面を用いて説明を行う．

データフローの視覚化例 HyperDEBU のトップレベルウインドウにおけるプログラムの実行のデータフローの視覚化例として，まず，**図5**のプログラムの実行を視覚化する．

このプログラムは，素数を生成するプログラムである．gen/3 が自然数の列をストリームとして生成して sift/2 にわたす．sift/2 は，受けとったデータについて，その倍数を取り除くようなフィルタ filter/3を生成し，これを多段につないでいくことにより，通過したデータを素数として結果のストリームに出力する．ここでは，フィルタを通過して素数が生成されるまでのストリームを視覚化する．

一つのストリームを視覚化するのにも，着目点のおき方によってブレークポイントの付け方が違ってくる．ここでは，2通りのブレークポイントの付け方でそれぞれ視覚化したものを**図6**に示す．図では，上下二つのトップレベルウインドウ(1)，(2)が表示されている．それぞれ図5のプログラムを同じストリームについて視覚化しているのであるが，ブレークポイントの付け方が以下のように異なる．

(1) [画面上] filter/3 の第2，3引数にブレークポイントをつける

(2) [画面下] sift/2 の第1引数にブレークポイントをつける

1は filter/3 が，2は sift/3 が，それぞれ何とストリーム通信をして，何を行っているかに着目している．

画面表示とユーザの操作 図6の各トップレベルウインドウの画面中央に表示されている，互いに直線で結ばれた図形が視覚化されたプログラムである．構成要素は以下のとおりである．

- データ：小さな正方形はストリームの出力によって作り出されたデータである．
- ガード：線分と丸からなるノードはガードであり，線分のついている側からデータを受けとっていることを表す．
- ゴール：中に文字の書かれた長方形はゴールであ

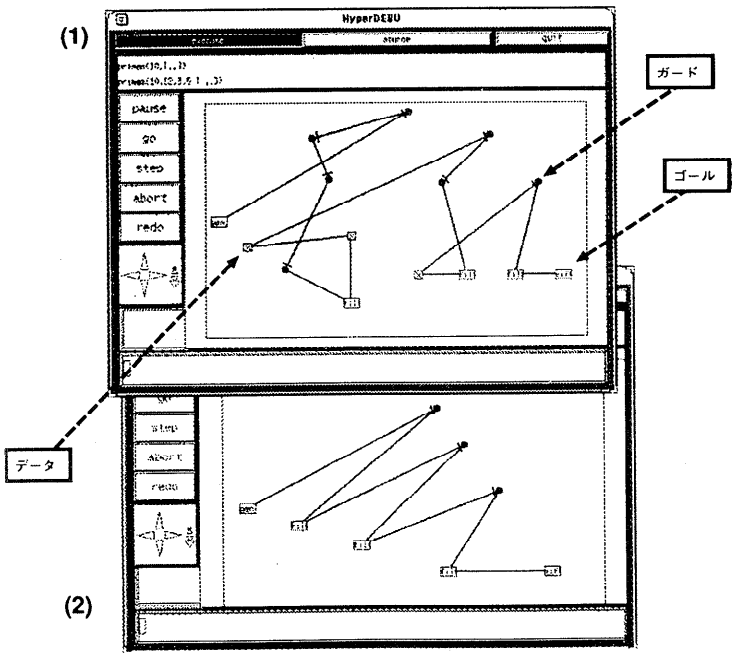


図 6 データフローの視覚化例1
Fig. 6 Example of dataflow visualization 1.

る．

- ストリーム：データ，ガード，ゴールの各ノードをつなぐ線は一つのストリーム変数に対応し，データの流れを表している．

このように視覚化された図形は，プログラムの実行が進むにしたがって変化していき，その実行状況をアニメーションによって表現する．マウスカーソルが各ノードの上に来ると，その内容がトップレベルウインドウの上の部分に表示される．また，これらをクリックすることにより，データをさらに詳しく観察するためのストラクチャウインドウが取り出せる．

着目点による視覚化の違い 図6の二つのトップレベルウインドウのうち，(1)の画面では，左端にあるゴール gen からストリームを表す直線が伸びており，その先にデータを受けとったことを示すガードが filter ゴールの手前まで一列につながっている．その filter ゴールの先には filter が出力したデータがつながっており，次の filter がそれを受けとっていることが分かる．(2)の画面では，filter 自体の動作は視覚化されず，右端にある sift がデータを受けとって，filter を生成する様子が視覚化されている．どちらの画面でも，gen がデータを出力している様子は視覚化されていないが，gen にもブレークポイントをつければその様子も視覚化される．

デバッガのデータフロー解析の効果 デバッガに導入された静的データフロー解析機能は，ブレークポイントをもとにプログラム中にストリーム変数を見つけ出すことにより，ユーザの指定を最低限に押える．ここでは，filter または sift のどちらか1か所を指定するだけで，sift，filter，filter 1，gen，gen 1 に渡るデータフローが解析され，それらのもつストリーム変数が検出される．

また，解析中に発見されるエラーはブレークポイント指定時に報告される．例えば，図5中の gen 1 の定義節中で，%correct と書かれた行を，現在コメント・アウトされている %erroneous と書かれた行と入れ換えると，データフロー解析機能は，参照を失ったストリーム変数 Ns を発見してエラーとして報告する．

コントロールフローの視覚化との融合 次に，コントロールフローのブレークポイントの併用による視覚化例を**図7**に示す．これは，図5のプログラムの実行を gen/3 と filter/3 をプロセスとして視覚化したものである．

図中のノードのうち，大きな矩形がコントロールフローのブレークポイントにより視覚化されたプロセスである．これをマウスで直接操作することにより，プロセスウインドウが取り出せる．このウインドウにより，プロセス内部の詳しい観察が可能となる．図中で灰色の矩形は filter であり，そのデータの入出力の様子が矩形の中に視覚化されている．このように，プロセス内部のデータフローを矩形の中に表示することで，データフローのグラフが階層化され，複雑なデータフローも段階的に把握することが可能となっている．

また，画面の拡大縮小や，プロセスウインドウによる画面の切り出しを可能にすることで，図形が複雑になった場合に対処している．**図8**は，複雑になった視覚化例について，画面を拡大してさらに一部をプロセスウインドウとして切り出した様子である．トップレベルウインドウ上で，切り出された部分は黒い矩形となっている．

実用規模のプログラムの視覚化例 実用規模のプログラムのデータフローを視覚化した例として，Hyper-

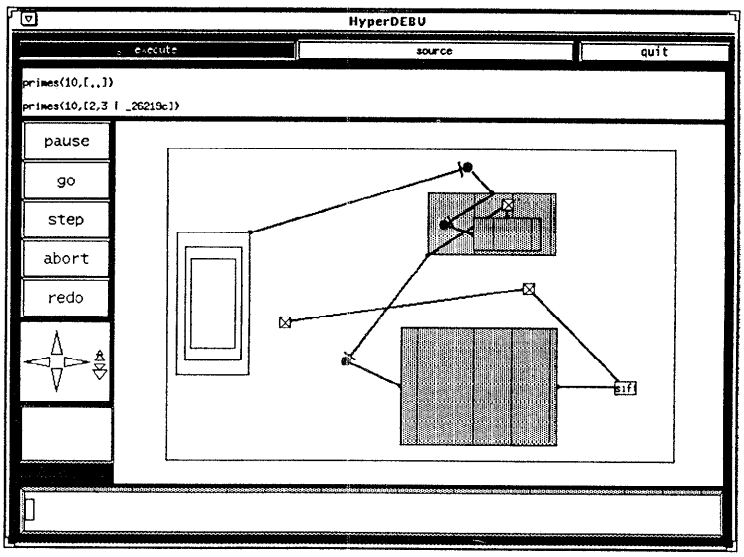


図7 データフローの視覚化例2—コントロールフローの視覚化との融合
Fig. 7 Example of dataflow visualization 2.

DEBU 自身の視覚化を行ったものが図 9 である．デバッグの対象となっている HyperDEBU は，あるプログラムのコントロールフローを，階層化されたプロセスの矩形として表示する．この矩形を表示している HyperDEBU 内部のプロセスについてデバッグする側の HyperDEBU が視覚化を行う．このプロセス自体にも階層関係があり，階層間ではストリーム通信が行われている．ここではこのストリーム通信を視覚化する．

画面上で最も外側の矩形が HyperDEBU 全体であり，その中の矩形が視覚化された「矩形視覚化」プロセスである．その内部に表示された図形のうち，最も右下のゴールが子プロセスを生成するゴールである．これは，対象プログラムに視覚化すべきプロセスが生成されると新たな「矩形視覚化」プロセスを生成し，既存のプロセスに対しては新たな配置情報を送信する．図 9 においては，灰色の矩形が既存のプロセスを，その下の白色の矩形が新たに生成されたプロセスを表現している．旧プロセスの内部では上の階層から配置情報を受信している様子が視覚化されている．新プロセスの内部ではさらに下の階層のプロセスが生成されている．全体の処理が再帰的な階層構造を持った処理であるので，得られた図形もそれに対応する再帰構造を持っていることがわかる．

ブレークポイントとしては，「矩形視覚化プロセス」の初期ゴールに対して process を，「矩形視覚化プロセス」の一つの引数に対して stream を指定した．HyperDEBU の現在の実装では，30 か所の述語の引数がこのストリームに関与しており，これらがブレークポイント設定時に解析される．

6. デバッグ例

HyperDEBU の視覚化機能を用いたデバッグの例として，経路探索問題のプログラム（図 10）をとりあげる．これは，next という述語で表現される有向グラフにおいて，start というノードから goal までの経

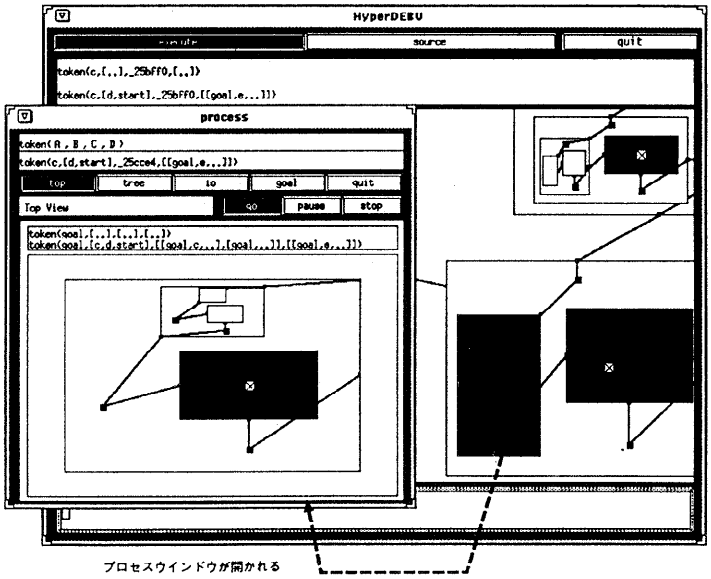


図 8 データフローの視覚化例 3—プロセスウインドウによる切りだし
Fig. 8 Example of dataflow visualization 3.

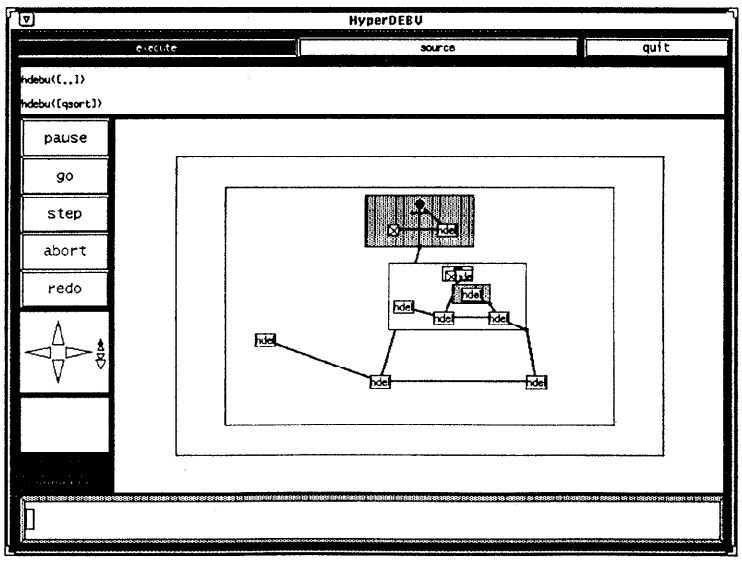


図 9 データフローの視覚化例 4—HyperDEBU 自身の視覚化
Fig. 9 Example of dataflow visualization 4.

路をすべて求めるプログラムである．初期ゴールとして path を投入すると，token というゴールが生成される．token は経路を調べながら，自分の子供を生成することにより並列に探索を行う．各 token は隣同士と共有変数を持っている．解が発見された時は，この変数を用いて解のデータをつなげることにより，一つのリストとしてすべての解が得られる．

しかし，next の定義節の一つは，データを出力すべきところを入力待ちの形に間違って記述されている．このプログラムを動作させると，サスペンドしたままで正しい結果が得られない．

```
path(A) :- token(start,[],A,[]).

token(Node,History,H,T) :-
    eq(Node,goal,F),
    token1(F,Node,History,H,T).
token1(true,Node,History,H,T) :-
    H = [[goal|History]|T].
token1(false,Node,History,H,T) :-
    next(Node,Next),
    checknext(Next,[Node|History],H,T).

checknext([],History,H,T) :- H = T.
checknext([N|Ns],History,H,T) :-
    member(N,History,Result),
    gonext(Result,N,History,H,T1),
    checknext(Ns,History,T1,T).

gonext(true,_,_,H,T) :- H = T.
gonext(false,Node,History,H,T) :-
    token(Node,History,H,T).

next(start,Next) :- Next = [a,d].
next(a,Next) :- Next = [start,b].
next(b,Next) :- Next = [a,c,goal].
next(c,[b,d,goal]).
                        %erroneous
%next(c,Next) :- Next = [b,d,goal].
                        %correct
next(d,Next) :- Next = [start,c,e].
next(e,Next) :- Next = [d,goal].
```

図 10 バグのあるプログラムの例
Fig. 10 Example of erroneous program.

このプログラムをデバッグするための第一段階として，視覚化を行うことにより実行状況を把握する．このため，まずはじめにユーザがプログラムの実行をどのように見たいかをブレークポイントにより指定する．コントロールフローの中心となっているのは解を並列に探索する主体である token である．一方，探索された解を回収するためのリンクが，このプログラムの動作を把握するためにもっとも重要なデータフローといえる．そこで，ブレークポイントとして token に “process” を，token の第 3，第 4 引数に “stream” を指定する．これにより，token，token1，checknext，gonext に渡るデータフローが視覚化される．

このようにしてプログラムを実行した結果が**図 11** である．

図中で，入れ子になっている矩形が token に関するプロセスであり，分裂しながら並列に解を探索している様子を表している．白い色のプロセスは現在動作している状態であり，濃い灰色のプロセスは実行を終了したプロセスである．これに対し，薄い灰色のプロセスは何らかのデータを待って停止しているプロセスである．これらのプロセスは一本のストリームによってつながっており，解が見つかるとそこにデータをつなげる．しかし，薄い灰色のプロセスではストリームにまだゴールがつながったまま停止してしまっている．このゴールについてストラクチャウインドウで観察すると，checknext というゴールがストリーム変数を持ったまま停止して，このため全体として解が得られないということがわかる．

このようにしてプログラムの実行状況を把握することにより，バグの存在位置が限定された．そこで，この薄い灰色の矩形で示されている token プロセスについて，プロセスウインドウを開いてさらに詳しくバグを絞り込んでいく．**図 12** は，プロセスウインドウを開いて詳しいデータフローを観察している様子である．

このウインドウの表示によって，全体のデータの流れを止めている checknext は，next というゴールからのデータを待って止まっていることがわかる．最終的に，next の定義にバグが存在することが突き止められる．

コントロールフローの視覚化と併用してデータフローの視覚化を行うことにより，コントロールフローを表す矩形の相互関係が把握でき，より早く的確に実行の状況をとらえることができた．

この例で視覚化したのは解を回収するための大域的データフローであったが，実際にはローカルなデータフローにバグがあった．大域的なデータフローを視覚化することにより，プログラムの実行状況を把握し，これをもとにバグの存在位置を絞り込んでローカルなデータフローを観察することによりバグの位置を効果的に探索することができた．

7. 諸研究との比較

CCL の実行の視覚化を行った最近の研究例としては，VISTA[6], Pictorial Janus[7], POD[8] があげられる．これらはそれぞれの目的に応じて視覚化の方針が異なっている．

VISTA は，FGHC で記述された探索問題などの高並列な論理型プログラムがどのように並列に実行されたかを知るためのツールであり，パフォーマンスメータの働きを持つ．これはゴールの呼びだし関係でできるツリーを色を用いて表示することによりコントロールフローの概略を示している．このシステムでは，データフローに関しては扱っていない．

Pictorial Janus は，プログラムの実行すべてを完全に視覚化することにより実行の視覚化と視覚的プログラミングの統合を可能にする．このプログラミング環境では，図形としてプログラムを与えれば，その図形がそのまま視覚的に動作する．しかしこれをデバッガとして用いる場合には，ユーザの意図に応じた情報の選択・抽象化，ユーザのインタラクティブな実行操作の機構が必要になる．

POD (Process Oriented Debugger) は，GHC の視覚的デバッガであり，HyperDEBU の立場ともっとも近い．それぞれの相違点は以下のとおりである．

- POD では対象とするストリームをリスト型に特化してそれに有用な機能を用意しているのに対し，

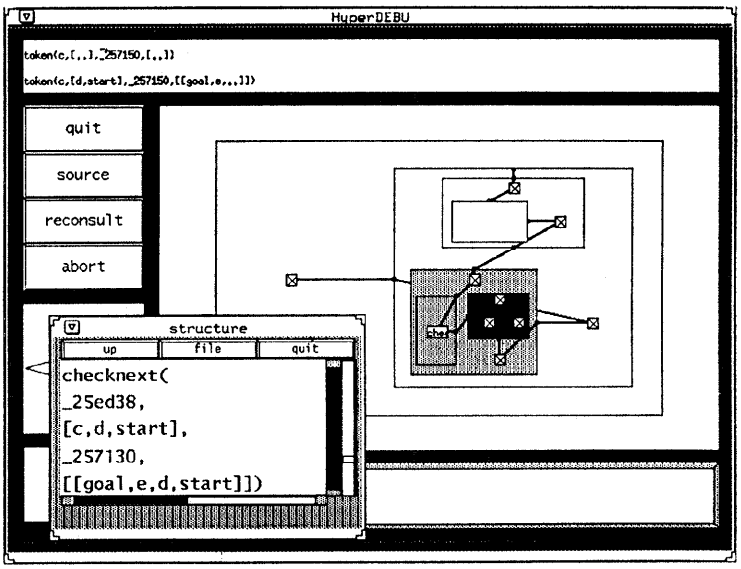


図 11 バグのあるプログラムの視覚化例 1
Fig. 11 Visualizing the erroneous program: scene 1.

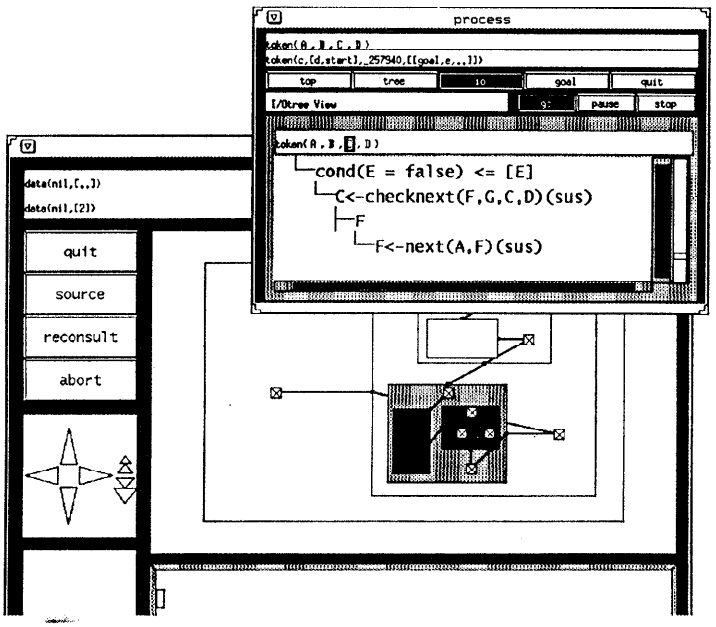


図 12 バグのあるプログラムの視覚化例 2
Fig. 12 Visualizing the erroneous program: scene 2.

HyperDEBU では一般化したストリームを対象としており，木構造など構造化されたストリームにも対応できる．

- POD ではストリームに関するすべての述語の引数について宣言が必要であるが，HyperDEBU では解析機能を用いて必要最低限の指定からユーザの意図を反映させる．
- POD では一列のゴールの実行をプロセスと見るのに対し，HyperDEBU ではゴールの集合をプロセスとして見ることで大規模なプログラムも階層的に把握できる．

これより，POD は比較的小規模で典型的なプログラムに特化して，その特性を生かした簡明で有用な機能を与えているのに対し，HyperDEBU ではより一般的で大規模なプログラムのデバッグを目的としているといえる．

8. 考　　察

6章で示されたように，データフローを視覚化することにより並列プログラムの実行の概略の把握がより効果的に行われる．ここでは，現在の実装における HyperDEBU のデータフローの表示のわかりやすさと課題点について考察・評価する．

8.1 得られた表示の利点と問題点

HyperDEBU が提供するデータフロー・グラフは，デバッガの表示として重要な以下の特長を持っている．

1. ユーザが配置情報を与えずに済む
2. 動的な変化に対応する

ただし，現在実装されている表示機能は，次のような場合については問題が残されている．

1. 一度に何種類ものストリームを視覚化する場合
   一つのプロセスからいくつもストリームが出て，それらが複雑につながるようになり，必然的にストリームの交錯が多くなって図形が見にくくなる．
2. 通信が長く継続される場合
   ノードが単調増加してしまうので図形が複雑になる．また，直線的なグラフでも複雑に折り曲げられて表示されるので，他のストリームとも交錯しやすくなる．

8.2 問題点の解決策

上記の問題点を解決するために次のような拡張機能を導入することが考えられる．

複数の流れの交錯の解決策

- マウスによるインタラクティブな識別
  マウスでポイントしたストリームを強調して表示する．また，ポイントしたノードにつながるストリームを強調して表示する．
- 色分けによる識別
  process および stream ブレークポイントで色も指定できるようにすることで，矩形やストリームを色分けして表示する．

グラフの単調増加の解決策

- 直列データのグルーピング
  ノードがストリームに直線的につながっている場合（すなわち，つながったリンク数が2以下のノードが隣接している場合）それらがデータまたはガードであれば，グルーピングする．それらは一方の矩形領域にまとめて表示され，他方の矩形領域は解放される（**図 13**）．
- より一般的なグルーピング
  ユーザの操作によるより一般的なグルーピング機能の提供も解決策として考えられる．“process” のブレークポイントの併用は，コントロールフロー指向のグルーピングともとらえられる．

これらの機能により，グラフの理解の複雑さを軽減することができるであろう．

9. 今後の課題

データフローの視覚化に関する今後の課題としては以下の点があげられる．

表示機能の強化　前章であげた表示機能の実装を行うことにより，よりわかりやすい表示を提供する．

静的情報の把握支援　ソースコードブラウザを強化してブレークポイントをより付けやすくする．静的データフロー／コントロールフローの視覚化を行ってユーザの着目したい部分を指定しやすくする．述語の呼びだし関係の中で，ストリーム変数がどのように伝搬し

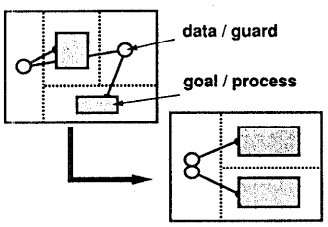


図 13 データのグルーピング
Fig. 13 Grouping method of data.

ているかを示す．

静的デバッギングの強化 現在の版では，視覚化のための解析機能を利用して簡易的な静的デバッギングを行っているが，より本格的で強力な静的デバッギング機能を開発し，ユーザからの情報を最大限活用する．これを行うには，CCL における型推論の機構を確立する必要がある．データに対するユーザの指定は型宣言と捉えられ，型チェックを行うことにより様々なバグをとり，また視覚化情報を得る．CCL のような並列論理型言語での型推論の研究がいくつか行われている[9]が，統枠な論理型言語と異なる点は，データが入力か出力か，またその依存関係を考慮することである．これにより，デッドロックなど並列プログラム特有の振舞いに適応が可能となるであろう．

10. おわりに

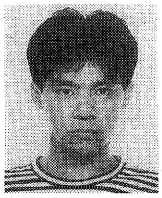
HyperDEBU は Fleng 自身で記述されており，UNIX ワークステーション上の逐次版 Fleng 処理系および Mach 並列ワークステーション上に実装された並列版 Fleng 処理系上で動作している．現在は，研究室内でアプリケーション開発において実用に供されており，これをもとに改良を続けている．

今後は，今回試作したデータフローの視覚化機能の評価および今後の課題として与えられた改善を行う予定である．

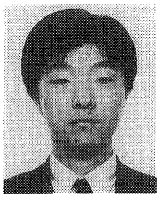
参考文献

1) Nilsson, M. and Tanaka, H.: Fleng Prolog—The Language Which Turns Supercomputers into Prolog Machines, In Wada, E. (ed.): *Proc. Japanese Logic Programming Conference*, ICOT, Tokyo, pp. 209-216 (June 1986).
2) 舘村純一，小池汎平，田中英彦：並列論理型言語 Fleng のマルチウインドウデバッガ Hyper-DEBU，情報処理学会論文誌，Vol. 33, No. 3, pp. 349-359 (1992).
3) 森下真一，沼尾雅之：PROLOG の視覚的計算モデル BPM とそれに基づくデバッガー PROEDIT 2, *Proceedings of the Logic Programming Conference '86*, pp. 177-184 (1986).
4) Brown, M. H.: Exploring Algorithms Using Balsa-2, *IEEE Computer*, Vol. 21, No. 5, pp. 14-36 (1988).
5) 市川 至，小野越夫，毛利友治：プログラム可可視化システム，情報処理学会論文誌，Vol. 31, No. 12, pp. 1801-1811 (1990).
6) Tick, E.: Visualizing Parallel Logic Programs with VISTA, *International Conference on Fifth Generation Computer Systems 1992*, pp. 934-942 (1992).
7) Kahn, K. M.: Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, *International Conference on Fifth Generation Computer Systems 1992*, pp. 943-950 (1992).
8) Maeda, M.: Implementing a Process Oriented Debugger with Reflection and Program Transformation, *International Conference on Fifth Generation Computer Systems 1992*, pp. 961-968 (1992).
9) Shin, D.: Towards Realistic Type Inference for Guarded Horn Clauses, 並列処理シンポジウム JSPP '91, pp. 429-436 (1991).

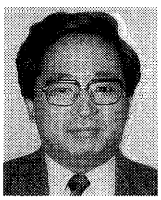
(平成4年9月24日受付)
(平成5年1月18日採録)



舘村 純一（正会員）
昭和 42 年生．平成元年東京大学工学部電子工学科卒業．平成3年同大学大学院工学系研究科情報工学専攻修士課程修了．現在，同博士課程在学中．並列処理，並列プログラミング言語とその環境，並列オブジェクト指向，ユーザインタフェース等に興味を持つ．



小池 汎平（正会員）
昭和 36 年生．昭和 59 年東京大学工学部電子工学科卒業．平成元年同大学院工学系研究科情報工学専攻博士課程満期退学．同年東京大学工学部電気工学科助手．工学博士．平成3年東京大学工学部電気工学科講師，現在に至る．並列計算機アーキテクチャ，および並列プログラミング言語に関する研究に従事．本会学術奨励賞受賞．日本ソフトウェア科学会，ACM 各会員．



田中 英彦（正会員）
昭和 18 年生．昭和 40 年東京大学工学部電子工学科卒業．昭和 45 年同大学院博士課程修了．工学博士．同年東京大学工学部講師，昭和46年助教授，昭和62年教授．昭和53年～54年ニューヨーク市立大学客員教授，現在に至る．計算機アーキテクチャ，並列推論マシン，知識ベース，オブジェクト指向プログラミング，分散処理，CAD，自然言語処理，等の研究を行っている．‘計算機アーキテクチャ’，‘VLSI コンピュータ I，II’，‘ソフトウェア指向アーキテクチャ’（いずれも共著），‘情報通信システム’著．電子情報通信学会，人工知能学会，日本ソフトウェア科学会，IEEE，ACM 各会員．