Vol. 33 No. 3 情報処理学会論文誌 Mar. 1992

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††

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．われわれは，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．次にわれわれ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さら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ごとのトレースを行うものにすぎず，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ごと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っても多数のプロセスが動的に生成・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．細粒度高並列言語は，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，依存関係がわかりやすいように抽象化した表示が必要である．

デバッガの役割はプログラムの実行をモデル化してユーザに示すことである．プログラムがユーザの思考の産物であるとすれば，デバッガはその思考を映す鏡であるといえる．ユーザはデバッガが示した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．細粒度高並列プログラムのデバッグの問題を解決するには，そ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．われわれは Committed-Choice 型言語の一つとして言語仕様が簡潔な Fleng をとりあげ，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．

細粒度高並列プログラムの実行では，制御・データの流れが複雑なグラフ構造を形成する．HyperDEBU は，このような多次元的情報を取り扱うために，多次元的インタフェースを用いて多様な視野をユーザに提供し，高並列プログラムの観察・操作を可能にする．

2章では，本デバッガが対象とする並列論理型言語 Fleng について簡単な説明を行う．3章では，デバッグのためにFlengの実行を表現するモデルとしての要件を考察し，これを実現する実行モデルについて概説する．4章では，Fleng のデバッガ HyperDEBU の特徴と機能について述べる．5章では，HyperDEBU の使用例を示し，このデバッガの有効性を主張する．

---

† HyperDEBU: A Multiwindow Debugger for Parallel Logic Programming Language Fleng by JUN'ICHI TATEMURA, HANPEI KOIKE and HIDEHIKO TANAKA (Department of Electrical Engineering, Faculty of Engineering, The University of Tokyo).

†† 東京大学工学部電気工学科

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，われわれの研究室では Fleng を高速に実行する並列計算機 PIE 64[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ので，非決定的な同期のバグは大幅に軽減される．

Fleng は，個々のゴールが並列実行される細粒度高並列言語である．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

並列論理型言語 Fleng のプログラムのデバッガを構築するためには，そのデバッガが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．そこで，CCL のプログラムの実行のモデル化に望まれる要件を，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．

（1） CCL プログラム

CCL ではホーン節にガードの概念を加えているため，純粋な論理型言語と異なり，宣言的意味の中にも操作的な要素が加えられる．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．

（2） 細粒度高並列プログラム

細粒度で高並列な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点は，ユーザの限られた情報処理量の中で，大量の情報をいかに抽象化してバグの発見を助けるかということである．これには以下の要件があげられる．

- 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ために，抽象化のレベルには高い自由度が要求される．
- 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 Fleng のプロセスモデル

われわれは Fleng プログラムのデバッグのために，通信するプロセスとしてその実行をモデル化した[6)]．本節ではこのプロセスモデルについて概説する．

（1） Fleng におけるプロセスの概念

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すべてのゴール(サブゴール)を含めて一つのプロセスと考える．

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．

（2） 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 の引数をすべて個別の変数でおきかえたものである．す

なわち，

$p(v_1, \cdots, v_i)$

$v_1, \cdots, v_i$ 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 の入力／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およ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．

- terminate：プロセス内のゴールの実行がすべて終了している状態を示す．
- 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．

(3)　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表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ような構造を持つ．

$$O ::= C|O$$
$$\quad |\ U \times O$$
$$\quad |\ O+O$$
$$\quad |\ \mathrm{Nil}$$

これはそれぞれ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用いると，そのゴールに対応するプロセスの入力とサブプロセスの出力からプロセスの出力とサブプロセスの入力が定義できるので，初期ゴールの入力と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＝プロセスはいくつかのサブプロセスに分割できる（いく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，CCL における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

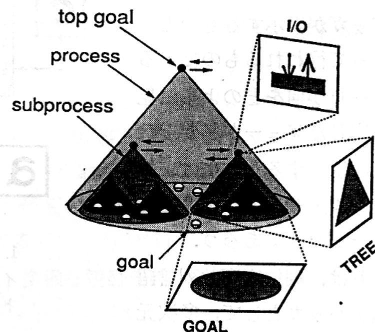

図 1　プロセスの概念図
Fig. 1　The process model.

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十分なモデルといえる．文献 6) では，これをアルゴリズミックデバッギングに適用した．

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が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．

われわれは，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[1]．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えない．ここで必要とされる多次元的インタフェースとは，単にマルチウィンドウやグラフィックを用いるという意味ではない．このデバッガは，細粒度高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．番号は図中の番号と対応している．

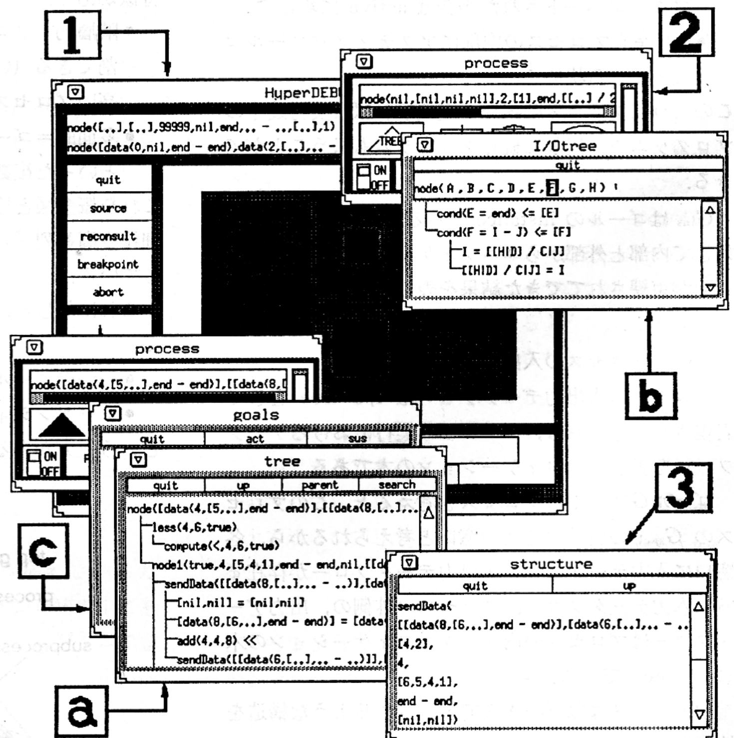

図 2 HyperDEBU の概観
1. トップレベルウインドウ，2. プロセスウインドウ（a. TREE ウインドウ，b. I/O tree ウインドウ，c. GOAL ウインドウ），3. ストラクチャウインドウ

Fig. 2 Overview of HyperDEBU.
1. toplevel-window, 2. process-windows (a. TREE view, b. I/O tree view, c. GOAL view), 3. structure-windows.

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
　(c) GOAL ウインドウ
3. ストラクチャウインドウ

4.2 デバッグに必要とされる機能

前節で述べた設計思想をもとにデバッガを実現するには，どのような機能が必要となるかが課題となる．本節では，細粒度高並列プログラムのデバッグはいかに行われるべきかを考察し，それに必要となる機能について論じる．

(1) 実行状況の把握

プログラムを実行させてみて予期せぬ動作をする場合には，まず実行の様子を把握することが必要である．逐次プログラムをデバッグする場合は適度に情報をフィルタリングしながら一つの実行の流れを順に見ていけばよい．しかし，高並列プログラムでは，制御・データの流れが分散して多数存在するので，それらをすべてトレースするのは困難である．その中からいくつかの流れを選択してトレースする手法では，どのように選択するかが問題なうえ，それらの相互関係が理解しづらい．このことから，逐次プログラムと比べて，細粒度高並列プログラムでは状況を把握することがいっそう困難かつ重要であるといえる．

これを解決するためには，まずプログラムの実行の巨視的な状態が見えることが重要であり，実行情報を抽象化したグローバルな視野が必要となる．このとき，実行情報をユーザにいかに見せるかが問題となる．

(2) バグの絞り込み

実行状況をもとにバグがあると思われる部分を大まかに判断したら，その部分をさらに詳しく観察し，最終的には具体的なバグの発生源に到達しなければならない．実行情報は複雑かつ巨大なので，バグに到達するには，抽象度／視野を適度の大きさにして，段階的に詳しく観察していくことでバグの存在範囲を絞り込んでいく必要がある．

効率よくバグを絞り込むには，ユーザに課する情報処理量を抑えつつユーザからの有効な情報を最大限引き出すことが必要である．このためには，並列プログラムの多次元的な実行情報を射影する際にどのような側面が選択できるかが問題である．さらに，この抽象化された情報がユーザにとって理解しやすく表示され，その表示に対してユーザの希望する指示が自由に与えられるかどうかが課題となる．このとき，抽象化の側面の自由度および，多次元的インタフェースが重要な役割を果たす．

(3) 静的情報の把握

通常，デバッグの過程でユーザは以下のように随時ソースプログラムを参照して静的情報を把握している．

- 全体の処理の流れを静的に把握する．
- 実行情報に対応する部分のプログラムを理解する．
- バグの位置をソースプログラム上に発見する．
- ソースプログラムに修正を加える．

コンカレントなプログラムでは同時に見なければならない処理の流れがソースの中に分散している．従来のマルチウインドウデバッガでは，ソースレベルデバッガがウインドウとして各プロセスにつくことにより対処しているが，細粒度高並列プログラムではこの方法を適用することは難しい．

(4) 実行制御

以上のようなデバッグの過程は，プログラムの実行の観察／操作をすることによって行われる．これを実現するためにはプログラムの実行を制御する機構が必要である．逐次プログラムのデバッグにおいては，ブレークポイントを指定してプログラムを停止させ，実行状態を観察するが，並列プログラムには実行の流れが複数存在しており，並列プログラムに対応したブレークポイントが必要である．

また，並列プログラムの動作には非決定性が含まれる．CCL では言語仕様に同期・通信の機能を持ち，値が書き込まれる前に読んでしまうような同期のバグは少なくなるが，非決定的な動作が記述されないわけではない．非決定的にバグが生じるようなプログラムをデバッグする場合には，その動作を制御する機能が必要となるであろう．

4.3 HyperDEBU の機能

前節では，細粒度高並列プログラムのデバッグに要求される機能を論じた．本節では，これを実現した Fleng のデバッガ HyperDEBU の具体的な機能について述べる．

4.3.1 多様な視野を用いたバグの絞り込み

(1) トップレベルウインドウとプロゼスウインドウ

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．実行情報をさら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ポートしており，各種のウインドウに表示された構造データをストラクチャウインドウとして取り出してブラウズすることができる．

（2）　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・側面の自由度を生かしたバグの絞り込みを可能にしている．プロセスモデルの三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

といった三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可能となる．

4.3.2　プログラムの実行の視覚化

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

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/O treeウインドウの上に表現されている．しかし，CCLのような細粒度高並列プログラムでは，個々のゴール・個々のデータをすべて視覚化したのでは繁雑になりすぎる．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．

以降では，制御／データそれぞれについての具体的な視覚化手法について述べる．

（1）　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．

（2）　データの流れ

データの流れについてはI/O tree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 による表示では複雑すぎる．これを解決するために，トップレベルウインドウで特定のデータ依存関係だけ視覚化することにより，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．このため，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．現在われわれは，ユーザの注目するストリームについてその生成・分配・入出力を視覚化する手法を提案し[7]，これをもとに，視覚化機能の主要部分を実装して実験を行っている段階である．

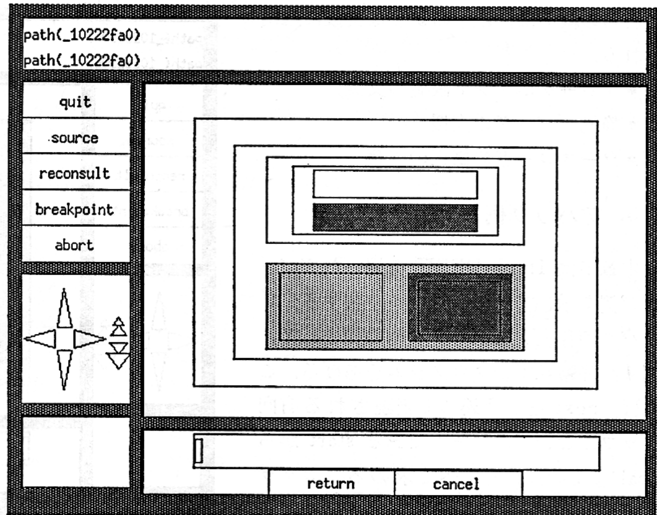

図 3 トップレベルウインドウ
Fig. 3 A toplevel-window.

4.3.3 並列プログラムのためのブレークポイント

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，「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・実行の視覚化・静的デバッグなどに活用することができる．

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ものとしては，現在以下のものがある．

- 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．
- 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．
- 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．
- 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．

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．

4.3.4 プログラムコードのブラウジング

Fleng プログラムの静的情報を把握の支援を行うために，プログラムコードのブラウジングが必要となる．これは，現在簡易版が実装されており，以下のような機能を持つ．

- 述語の呼び出し関係の把握
  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．
- 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，各所の表示に現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される．

- 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

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．

バグの存在するプログラムの例として，**図 4** に経路探索のプログラムをあげる．これは，next という述語で表現される有向グラフにおいて，ノードである start から goal までの経路をすべ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 5** 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 6**）．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

```
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(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].

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

図 4　バグのあるプログラム例
Fig. 4　An example of erroneous programs.

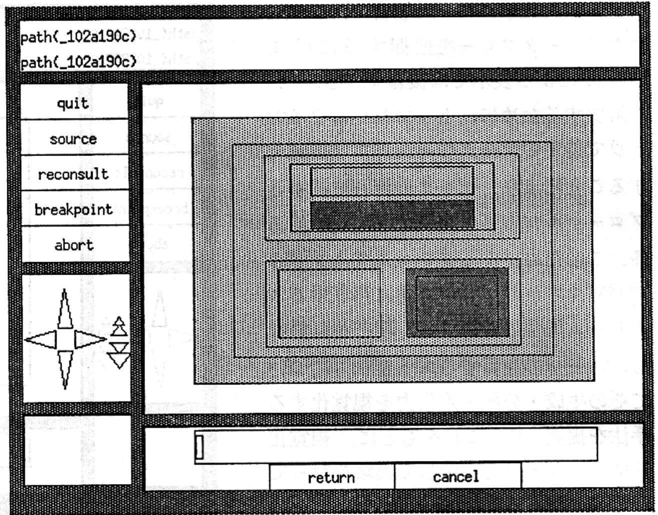

図 5　デバッグ例：場面 1
Fig. 5　Example of debugging : Scene-1.

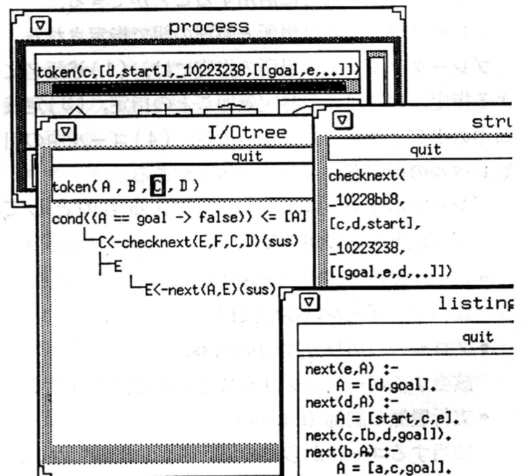

図 6　デバッグ例：場面 2
Fig. 6　Example of debugging : Scene-2.

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．

この例からもわかるように，HyperDEBU では，まずプログラムの視覚化を行って実行の状況を把握し，その上でプロセスウインドウを用いてバグを絞り込んでいくことにより，効果的にバグを発見できる．

6. 考察と課題

6.1 その他の並列言語への拡張性

HyperDEBU は CCL の一つである Fleng を対象に実装されたデバッガである．しかし，Fleng は他の CCL よりも言語仕様が簡潔であるので，HyperDEBU の基本的な設計は一般の CCL に対して適応できる．各言語仕様に即した詳細設計の変更により，他の CCL のデバッガとして拡張することは可能であろう．Full GHC の場合はガード部で OR 並列に実行される部分をいかにとり扱うかといった考察が必要となるが，Flat GHC はガードゴールが組み込み述語のみなので拡張のための大きな問題は存在せず，HyperDEBU を容易に適用することができる．

また，4.1 節の設計思想や 4.2 節のデバッガの機能に対する要求は，一般の高並列な言語に通用する課題なので，並列関数型言語などへの応用も考えられる．

6.2 CCL のデバッガの諸研究との比較

ここでは，Committed-Choice 型言語のデバッガの諸研究と，本デバッガとの比較を行う．

GHC を実用化した言語 KL1 の OS である PIMOS にはデバッグのためのトレーサが備わっている[8]．これは，ゴールの親子関係のみに注目してトレースし，それ以外のゴール実行のトレースを抑制することで，特定の制御の流れのみを観察しようとするものである．しかし，制御の流れが多数ある場合にはその表示はやはり繁雑なものになるであろうと思われる．また，データの流れに関するトレースも必要であろう．PIMOS にはこのほかに，デッドロックしたゴール群を報告する機能が存在するが，それらの依存関係や，それ以前の履歴がなければバグの原因を発見するのは困難であろう．これに対し HyperDEBU では，実行履歴を制御の流れやデータの流れに基づいて観察しながら，グローバルな視野からよりローカルな視野へとバグの存在位置を絞り込んでいくことが可能であり，これによって上記の問題が解決される．一方，PIMOS のデバッガの優れている点は，OS の機能を利用することで実行のオーバヘッドが小さく抑えられていることである．今後の課題として，HyperDEBU でも，実行時のオーバヘッドを抑える機能を処理系がサポートすることが考えられる．

Committed-Choice 型言語の実行をプロセス間通信と見て視覚的にデバッグを行うデバッガとしては，GHC のプロセス指向デバッガ[9]が存在する．しかし，ここでは逐次的なゴールの列をプロセスとして捉えている．このような場合，制御の流れが多数存在するプログラムではその実行の把握が困難であろう．このためには，より抽象的で巨視的な視野からプログラムの動作を把握することが必要と思われる．HyperDEBU は，ゴールの集合を階層的なプロセスとして捉え，これを観察・操作するために抽象化のレベル・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することで，大規模並列プログラムでも効果的にバグを絞り込むことを可能にしている．

6.3 非決定性に対する課題

CCL は同期・通信の機能を言語仕様自体に持つので非決定的な同期のバグは少なくなる．このため非決定的な動作によるバグを扱わなければならない場合はほとんど存在しないといわれている．現在の HyperDEBU では非決定的な動作に対する機能はまだ実現されていないが，実用上はほとんど問題になっていない．しかし，CCL で非決定的な動作を記述することは可能であるから非決定的に生じるバグは存在しうる．ここでは，このようなバグに対応するための課題について考察する．

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うるかどうかはよ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．

6.4 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 は実用規模のプログラムのデバッグに実際に用いられているが，実行速度・使用メモリの大きさに不満が残る．

実行時間のオーバヘッドに関して改良すべき点は，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部分である．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，履歴情報をいか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．

謝辞 東京大学工学部田中研究室の諸氏には，デバッガのユーザとして貴重なご意見をいただきました．なお，本研究は文部省特別推進研究 No. 62065002 の一環として行われた．

参 考 文 献

1) Mcdowell, C.E. and Helmbold, D.P.: Debugging Concurrent Programs, *ACM Comput. Surv.*, 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*, pp. 1031-1040 (1988).
3) 小池，田中：並列推論エンジン PIE 64, bit 臨時増刊並列コンピュータアーキテクチャ，Vol. 21, No. 4, pp. 488-497 (1989).
4) Murakami, M.: A Declarative Semantics of Flat Guarded Horn Clause for Programs with Perpetual Processes, *Theor. Comput. Sci.*, Vol. 75, No. 1/2, pp. 67-83 (1990).
5) Takeuchi, A.: Towards a Semantic Model of GHC, Tech. Rep. of IECE, Comp., 86-59 (1986).
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*, pp. 133-142 (1989).
7) 舘村，小池，田中：細粒度高並列プログラムの実行の視覚化，情報処理学会研究報告，91-ARC-89-15, pp. 103-110 (1991).
8) 中尾，近山，中島，大西：PIMOS のトレーサ，第 39 回情報処理学会全国大会論文集，6 N-3, pp. 1135-1136 (1989).

9) 前田，魚井，都倉：GHC 上のプロセス指向デバッガ，*Proceedings of the Logic Programming Conference '90*, pp. 169-178 (1990).

(平成3年 8 月6日受付)
(平成3年12月9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．