計算機アーキテクチャ 83−33
（1990. 7. 18）

PIE64 のマルチウインドウデバッガ HyperDEBU における並列プログラムの実行制御

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

概要

並列推論エンジン PIE64 上の並列論理型言語 Fleng は、各ゴールがデータの依存関係によって同期をとりながらそれぞれリダクションされていくファイングレインな並列言語である。我々は Fleng プログラムの実行の様子をモデル化・抽象化し、このモデルを通してプログラムを観察・制御するマルチウインドウデバッガ HyperDEBU を開発した。本論文ではこの HyperDEBU について概説し、特にデバッガに必要とされるゴールの管理・実行制御機能について述べる。

Multiwindow debugger HyperDEBU and its control mechanism for parallel programs

Jun-ichi Tatemura, Hanpei Koike, Hidehiko Tanaka
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

Abstract

This paper describes the design and implementation of HyperDEBU, a multiwindow debugger for Fleng which is a parallel logic programming language on Parallel Inference Engine PIE64. Fleng is a fine grained language, which goals are executed in parallel synchronizing by data causality. We propose a view which shows an execution of a Fleng program. With this view, a programmer can examine and manipulate a Fleng program on HyperDEBU. Then a control mechanism for Fleng goals is described.

1 はじめに

並列推論エンジン PIE64 上の並列論理型言語 Fleng は、各ゴールがデータの依存関係によって同期をとりながらそれぞれリダクションされていくファイングレインな並列言語である。我々は Fleng プログラムの実行の様子をモデル化・抽象化し、このモデルを通してプログラムを観察・制御するマルチウインドウデバッガ HyperDEBU を開発した。本論文ではこの HyperDEBU について概説し、特にデバッガに必要とされるゴールの管理・実行制御機能について述べる。

2 並列論理型言語 Fleng

我々の研究室で開発している並列計算機 PIE64 で採用されている言語は Fleng [1] という Committed-Choice 型言語 (以下 CCL と呼ぶ) である。

Concurrent Prolog や GHC などの 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。我々の研究室で開発した言語 Fleng は GHC など他の 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。このように言語を単純化し実装容易性・柔軟性を増すことによって Fleng は並列記号処理計算機の核言語として適したものとなっている。その反面、言語の可読性・記述性は低下するが、拡張構文を用いてこれに対応する。

並列論理型言語においてはその並列実行の単位が一つのゴールであり、各々が並列にリダクションされ新しいゴールを生成する。このようなファイングレインな言語では、プロセス毎に逐次用のデバッガを割り当てるといった従来のデバッグ手法は用いることができない。また、ゴール一個単位でトレースをしてもそこからプログラムの実行の様子を掴むことは容易ではなく、何らかの抽象化が必要となろう。

3 Fleng プログラム実行のモデル化

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。ユーザは与えられたモデルと自分の意図するモデルとを比較することによってバグを発見する。我々は 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。本節ではこのプロセスモデルについて説明する。

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。

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、「G に対応するプロセス」ということにする。これ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、その引数 (変数) が外部との通信の窓口になる。S はプロセスの状態 (terminate / suspend / active) を表す。G_{ins} はゴールの *instance* である。

I,O は $Input/Output$ であり、これがプロセスの入出力を表す。G_{skel} の変数がプロセスの外部及び内部からどのように具体化されていくかを示したものである。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、その入出力の関係も決められる。これによりプロセスの入出力は再帰的に定義できる [2]。

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。よって 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。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。

以上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は充分なモデルといえるが、これがユーザにとって良いモデルを与えている必要がある。プログラマが Fleng のような CCL のプログラムの動作を考える時は、個々のゴールの動作を把握しているというより、適度な大きさのゴールの集合として抽象化して捉えていると思われる。実際、論理型言語で述語を定義していく過程はこの抽象化を行なう過程と見ることができる。幾つかのプロセスによってより抽象度の高いプロセスが構成されるというモデルはこのようなプログラムの見方に適しているといえよう。シーケンシャルなゴールの列をプロセスと見る見方もあるが、これではシーケンシャルな列が多数あるような場合には向かない。

4 マルチウインドウデバッガ HyperDEBU

我々はプロセスモデルを用いたデバッガ DEBU を試作した [3] 。しかし、これはインターフェースが一次元的であり、プログラムとユーザとの間がボトルネックとなり、思うようにデバッグができない。並列プログラムの実行を表示するのにも操作するのにも多次元的なインターフェースが必要と感じられた。このため、UNIX 上の Fleng 処理系 (インタプリタ) に X-window インタフェースをつけ、マルチウインドウ化したデバッガ HyperDEBU の試作版を開発した。

本節では HyperDEBU の機能・特徴について述べる。

4.1 並列プログラムのデバッグに必要な機能

まず、一般に並列プログラムをデバッグするにはどのような機能が必要となってくるかを挙げてみる。

- 非決定的動作への対応
  並列プログラムには実行のタイミングによってその結果が違ってくるといった非決定性が存在する。ここで特に問題になることは、デバッガの介入が非決定的動作に影響し、バグ発生時の動作が再現できないといった probe effect が

発生する可能性があることである。しかし、CCL の場合はデータの依存関係を記述する言語といえるので実行のタイミングによる非決定的動作は少ない。よって、この影響はあまり気にしなくても良い。また、CCL における非決定性は定義節のコミット、ユニフィケーションの競合に帰着されるので、ここに着目すれば非決定的動作に対応することができよう。

- 多次元的表示機能
  並列プログラムの実行は一直線ではなく、制御の流れもデータも分散している。このようなプログラムの実行の様子を表現するのには一次元的表示では不十分である。
- イベント履歴
  並列に発生する多くのイベントをただ表示するだけでなく履歴を記録することが必要となる。これは巨大になるのでブラウジングの機能が必要であろう。また、履歴のためのメモリも無視できない。CCL では履歴として記録するものは実行されたゴールを親子関係で表した計算木であろう。
- 抽象化
  一般には各プロセスのイベントをより抽象化されたレベルで扱うことなどが挙げられるが、ファイングレインな並列プログラムでは特に並列プロセスの数が多いので、多数のプロセスを抽象化するようなマクロ的な視野を提供する必要がある。

4.2 HyperDEBU の特徴

前述したような並列プログラムのデバッグのための要求を考慮しながら、HyperDEBU の特徴について述べる。

近年、ハイパーテキストの研究・開発が盛んであるが、これは従来直線的にしか表されなかったテキストを多次元的に表現することができるものである。HyperDEBU は並列プログラムのデバッグにこれと同様なアプローチを導入し、プログラムの実行を多次元的に表現するデバッガである。これは、従来のマルチウインドウデバッガのように各プロセスに逐次用のデバッガを割り当てるのではなく、制御 / データの流れが形成する複雑なリンク構造を観察するための多様な視野としてウインドウを用いる。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ウインドウを開いてゆく。また、このようにして表現されたモデルを通して、逆にユーザ側からプログラムの実行を操作することができる。HyperDEBU の目標とする所は単にバグを取るためのデバッガではなく、プログラムとユーザとのインターフェース (プログラム自身とユーザとのインターフェースでなく、プログラムをメタレベルから見たインターフェース) を提供するハイパーデバッガである。

これには、プロセスモデルによって得られた以下の特徴が役だっている。

- 抽象化の側面の自由度
- 抽象化のレベルの自由度

一般に幾つかの逐次プロセスからなる並列プログラムでは、その実行を視覚的に表現する場合、抽象化の側面として

- 各プロセスを時間軸にそって表示する。
- スナップショットをとって時間毎のアニメーションにする。

といった複数の切口がある。一方、CCL では他の並列言語に対して次のような特徴を持つ。

- ファイングレインなのでプロセス (ゴール) が多数生成・消滅する。
- データフロー的なので時間軸に重大な意味はない。

これを考慮すると抽象化の側面として、(一つのプロセス内の並列実行においては) 計算木・ゴールの集合が挙げられる。HyperDEBU ではプロセスに対するこのような見方をそれぞれサポートしている。

並列プログラムのデバッグでは各プロセスの内部状態だけでなく、プロセス間レベルでの実行の様子も重要である。HyperDEBU ではプロセス間レベルの見え方として各プロセスの外部に対する入出力が見える。この機能により、プロセスが引数を通じて通信している様子が分かる。HyperDEBU ではプロセス間レベルとプロセス内レベルの切り方に自由度があり、プロセス内も更にプロセスに分割してプロセス間レベルとして扱うこともできる。

同様なデバッガとしては、GHC のプログラムの実行をストリーム通信をする逐次的プロセスとしてとらえ、これを視覚的に表示するデバッガが研究されている [4]。ただしこれはプロセス間通信がリストを用いたストリーム通信に限定されるので、プログラム全般に対応しているわけではない。また、concurrency の数だけプロセスが表示されるので、プログラムが大きくなった場合に、よりマクロ的な視野を与えて抽象化することが必要となろう。これは再帰的なゴールなどの一列のゴールを一つのプロセスと見た逐次プロセスを扱っているためであるが、これに対して HyperDEBU ではいくつかのサブプロセスが集まってまた一つのプロセスとして捉えられる。また、実際的なデバッガとしては、逆によりミクロな視野も必要であろう。HyperDEBU ではゴール単位のトレース、及びデータのブラウザ / インスペクタを備えている。

GHC プログラム実行の可視化の研究としては VISTA も挙げられる [5]。このような可視化のためにはユーザからの付加情報が必要である。VISTA においても述語ごとに描画指示データを与えたり、描画コマンドをプログラムに挿入する必要がある。HyperDEBU では付加情報を用意しなくてもプログラムの意味だけから一般的なプロセスとしてウインドウ上に表現し、ユーザからの指示によって多様な視野を提供する。しかし、可視化情報を与える手法はプログラム実行の理解には大変役立つので、HyperDEBU としても今後の課題として、プロセスウインドウに各種の view を用意し、ユーザがプログラム以外に記述した情報からこれをよりわかり易く可視化することを検討している。

4.3 HyperDEBU の機能

HyperDEBU の構成は以下のようになる。

- プロセスウインドウ
  任意の一つのプロセスに割り当てることができる。サブウインドウとして、以下に挙げる TREE、GOALS、I/O tree ウインドウを持つ。これらはプロセスに対して多様な

見方を提供する。

- TREE ウインドウ
  プロセスを計算木としてみた場合に対応する。部分木をサブプロセスとして取り出すことができる。
- GOALS ウインドウ
  プロセスをゴールの集合としてみた場合に対応する。
- I/O tree ウインドウ
  入出力因果関係を木構造として表示する。計算木とコミットされた定義節により生成される。プロセスが通信している様子がわかる。CCL の非決定性は定義節のコミットと、競合するユニフィケーションであるが、それぞれ因果関係の OR 木、入出力のパスの競合という形で表せるであろう。

- データのブラウザ / インスペクタ
  複雑なデータのブラウジングをするウインドウと、そこからデータの一部を切り出してそれがバインドされていく様子を観察するウインドウがある。
- トップレベルウインドウ
  ユーザがトップレベルのゴールを投入するコンソールで、プロセスウインドウを操作・管理する。プロセスを画面にマッピングして生成されていく様子を図示することによってよりマクロ的な (グローバルな) 視野を提供する。(現在はまだ開発中である。)

また、各ウインドウの表示は未定義変数が具体化されるのに応じて動的に変化する。

各機能の詳細・実現方式はここでは省略し、次節ではこれらの機能を満たすようにプログラムの実行をどう制御するかについて述べる。

5 プロセスモデルに基づいたゴール管理

デバッガはプログラムに実行の様子をユーザに示し、ユーザの指示通りにプログラムの実行を制御するわけであるから、デバッガにはゴールを管理する機能が必要になる。我々は、HyperDEBU の機能を満たすようなゴール管理機構として、以下に述べるようなゴールキュー (ゴールプールと言った方が正しいが) を用い、これにより「プロセス」を実現した。その特徴は次の 2 点が挙げられる。

- 階層的
  計算木が部分木を内部に持つように、プロセスも内部にサブプロセスを持つ。これに対応するためにゴールキューは階層的になる必要がある。
- 仮想的
  PIE において実行する時の実際のゴールキューとは独立であり、スケジューリング・負荷分散といった要素は抽象化される。これによりキューの内部も並列に実行されうる。HyperDEBU のゴールキューは concurrency に関係し、実際の IU のゴールキューは並列度に関係する。しかし、並列度と concurrency とは別のものであり、HyperDEBU のゴールキューをそれぞれ逐次的に実行するのは必ずしも得策ではない。ファイングレインなプログラムの実行の柔軟性を活かし、効率に実行するためにはスケジューリングなどに自由度を残す方が良いと考えた。

このゴールキューのインターフェースは

```
call(Goal,CommandStream,StatusStream)
```

という述語で実現される。第一引数はキューに与えるゴール、第二引数はキューにコマンドを送るストリーム、第三引数はキューから状態情報が送られるストリームである。それぞれのストリームの機能は以下のようになる。

CommandStream :

- ブレークポイントの設定。
  `break(BreakPoints)`
- 実行状態の変更。
  `go / stop / pause`
- ゴールのリストを得る。
  `goals(ActiveGoals,SuspendGoals,Modified)`
- 計算木を得る。
  `tree(Tree,ReturnedTree)`
- ストリームを閉じることにより終了。
  `CommandStream = []`

StatusStream :

- プロセス全体としての状態変化。
  `suspend / activate / terminate`
- ゴールのフェイルなどのエラー。
  `failed(Goal)` など。
- 終了時に残ったゴールを返して閉じられる。
  `closed(ActiveGoals,SuspendGoals,Tree)`

同様なゴール管理機構としては、KL1 の OS である PIMOS の「荘園」が挙げられる。これは、メタ機能の形をしている点、ストリームで通信する点は同じであるが、その目的・機能が多少異なっている。「荘園」はまず最初に OS として資源管理が必要であるという立場から導入されたものであり、おもに資源管理と例外処理の機能を実現している。「荘園」にもデバッグをサポートする機能があり、これを用いてトレーサが開発されている [6]。これは実行されたゴールをフィルタリングして表示する機能を持つ。しかし、並列プログラムのデバッガとしては実行の中の一つの流れだけを表示しているだけでは不十分である。抽象化のレベルがゴールのリダクションに限られていて、プロセス間通信などの情報が把握できない。

一方、HyperDEBU のゴールキューは、プログラミング環境としてのデバッガのゴール管理として導入したものである。その結果として機能的な差異が生まれることとなった。

以下では、このゴール管理機構の特徴的な機能について述べる。

5.1 プロセスの階層

プロセスの中にサブプロセスが定義できるというモデルを実現するためには、キューの中にキューを階層的に作り、それぞれにプロセスウインドウを割り当てることが必要となってくる。以下ではこの階層的なキューをどのように構成するかについて述べる。

プロセスの外側から見た場合、内部のサブプロセスは抽象化されていなければならないので、キューの中にキューを作って

も、これが外側に対しては原則的に影響を与えないことが必要とされる。このためには、外部からの操作と外部に見える状態が内部にサブプロセスウインドウ(キュー)を割り当てても変化しないことが必要である。すなわち、

- 操作 (go /stop, ブレークポイント) が上から下へ
- 状態 (suspend / activate / terminate) が下から上へ

反映されることが必要である。

ただし、プロセスウインドウに対して、サブプロセスウインドウが完全に従属するのではなく、ユーザからの指定により独立な操作を与えられる方が望ましい。このため、以下の機能を導入する。

1. 上からの go /stop の遮断
   stop の状態に pause /stop の区別をつけることで実現する。 pause の場合は上(親プロセス)から go と言う操作が伝わった場合、 go の状態になって実行を開始するが、 stop の場合は stop のままで、サブプロセス自身のウインドウによって直接操作を受けない限り停止したままである。
2. ブレークポイントの独立設定
   デフォルトではブレークポイントは上から継承されるものとする。

これらの機構により、細かい実行の操作が可能となる。

以上のようなプロセスの階層構造は図1のようになる。親 (左側) のキュー Q からは、 G' を通してサブプロセスが見え、通常のゴールと同様に扱われる。この G' とサブプロセスに対応する Q 、およびプロセスウインドウ W が通信し合いながら状態 / 操作の反映が実現される。

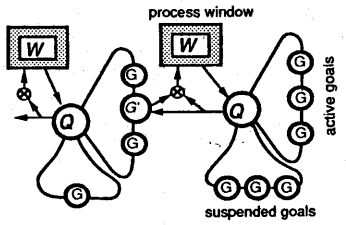

図 1: プロセスの階層構造

5.2 計算木としての観察 / 操作

HyperDEBU では計算木を実行履歴として残してブラウジングできる。このとき、以下のような機能が望まれる。

1. 実行による表示の変化の対応
   リダクションが起きて実行が進んだら、それを計算木に反映しなければならない。木の末端のアクティブなゴールの状態も反映する必要がある。
2. 計算木からのゴール操作
   計算木の中からサブプロセスを切り出したい。

このような要求を満たすために、ゴールとそれに対応する計算木の末端とは通信させる必要がある。そこで、計算木の末端にはストリームを埋め込んでおく。ゴールはリダクションされるとストリームを閉じて計算木を束縛する。計算木にはストリームが埋め込まれているので、キューから計算木を得た場合には使用後の計算木を返さなければならない。このため、キューから計算木を得るコマンドは前述のように2引数になっている。

5.3 ブレークポイント

従来の逐次型プログラムのデバッグ手法は cyclical debugging といわれるもので、実行時にプログラムを中断して状態を調べ、続行したり再実行したりすることを繰り返すことによりバグを発見する。

しかし、並列プログラムでは実行の流れが一本ではない。並列言語ではブレークポイントにたどり着いた場合に次の二通りの動作が考えられる。

1. プログラム全体が停止する
2. その流れ(プロセス)だけ停止する

前者は少ないオーバヘッドで実現するのは困難である。後者は、通常の並列言語では一つのプロセスだけ止めて他のプロセスが勝手に動いていると実行結果に重大な影響を与えうる。しかし、 Fleng のような CCL ではデータの依存関係によって実行が制御されているので、サスペンドが起こるといった実行過程は変化しても、実行結果にはそれほど影響を与えることはない。 CCL は一種のデータフロー言語であるから、止めたい場所だけ止めれば、そこに依存する部分は止まってくれるのでブレークポイントに達した瞬間に全体をとめる必要性はその手間に較べると大きくない。よって HyperDEBU では後者を選ぶことにする。

HyperDEBU におけるブレークポイントとは、単なる実行の停止ではなく、ブレークポイントに達したゴールに対して処理系がどのような操作を行なうかを記述したもので、「場所」と「処理」の組の集合になる。この「場所」と「処理」にはそれぞれ次のようなものがある。

1. ブレークポイントとなる場所
   - 述語単位
     述語名で指定 / 述語名 + 引数で指定という二通りが考えられる。
   - 定義節単位
     一般に一つの述語に対しては複数の定義節が定義されている。この内のどれが選ばれたかによって区別したい場合も考えられる。
   - ボディーゴール単位
     どのゴールから呼ばれたのか区別したい場合、呼びだし側のボディーゴールで指定することが考えられる。
2. ブレークポイントにおける処理
   - 停止 (stop / pause)
     サブプロセスの場合と同様に、 stop は再度実行が命令されても停止しているが、 pause は実行を再開する。 pause を指定することによって例えば再帰的なプログラムで一サイクル毎のステップ実行を行なうこともできよう。
   - 新しいキュー (サブプロセス)

- 計算木記録モードの解除
  実行効率（おもにメモリ効率）のため、正しいとわかっている述語など必要のない部分は履歴をとらない方が良い。

ブレークポイントはプロセス(キュー)毎に独立に設定できることが望ましい。

6 試作版の実装

HyperDEBU 試作版はゴールキューの部分をメタインタプリタで実現した。

各ゴールは goal(Goal,...) というゴールで管理され、これらをショートサーキットと呼ばれる手法でつないでアクティブなゴールのリンクとサスペンドしたゴールのリンクにして、 que(Active, Suspend, ...) に管理させる。この que がコマンドを受け取るストリームと状態情報を送るストリームを持つことによって call(Goal, CommandStream, StatusStream) を実現している。

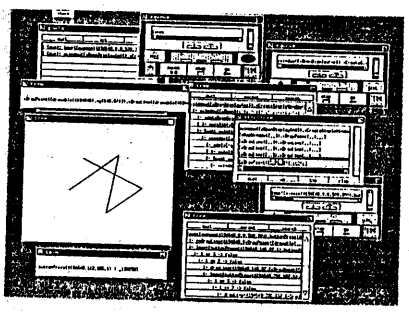

図 2: HyperDEBU の画面

また、メタインタプリタでは実行効率が良くないので、Fleng インタプリタ処理系（UNIX 上）にデバッグサポート機能をつけることを検討中である。

7 PIE64 での実装

実際の PIE での処理系ではデバッガをサポートするための機能が必要である。以下ではどのような機能を用意する必要があるかについて述べる。

PIE64 の推論ユニットでは、次の三つのプロセッサが協調動作している。

- UNIRED（推論プロセッサ）：Fleng のプログラムを実行するための専用プロセッサ。ユニフィケーション、リダクションを効率良く実行する命令セットを持っている。
- NIP（ネットワークインタフェースプロセッサ）：相互結合網とのインタフェースを持ち、推論ユニット間の通信処理と Fleng プログラムのプロセス間同期処理を行なうプロセッサ。
- SPARC（管理プロセッサ）：ゴール管理・メモリ管理・浮動小数点演算・入出力処理などの様々な処理を行なうための汎用マイクロプロセッサ。

HyperDEBU のゴール管理機構を PIE で実現するには UNIRED、SPARC、コンパイラでその役割を分担する必要がある。

各ゴールには付加情報がつけられる。これには、キューへのポインタ、キューからのストリーム、ショートサーキット用のリンク、ブレークポイントの情報などである。ゴールのリダクション時には子ゴールへのリンクの分配、計算木の生成などの処理を行なう。また、アクティベート / サスペンド時や、フェイルなどの例外処理にはキューへのポインタを用いる。これらは、コンパイルされたコードに付加的な処理を埋め込んで、必要に応じて SPARC のルーチンを呼び出すことによって実現する。

ゴールキューは一つのゴールとして実現される。普段はサスペンドしているが、コマンドストリームとショートサーキットでアクティベートされ、それぞれに応じた処理をする。ゴールのアクティベート / サスペンド時には各ゴールによってリンクが破壊的に書き換えられる。また、各ゴールにブロードキャストするためのストリームを持つ。

ブレークポイントは場所と処理の組を集めたベクタをキュー内の各ゴールが共有することで実現する。ソースコード（定義節）との対応をとるためにコンパイル時にコードにブレークポイントとなるところを埋め込む必要がある。

8 おわりに

現在、HyperDEBU ではゴールのスケジューリングなどの要因が抽象化された見え方を提供しているが、これに加えて負荷分散、スケジューリングの制御を明示的に扱うような見え方をサポートすることが今後の課題として考えられる。これは例えば並列度の評価（プログラム自体の並列度の評価、負荷分散・スケジューリングを変えた場合の評価）等に応用できるであろう。

なお、本研究は文部省特別推進研究 No.62065002 による。

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] 舘村純一, 田中英彦：“並列論理型言語 FLENG のデバッガ”, Logic Programming Conference '89, 1989.

[3] 舘村純一, 田中英彦：“Committed-Choice 型言語 FLENG のデバッガ DEBU”, 第 39 回情報処理学会全国大会 6N-4, 1989 年 10 月, pp.1137-1138.

[4] 前田宗則, 魚井宏高, 都倉信樹：“GHC 上のプロセス指向デバッガ”, Logic Programing Conference '90, 1990.

[5] 奥村晃, 他：“並列プログラム変換 / 可視化システム VISTA における可視化技術について”, ICOT Technical Report TR-464, 1989.

[6] 中尾浩一, 近山隆, 中島克人, 大西論：“PIMOS のトレーサ”, 第 39 回情報処理学会全国大会 6N-3, 1989 年 10 月, pp.1135-1136.