並列論理型言語 Fleng のマルチウインドウデバッガ HyperDEBU

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

並列プログラムのデバッグは逐次プログラムの場合よりも困難といわれている。並列論理型言語の一種である Committed-Choice 型言語（CCL）は細粒度で高並列な実行を実現しているが、制御やデータの流れが多数存在しており、それを観察・操作することは難しい。デバッガはプログラムの実行をモデル化してユーザに示すものであると考えれば、細粒度高並列プログラムの実行を表現するのに適したモデルを導入する必要がある。我々は、通信し合うプロセスとして CCL のプログラムの実行を表現したが、このモデルは抽象化のレベル・側面に自由度があり、細粒度高並列プログラムのモデルとして適している。次に我々はこのモデルを用いたマルチウインドウデバッガ HyperDEBU を開発した。並列プログラムの実行過程は多次元的な情報としてとらえられるから、これをデバッグするには多次元的なインタフェースが必要となる。また、ユーザはデバッガが示したモデルと自分の意図するものとの比較によりバグを発見するので、デバッガはユーザが望むビューを提供する必要がある。HyperDEBU は、抽象化のレベル・側面に自由度を持ったビューを多次元的インタフェースの上に提供し、多数の制御／データの流れが形成する複雑な構造の観察・操作を可能にしている。

1　はじめに

並列プログラムのデバッグは逐次プログラムと比べて極めて困難である。これは、複数のプロセスが互いに干渉しながら同時に動作しているため、実行の理解・制御が困難であることが原因であり、その動作にしばしば非決定性が含まれることが更にデバッグを困難にしている。従来の並列デバッガの多くはプロセス毎のトレースを行なうものに過ぎず、プロセス間の関係の把握が困難であるといった問題を抱えている。そこで、プロセス－時間ダイアグラムやアニメーションなど、時間軸に平行／垂直な面へ射影して実行過程を表現する手法が研究されている [1]。

Committed-Choice 型言語は並列論理型言語の一種であり、細粒度で高並列な計算を実現している。並列度が高くてプロセスが多数ある場合、特に Committed-Choice 型言語のような細粒度高並列プログラムの場合は、粒度の小さいプロセスが多数存在するのでプロセス毎のトレースが困難な上に、プロセスダイアグラムをとってもデータ／制御の流れが多数あって理解しづらい。アニメーションを行なっても多数のプロセスが多数が動的に生成・消滅するような場合には、その動作を追っていくことは難しい。細粒度並列言語は、各動作の時間関係よりもプロセス・データの依存関係が重要であるため、時間を軸にとるよりも、依存関係が分かりやすいように抽象化した表示が必要であろう。

デバッガはプログラムの実行をモデル化してユーザに示すので、デバッグの問題を解決するには、細粒度並列プログラムの実行を表現するのに適したモデルを導入しなければならない。我々は Fleng プログラムの実行を互いに通信し合うプロセスとしてモデル化し、これに基づいたマルチウインドウデバッガ HyperDEBU を開発した。

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 [2] は CCL の一つであり、我々の研究室では Fleng を高速に実行する並列計算機 PIE64 [3] を開発している。Fleng は他の 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 は言語の仕様の中に同期・通信の機能が組み込まれている。まだ書き込んでいない変数を他のプロセスが読んでしまうようなバグは避けられる。これにより、非決定的な同期のバグは大幅に軽減される。

HyperDEBU : a Multiwindow Debugger for Parallel Logic Programs
Junichi TATEMURA, Hanpei KOIKE, Hidehiko TANAKA
Faculty of Engineering, The University of Tokyo

Fleng は、個々のゴールが並列実行される細粒度並列言語である。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 Fleng プログラムの実行のモデル化

3.1 モデル化のための条件

デバッガはプログラムの実行をモデル化してユーザに示すものであると考えられる。ユーザはこのモデルを通してプログラムの実行を観察・制御し、自分の意図するものとの比較によりバグを発見する。

そこで、CCL のプログラムの実行のモデル化を行なう場合にどのようなものが望ましいかについて、純粋な論理型言語との違いと細粒度高並列言語という二つの点から述べる。

CCL プログラム　CCL ではホーン節にガードの概念を加えており、純粋な論理型言語ではなくなっているので、宣言的意味の中にも操作的な要素が加えられる。CCL におけるセマンティクスの問題が論じられているが、そこには純粋な論理型言語にはないような問題が出てくる。

CCL のプログラムの意味を考える場合、入力と出力の同期の関係 (入出力因果関係) も記述する必要があるということが、GHC のセマンティクスに関する諸研究 [4] [5] で論じられている。

細粒度高並列プログラム　細粒度な並列プログラムのデバッグのための実行モデルとして望まれる特徴は以下のような点である。

- プロセスのグループ化 (抽象化のレベル)
  多数のプロセスを抽象化して、より抽象度の高い一つのまとまりとして扱う必要が出てくる。抽象化のレベルには高い自由度を持たせることが望ましい。
- 複数のビューを持つ (抽象化の側面)
  時間軸にそった表現だけでなく、通信あるいは制御などに着目して表現するといった抽象化の切口の自由度が要望される。

3.2 プロセスモデル

プロセスの概念　並列論理型言語ではゴール一つをプロセスとみなすことがあるが、ここでは一つのゴールだけでなく、そのゴールから生成される全てのゴール (サブゴール) を含めて一つのプロセスと考える。プロセスの動作の様子は外部からはプロセスの入出力としてとらえられる。

ゴールがリダクションされていくつかのサブゴールができるのに対応して、プロセスの内部はまたいくつかのサブプロセスとして表現できる。これを計算木と対応付けて考えてみる。計算木はプログラムの実行された様子を木の形で表したものである。その内部には、あるノードをルートとするような部分木が存在するわけであるが、この部分木は一つのプロセスにあたる。計算木が部分木に分割されるように、プロセスはいくつかのサブプロセスに分割される。

このようにプロセスを考えると、定義節はプロセスとサブプロセスの関係を規定するものであると考えることができるであろう。

プロセスモデルの定義　上で述べたプロセスモデルは以下のように定義される。

ここで、プログラム中で実行されるあるゴール G と、そこから生成されるゴールからなる集合を実体とするプロセスを P としたとき、P を「G に対応するプロセス」、G を「P のトップゴール」ということにする。

ゴール G に対応するプロセス P は外部から見た場合以下のように表現できる。

$$\langle G_{skel},\ I,\ O,\ S,\ G_{ins} \rangle$$

G_{skel} は G の *skeletal predicate* であり、ゴール G の引数を全て個別の変数でおきかえたものである。すなわち、

$$p(v_1, \ldots, v_i)$$

$v_1, \ldots, v_i$ はそれぞれ個別の変数で、この変数が外部との通信の窓口に用いられる。

I,O は入力 / 出力データフローであり、これがプロセスの入出力因果関係を表す。G_{skel} の変数がプロセスの外部及び内部からどのようにユニファイされ具体化されていくかを示したものである。これはユニフィケーションの木として表されるので、I/O tree と呼ぶことにする。この詳細は後述する。

S はプロセスの状態を表す。これには次の状態がある。

- *suspend* : プロセスの内部にアクティブなゴールがなく、サスペンドしたゴールのみとなった状態を示す。 外部からの入力によって内部のゴールがアクティベートされた場合は *active* に変化する。
- *active* : プロセスの内部にアクティブなゴールが残っている状態を示す。

このようにプロセスの状態を定義すれば、実行途中のプログラムや、無限に続くプログラムも扱うことができる。

G_{ins} はゴールの *instance* である。 これは、 G_{skel} に対して内部と外部からユニフィケーションが行なわれ、変数が束縛されてできた結果を表したものと考えることができる。

プロセスの入出力 プロセスの入出力データフローは I/O tree として表現される。これは、プログラム実行におけるデータフローをあらわすユニフィケーションの木である。

ゴール G に関するプロセスを考えると、このプロセスの G_{skel} の引数は通信の窓口と考えられるから、各窓口に入力データフロー・出力データフローが存在する。入力データフローはプロセスの外側の、出力データフローはプロセスの内側のユニフィケーションの木 (I/O tree) である。

I/O tree O は以下の構文で表現される。

$$
\begin{aligned}
O \;\; ::= & \;\; C|O \\
| & \;\; U \times O \\
| & \;\; O + O \\
| & \;\; Nil
\end{aligned}
$$

上に現われる演算子は以下のような意味を持つ。

1. $C|O$: C に示されているような入力が存在することを条件に出力 O が存在する。
2. $U \times O$:ユニフィケーション U が行なわれ、さらに出力 O が存在する。
3. $O + O$:入出力のパスの分割を表す。

この I/O tree は、プログラムの実行においてコミットされた定義節から定義される。定義節はプロセスとサブプロセスの関係を規定するものであり、プロセスの入出力とサブプロセスの入出力の関係も定義節によって決まる。あるゴールについてコミットされた定義節を与えれば、プロセスの I/O tree は再帰的に定義できる [6]。

3.3 プロセスモデルの特徴

ここで提案しているプロセスモデルには次のような特徴がある。

- 階層的 = プロセスは幾つかのサブプロセスに分割できる (幾つかのプロセスからより抽象度の高いプロセスが構成される)。
- 多面的 = ゴールの集合 / 計算木 / 入出力因果関係といった複数のビューを持つ。

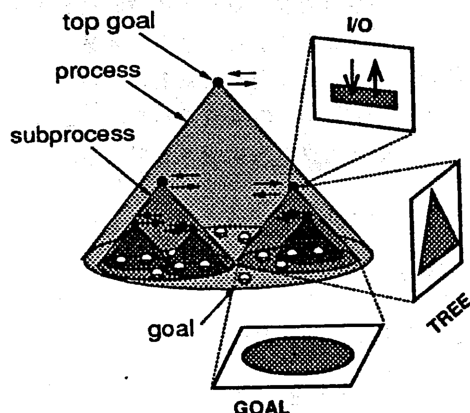
これを概念図として表したのが図 1である。これは、細粒度高並列プログラムに適したモデルの要件を満たしている。

これに対して、プロセスの見方として、

- 一つのゴールをプロセスと見る
- シーケンシャルなゴールの列をプロセスと見る

という見方があるが、これらは大規模で高並列なプログラムでは、シーケンシャルな列が多数生成・消滅するのでデバッグには向かないと思われる。

CCL のプログラムの意味を考える場合は従来の論理型言語の意味に加えて入出力の因果関係を考慮しなければならないが、ここで述べた Fleng のプロセスモデルでは入出力因果関係がわかるように実行のモデル化が行なわれている。このことから、このプロセスモデルはプログラムの意味として充分なモデルといえる。



4　マルチウインドウデバッガ HyperDEBU

4.1　細粒度高並列プログラムのデバッガ

逐次的なプログラムは実行の流れが一本だけなので、逐次的で一次元的なインタフェースでデバッグが可能であった。しかし、並列プログラムでは、制御の流れとデータの流れが多数存在して、それらが複雑に交錯している。これは、並列プログラムの実行過程は多次元的情報であるということを意味する。このため、一次元のインタフェースではユーザとプログラムの間がボトルネックとなって、プログラムの実行を観察することも、逆にユーザから操作を加えることも困難である。これは細粒度高並列プログラムでは特に問題であり、プログラムの大規模並列化が進めば一層顕著になるであろう。並列プログラムの実行を観察・制御するには多次元的インタフェースが必要である。

デバッグはユーザが意図するものとデバッガが表示するものとの比較によって行なわれるものであるから、ユーザが何をどのように見たいかに応じたビューが必要となる。これを提供するためには、抽象化のレベル / 側面の自由度を持ったビューが必要であろう。

我々は、細粒度高並列言語 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。

従来のマルチウインドウデバッガの多くは各プロセスに逐次プログラム用のデバッガを割り当てるものであった。しかし、単にウインドウを割り当ててプロセスごとの情報を分割しただけでは多次元的な情報を取り扱い得ない。このデバッガは、細粒度並列プログラムの実行過程において制御 / 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。

図 2は HyperDEBU の概観であるが、このデバッガは以下のウインドウから構成されている。

1. トップレベルウインドウ
2. プロセスウインドウ
   (a) TREE ウインドウ
   (b) I/O tree ウインドウ

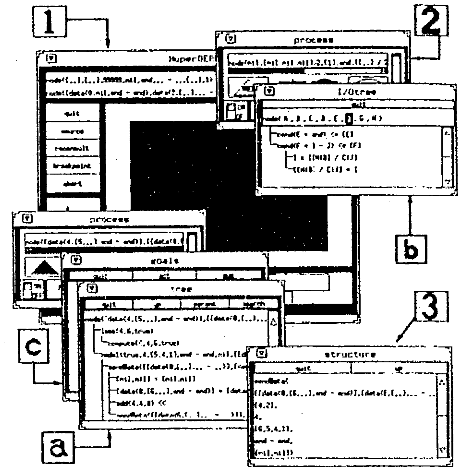


図 2: HyperDEBU の概観

4.2　HyperDEBU の機能

4.2.1　多様な視野を用いたバグの絞り込み

トップレベルウインドウとプロセスウインドウ　プログラムの実行を観察・操作しつつ、バグを絞り込んでいき、バグに到達するためには、グローバルな視野からよりローカルな視野までをサポートすることが要求される。これを実現するために、 HyperDEBU ではトップレベルウインドウとプロセスウインドウが用意されている。

トップレベルウインドウはグローバルな視野を提供するウインドウである。 HyperDEBU を起動するとまずこのウインドウが開き、ユーザはこれに対して初期ゴールを投入する。このトップレベルウインドウの上で、ユーザはプログラム全体を観察・操作できる。実行情報を更に詳しく調べるためにはプロセスウインドウが用いられる。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それぞれから開くことができる。トップレベルウインドウはこれらのウインドウを管理する働きを持っている。

プロセスウインドウは、任意のゴールに関するプロセスに対して割り当てることができるウインドウであり、そのプロセスに対する観察・操作が行なえる。プロセスからは、サブプロセスをウインドウとして取り出すことができ、バグの絞り込みを可能にしている。

また、HyperDEBU ではデータレベルの視野もサ

プロセスウインドウの提供するビュー　プロセスウインドウは、プロセスモデルの

- 抽象化のレベルの自由度
- 抽象化の側面の自由度

といった特徴を生かしたバグの絞り込みを可能にしている。プロセスモデルの3つのビューに対応して、プロセスウインドウは

- TREE ウインドウ：計算木
- I/O tree ウインドウ：入出力因果関係
- GOAL ウインドウ：ゴールの集合

といった3つのサブウインドウを用意しており、多様な側面からプロセスを観察することができる。それぞれから、よりローカルで抽象度の低いプロセス (サブプロセス) をウインドウとして取り出すことができる。これにより、効果的なバグの絞り込みが期待される。

4.2.2 プログラム実行の視覚化

プログラムの実行の視覚化は、トップレベルウインドウのグローバルなビューとして実現されている (将来的には各プロセスウインドウでも同じ視覚化を行なうことを検討している)。

CCL の実行は、以上の二つの流れを視覚化することにより表される。

- 制御の流れ：ゴールリダクション
- データの流れ：ガードとユニフィケーション

それぞれの履歴は計算木と入出力因果関係という形でTREE ウインドウと I/Otree ウインドウの上に表現されている。しかし、CCL の様な細粒度高並列プログラムでは、個々のゴール・個々のデータを全て視覚化したのでは繁雑になり過ぎる。トップレベルウインドウでは、制御の流れに関しては、ある特定のゴールについてのプロセスのみを表示し、データの流れに関しては、そのプロセス間に存在する特定のデータの流れだけを表示することにする。ここで、何が「特定」かはユーザの主観によるところが大きいので、ユーザが指示を与える必要がある。HyperDEBU では、これをブレークポイントとして指定する機能を備えている。

また、Fleng などの CCL の実行の視覚化において注意すべきことの一つとして、動的にデータ・ゴールが生成されるために静的に配置が決まらないという点がある。このために、動的な視覚化手法が必要となる。

制御の流れ　トップレベルウインドウでは、特定のゴールに関するプロセスのみを表示することにより、制御の流れに関するグローバルな視野を提供する。図3はトップレベルウインドウの表示である。これはプロセスの概念図1を上から見たものととらえることができる。

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。

- 矩形の模様はプロセスの状態を表す (白色 = active / 淡灰色 = suspend / 濃灰色 = terminated)。
- 矩形の入れ子構造はプロセスとサブプロセスの関係を表す。
- マウスカーソルが矩形上に入るとゴール表示部にトップゴールが表示される。
- 矩形をマウスで操作することにより、それぞれプロセスウインドウを開くことができる。

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、トップゴールの引数のデータの変化が把握できる。これにより、プログラムの実行状況を把握することができる。

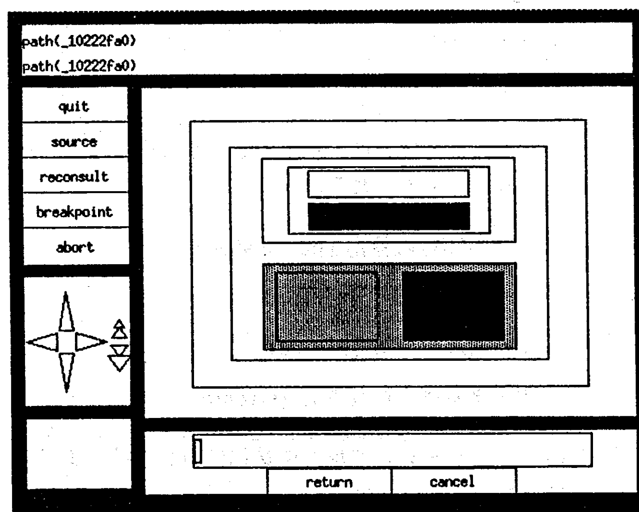


図 3: トップレベルウインドウ

データの流れ　データの流れについては I/Otree ウインドウで入出力因果関係の木として表現されている。しかし、プログラムが大規模になった場合に、広域にわたるデータフローを把握するには I/O tree による

することにより、データの流れに関するグローバルな視野を実現する必要がある。これには、トップレベルウインドウで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することが考えられる。

4.2.3 ブレークポイント

逐次プログラムでは、ブレークポイントで停止して、実行状態を見る、もしくはトレース情報として表示するといったデバッグ手法がとられてきた。しかし、実行の流れが多数ある場合は、それらを一旦停止させても観察するのが困難で、逐次プログラムのようにステップ実行や、トレース情報の表示等を繰り返すのは容易ではない。

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、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。デバッガは、この情報をプログラムの実行制御・実行の視覚化・静的デバッグに活用することが考えられる。

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としては、以下のものが考えられる。

- 述語
- 定義節
- ボディーゴール
- 引数

ブレークポイントによって指定できるものとしては、以下のものが考えられる。

- ゴール実行の停止 (pause,stop)
- プロセスの切り分け (process)
- 実行履歴の制御 (notree)
- データレベルの視覚化 (stream)

このうち、データレベルの視覚化に関しては、まだ実装は行なわれていない。

ブレークポイントはあらかじめ設定していなければならない点が繁雑であり、その時のユーザの負担を軽減するためのサポートが必要となる。ブレークポイントは静的な情報をもとにユーザが判断するものであるから、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。

4.2.4 プログラムコードのブラウジング

- 述語の呼びだし関係の把握
  処理の流れを静的に把握するために、ある述語がどのような述語を呼び出しているか、どのような述語から呼び出されているかといった相関関係のグラフをトレースする機能で、それぞれの述語からその定義を取り出すことができる。現在はトレース機能だけだが、より広い視野で全体的な流れを把握するためには、グラフィックに表示する機能が必要であろう。
- 実行情報とソースプログラムの対応付け
  実行履歴を観察している時に、表示されている部分からそこでコミットされた定義節を直接取り出す。また、各所の表示にあられる述語名からその定義を取り出す。
- 述語名による検索
  述語名を入力してその述語に関する情報を取り出したり、実行履歴の検索機能などで、述語入力時に述語名の補完機能をサポートする。

これらの機能は、以下のような支援に適用できるであろう。

- ブレークポイント設定の支援
- 静的デバッグの支援
- ソースコードの修正の支援

プログラムコードのブラウジング機能は、現状ではまだ簡単なものが試作されている段階であり、今後さらに機能を拡張する必要がある。また、静的解析との融合も課題である。

5 デバッグ例

本章では HyperDEBU を用いたデバッグの例を示してその有効性を主張する。

以下の例は経路探索プログラムである。

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

token(Node,History,H,T) :-
        (Node == goal ->
           H = [[goal|History]|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(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].

member(_, [], R) :- R = false.
member(X, [Y|Z], R) :-
        (X == Y ->
            R = true
        ;
            member(X, Z, R)).
```

これは、next という述語で表現される有向グラフにおいて、ノードである start から goal までの経路を全て求めるプログラムであり、token というゴールを経路にそって生成することにより解を求めている。しかし、next の定義節において、入出力の間違いが存在する。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。

まず、トップレベルウインドウにおける視覚化のためのブレークポイントを設定する。ここで中心となっている述語は token であろう。これをブレークポイントとして指定すれば、token がどのように生成されていくかが見えるはずである。図 4はトップレベルウインドウでの制御の流れの視覚化の様子である。ここで、サスペンドしているプロセスをプロセスウインドウとして取り出す。正しいプログラムでは第 3 引数に出力があるはずであるが、トップゴールを見ると変数のままである。そこでI/O tree ウインドウを用いて第 3 引数の出力を見てみる (図5)。これを見ると、checknext が出力せずにサスペンドしている。ストラクチャウインドウでインスタンスを見てみると第一引数が変数のままであり、これは next がサスペンドしているためである。表示されている next から定義節をウインドウとして取り出せば、バグのある定義節を発見することができる。

このように、あるゴールがサスペンドすると、それによって他のゴールもサスペンドし、その連鎖反応でサスペンドゴールが多数になる時がある。このような時は I/O tree ウインドウによってデータの依存関係がわかるとバグの発見に役立つ。また、トップレベルウインドウにおける視覚化により、プログラムの実行状況の把握が可能となり、これがバグの絞り込みを効率化している。より大きなプログラムではこれが顕著となろう。

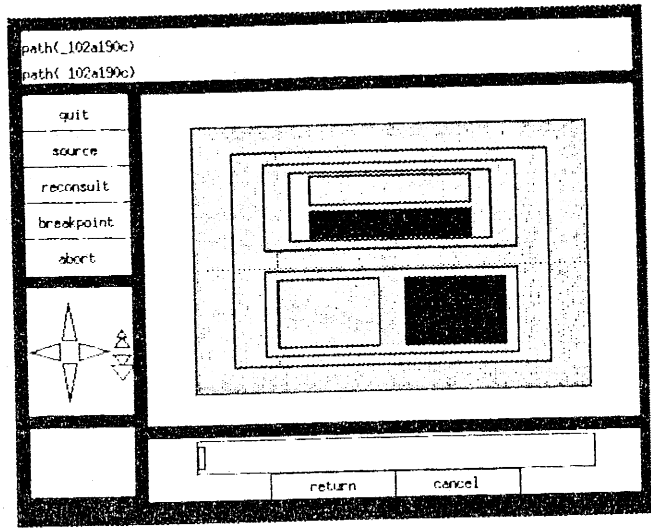


図 4: デバッグ例 1-1

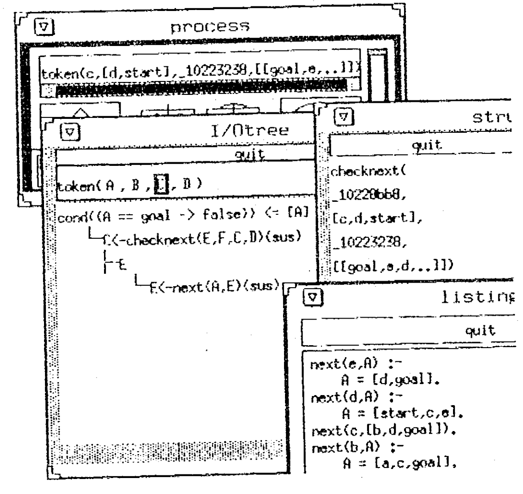


図 5: デバッグ例 1-2

6 考察と課題

6.1 非決定性

CCL における非決定性は、以下の二点に帰着される。

- 定義節の非決定的コミット
- 競合するアクティブユニフィケーション

後者はどちらかのユニフィケーションが失敗するので検出が可能である。通常、正しいプログラムはゴー

ルの実行が失敗するようなことはないので、このような時はバグが存在するとみなされる。このとき、I/O tree を用いてユニフィケーションのパスを見れば、複数の出力が一つの変数に対して競合しているのが発見されるはずである。

前者の非決定的なコミットに関しては、入出力因果関係が OR の関係になっていることで表現できるであろう。しかし、それが実際に非決定的な結果を起こし得るかどうかは良く調べないとわからないので、解析ツールが必要となろう。ただし OR になった I/O tree 全体を完全に調べるのは計算量が爆発するので取り扱わない。プログラム検証ではなくデバッグという立場であれば、実際に起こった結果について取り扱い、他の可能性に関してはその存在を示唆するのみで十分であろう。

また、このようなプログラムを操作するには、非決定的動作に対する実行制御が必要になる。このために、非決定的なコミットに関するブレークポイントを用意する必要があろう。これに関しては、以下のようなものが考えられる。

- 非決定的なコミットが起こる部分でゴールが停止する。
- 指定された方の定義節がコミットされる。

さらに、実行結果のオルタナティブが存在するような場合は、その分岐点から再実行する機能が有効であろう。このためには、実行の途中状態を外部に記憶してその点からリプレイする機能が望まれる。

6.2 計算量・メモリ消費量

デバッガでプログラムを実行した場合のオーバヘッドとしては、デバッグ用の処理を行なうため計算時間がかかる、履歴をとるとメモリの消費量が増大するといった点が挙げられる。

現在、HyperDEBU では、対象プログラムのゴール実行管理にメタインタプリタを使用して試作した機構を用いているため、実行時間のオーバヘッドが大きい。この問題に対しては、(1) 対象プログラムをプログラム変換する (2) コンパイラを拡張してデバッグ用のコードを埋め込む (3) 処理系による直接のサポートを行なうなどの改良版を検討している。

並列プログラムのデバッグにおいて実行履歴は重要であるが、全実行について履歴を記録するのは記憶領域の制限上、現実的ではない。

現在は実行履歴 (計算木) の記録をするかしないかを制御するブレークポイントでこれに対応している。実行履歴をすべて記録しないでも、視覚化を行なうことによって、必要な制御依存関係、データ依存関係だけを残すことが可能である。

将来的な課題としては、履歴情報を以下に効率良く記録するかといった技術や、部分的に計算し直すことにより、履歴を再現するといった手法が考えられる。

7 おわりに

本論文では、細粒度で高並列な Committed-Choice 型言語 Fleng のプログラムの実行を表現するモデルを示し、これを用いたデバッガとして多次元的なインタフェースを用いたデバッガ HyperDEBU について述べた。

現在、UNIX 上の Fleng 処理系の上で Fleng 自身で記述された HyperDEBU が動作しており、Fleng アプリケーションプログラマに使用してもらうことにより評価中である。

参考文献

[1] Mcdowell,C.E. and Helmbold,D.P.: *Debugging Concurrent Programs*, ACM Computing Surveys, Vol.21 No.4, pp.593-622 (1989).

[2] Nilsson, M. and Tanaka, H.: *Massively Parallel Implementation of Flat GHC on the Connection Machine*, *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, p1031-1040 (1988).

[3] Koike, H. and Tanaka, H. : *Parallel Inference Engine PIE64*, in *Parallel Computer Architecture*, bit, Vol.21,No.4,1989, pp.488-497 (in Japanese).

[4] Murakami, M. : *A Declarative Semantics of Parallel Logic Programs with Perpetual Processes*, Technical Report TR-406, ICOT, 1988.

[5] Takeuchi, A. : *A Semantic Model of Gurded Horn Clauses*, Technical Report, ICOT, 1987.

[6] 舘村, 田中: 並列論理型言語 FLENG のデバッガ, PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '89, pp133-142 (1989).