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
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学工学部

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。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。ここでは主にデータフローの視覚化の方式と実装された機能を述べる。

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. 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.

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、どこ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。

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 PROEDIT2 [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重要である (3) バグ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なうというものであり、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 HyperDEBU を開発している。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

Visualizing Data Flows of Fine-grained Highly Parallel Programs on a Multi-window Debugger HyperDEBU, Junichi TATEMURA, Hanpei KOIKE, Hidehiko TANAKA

覚化を行なう。このために、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・操作を行な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、ここではおもに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・同期が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] もこの 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。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節がコミットされる。

Fleng プログラムは次のような定義節の集合である。

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

:- の左側はヘッド部、右側はボディ部と呼ばれ、B_i はボディゴールと呼ばれる。

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、2 つの種類に分けられる。一つはヘッド部で行な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な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作用をし合うのではなく、ゴールは動的に生成・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クをたどることによってさらに希望するウインドウを開いてゆくことが可能である。

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から構成されおり、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が行なわれる。
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予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で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。CCL においては、前者としてはゴールリダクションが、後者としてはガードとユニフィケーションが相当する。HyperDEBU においては、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特有の問題は、表示すべきデータ・ゴールが動的に生成される点である。このために実行時に動的に画面上の配置を行なう手法が必要となる。

4.2 ブレークポイント

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なう場合、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) 定義節中のボディーゴールごとの指定 (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。

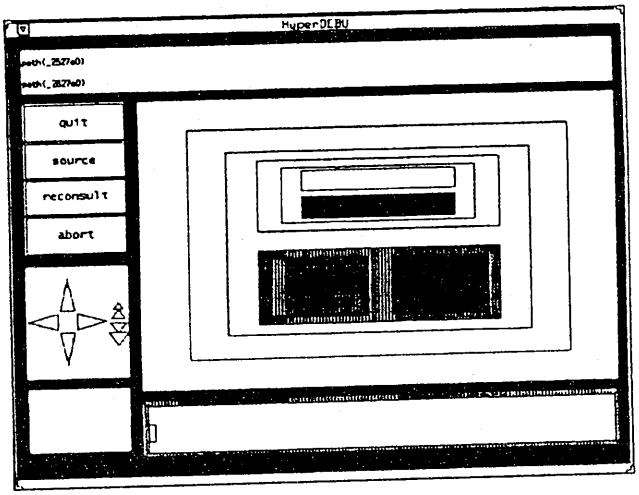

図 1: コントロールフローの視覚化

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。各矩形に対応するゴールを表示することにより、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、HyperDEBU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なう。

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なわれる。

> 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)。他のゴールはその値が確定すると、それを読んで処理を行なう (ストリーム入力)。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なわれる。

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様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。

以降では、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中心に述べる。

5 データフローの視覚化

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なわれる様子を図 2のように視覚化する。

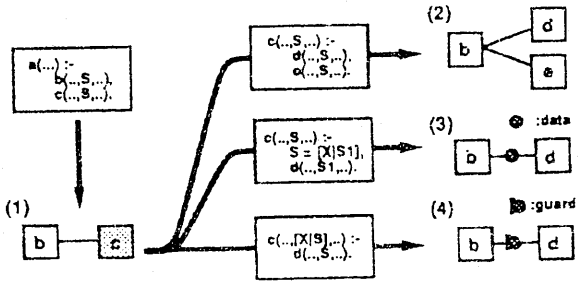

図 2: ストリームの生成・分配・入出力

- 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。ストリームが生成されるのは、定義節のボディーゴールに新たな共有変数が生まれた時である。
- 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である。
- 図 2-(3) はストリームに出力が行な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な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なわれた時である。このデータがガードとして表現されている。

5.2 ユーザの指定方法

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にあるような生成・分配・出力・入力の各動作にあたる部分をプログラム中から指定できればよい。しかし、このような部分を各定義節を調べながら一つ一つ指定するのはユーザにとって繁雑である。HyperDEBU では、これを解消するために、ユーザはどのストリームに着目するかだけを指定し、この情報を用いてデバッガが自動的に定義節中の各部分の動作を視覚化する。

ユーザは、ストリームを視覚化するために、各述語の引数について着目したい部分をブレークポイントとして指定する。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、画面中央左上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。

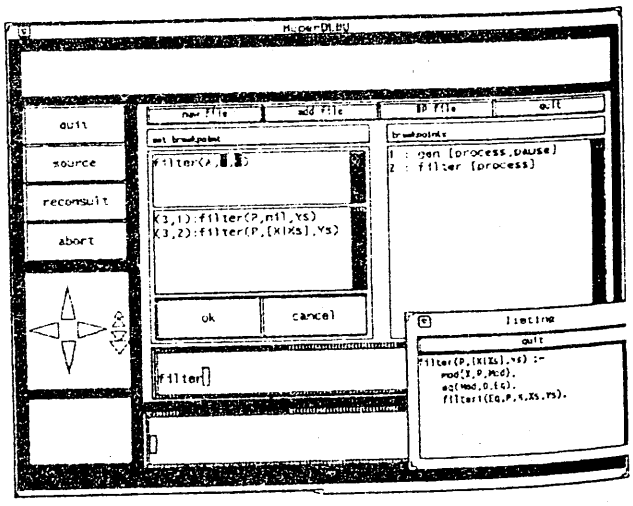

図 3: データフローのブレークポイントの設定

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が生成される。

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の開発を予定している。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などにより、プログラムのバグを発見することができるであろう。

デバッグのために視覚化を行なうには、あらかじめ実行前にブレークポイントを設定する必要がある。これ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、この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要求される。そこで、ソースコー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する。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、機能をより強力にした改良版の開発を進めている。

5.3 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、画面上にどのように配置するかが問題となる。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なわれる必要がある。視覚化のための手間がかかり過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すことが望ましい。
- 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過ぎないよう、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なわれる。

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ひとつのグループとし、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なわれる。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,…,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2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。

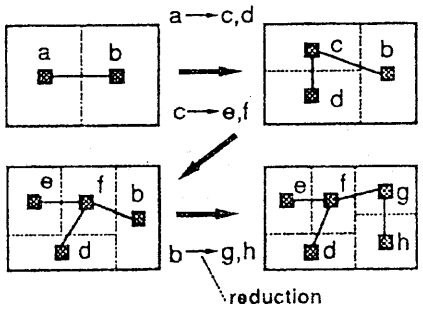

図 4: ゴール・データ・ガードの配置手法

データフローのためのブレークポイントとともに、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なわれる。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。

5.4 実装された機能

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視覚化例を図5 に示す。画面中央の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。

図中のノードのうち、大きな矩形は、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。小さな正方形はストリームの出力によってつくり出されたデータである。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。中に文字の書かれた長方形はゴールである。これらをつないでいる線がストリームで、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、マウスで直接操作をすることにより、プロセスからは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。データ・ガード・ゴールをクリックすると、それらを観察するためのストラクチャウインドウが開く。図の中で、手前側のウインドウが開かれたストラクチャウインドウである。

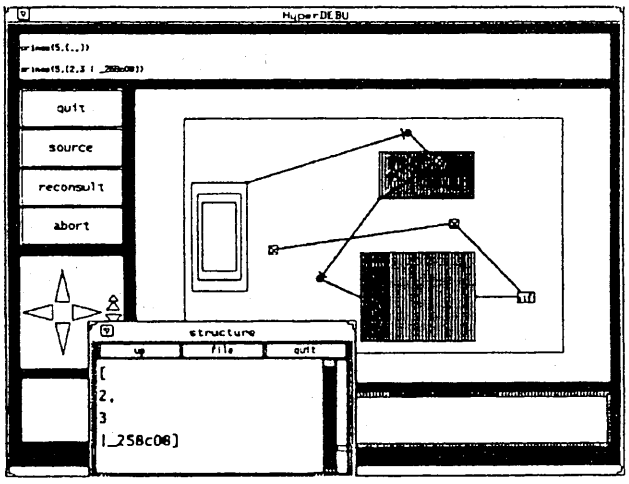

図 5: データフローの視覚化例 1

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 6は、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。

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 7) をとりあげる。

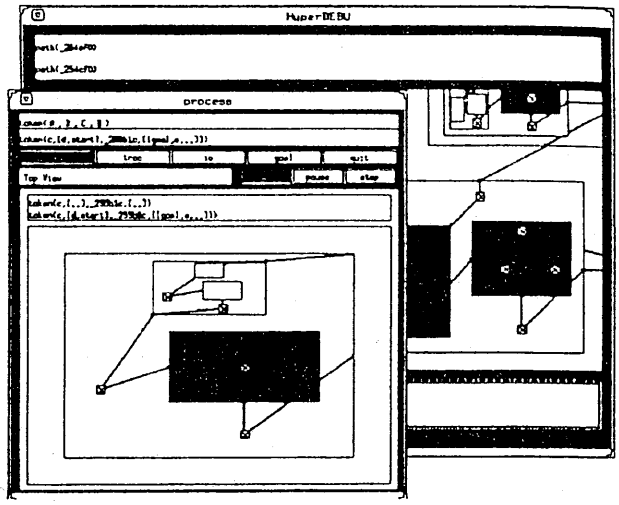

図 6: データフローの視覚化例 2

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 までの経路を全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なう。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。

このプログラムをデバッグするための第一段階として、視覚化を行な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。このようにしてプログラムを実行した結果が図 8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

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

図 7: バグのあるプログラムの例

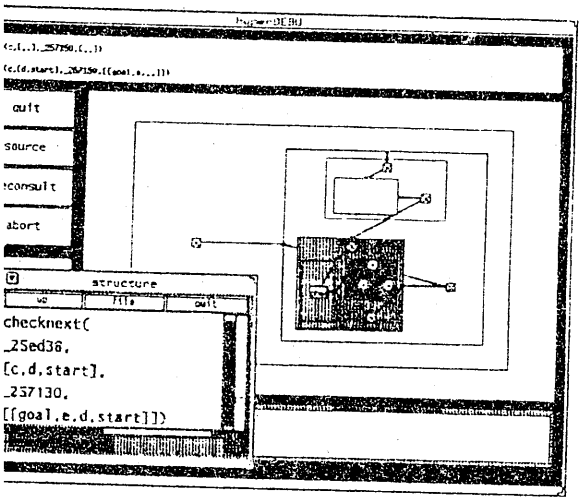

図 8: バグのあるプログラムの視覚化例 1

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 9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。

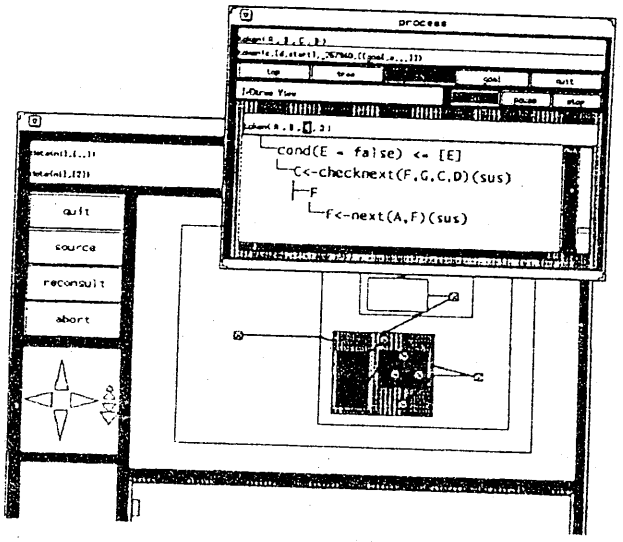

図 9: バグのあるプログラムの視覚化例 2

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。

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 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。

- 配置アルゴリズムの改善をおこない、ストリーム同士の交錯をよりすくなく、より見やすい形に配置を行なう。特定のストリームまたはノードに着目した配置法も考えられる。
- グラフの単調増加を防ぐためにグルーピングによる抽象化を行なう。"process" のブレークポイントを併用すれば、コントロールフロー指向のグルーピングができるが、他に、ストリームにつながったデータ群を一つにまとめるようなデータフロー指向のグルーピングも検討する。
- 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なってユーザの着目したい部分を指定しやすくする。
- 静的デバッギングとの融合をはかる。ブレークポイントとしてユーザから与えられた情報を有効に利用して、静的データフロー解析を行なう。
- 色の利用などの表示機能の強化を行なう。

8 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なう予定である。

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. June 1986. p209-216.

[2] 舘村, 小池, 田中 : 並列論理型言語 Fleng のマルチウインドウデバッガ HyperDEBU, 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 PROEDIT2, PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '86, pp.177-184 (1986).

1988).

[5] 市川至, 小野越夫, 毛利友治: プログラム可
ステム, 情報処理学会論文誌, Vol.31 No.12
1811 (1990).