2E−9

並列論理型言語 Fleng のデバッガ HyperDEBU における入出力トレース

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*

1 はじめに

並列論理型プログラムは、複数のゴールが互いに同期を取り合いながら並列にリダクションされていくので、逐次論理型プログラムと同様な手法でデバッグするのには限界がある。これはデバッガがプログラムに実行の様子を従来のように直線的に示すのでは不十分であるためである。これを解決するには、並列論理型言語に適した実行モデルを用いなければならない。

Fleng は我々の研究室で開発された Committed-Choice 型言語であるが、このプログラムをデバッグするために、実行の様子を通信し合うプロセスとしてモデル化することを提案した [2]。

HyperDEBU は、Fleng プログラムをウインドウ環境の上にモデル化することによって、実行の様子をユーザにわかりやすく示し、また、実行に対してユーザーが視覚的に制御を行なえることを目指したものである [1]。

ここでは、HyperDEBU においてどのようにバグを探索していくことが可能か、そのデバッグ手法について述べる。

2 プロセスモデルによるデバッギング

ここで言う「プロセス」とは、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全てのゴールからなる集合をさす (これを「G に対応するプロセス」という)。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り、半直線的な入出力因果関係が木構造のように形成されるので、I/O tree とも呼ぶことにする。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]。

[2] ではこのプロセスモデルを用いたアルゴリズミックデバッギングを提案した。これはデバッガが提示したインスタンスの正誤をユーザが答える際にプロセス間の通信の因果関係を考慮することにより、従来では発見できないバグにも対応できるものである。しかし、これはユーザ側からの答えが「正しいか間違っているか」という情報だけで、ユーザに与えられた一つの質問が複雑であるわりにデバッガに与えられる情報が少ない。デバッガはこの情報を用いて計算木を探索するだけなので、プロセスの通信のどの部分が間違っているかなどの情報はバグ探索には十分いかされていない。

Fleng プログラムの実行は全体として互いに対応付けられた計算木と入出力の木によって表されるとも言えるわけであるから、バグのある場所を絞り込む場合もこの両者を用いることが望ましい。HyperDEBU はマルチウインドウ上にプログラムを「プロセス」として抽象化しつつ、このリンクされた二つの木の中でのバグ探索を支援する。

3 HyperDEBU のウインドウ構成

HyperDEBU ではプログラム実行中の任意のゴールに対応するプロセスに対して一つのウインドウを割り当てることができる。これを「プロセス・ウインドウ」と呼ぶ。プロセスは、(1) ゴールの集合、(2) 計算木中の部分木、(3) 入出力因果関係による I/O tree という見方ができるのでこれに対応して、

- ゴールサブウインドウ
- 計算木サブウインドウ
- I/O tree サブウインドウ

といったサブウインドウを持つ。また、他に複雑なデータ構造を見るためのインスタンスウインドウなども用意される。

4 I/O tree の表示

I/O tree は入力の木、出力の木があり、それぞれがプロセスの G_{skel} の引数一つ一つに対して存在する。図 1 は I/O tree の簡単な例である。

I/O tree を形作るのはコミットされた定義節であり、これによって I/O tree と計算木は対応付けられる (図 2)。

5 I/O tree の抽象化

プログラムの規模が大きくなるに従って、I/O tree も大きく複雑になる。そのため、これを簡略化して表示する必要がある。これには、以下のような方法が挙げられる。

- 入出力因果関係を保存しながら縮退する。
- I/O tree の一部をサブプロセスとして抽象化する。

前者は [2] に述べられている方法である。それに対して後者はプロセス中のあるゴールをシステム組み込み述語にみなしたのと同様なものである (図 3)。

*Input/Output Tracing on HyperDEBU, Debugger for Fleng Programs
Junichi TATEMURA, Hanpei KOIKE, Hidehiko TANAKA,
the University of Tokyo

HyperDEBU では、この方法を基本にして I/O tree を表示する。これを更に詳しく見たい時は、抽象化された部分を I/O tree で表現するよう指示するか、またはそのサブプロセスをウインドウとして切り出す。

6 HyperDEBU におけるトレース

HyperDEBU においてバグを探索する過程は、バグを含むプロセスを絞り込んでいくことと見ることができる (図 4)。

プロセスの内部を調べる時、I/O tree の中で、間違ったパスが特定できればそこに対応する定義節にバグがあるということがわかる。入出力が複雑過ぎて間違った部分を正確に特定できなければ、サブプロセスとして抽象化し、その上で間違った部分を探せばよい。もしそのサブプロセスの部分が間違っているとすれば、それをまたバグを含むプロセスとして探索を繰り返せばよいであろう。また、バグを含むサブプロセスを絞り込むもう一つの手段として、プロセス内部を計算木としてみることもできる。これらを可能にするため、HyperDEBU では、I/O tree や、計算木の中からサブプロセスをウインドウとして切り出すことができる。

このようにして、入出力、または計算木の間違った部分を特定することにより、バグのある定義節を発見できる。

7 今後の課題

HyperDEBU ではバグを探索する場合にユーザが指示をしてウインドウを開くことになっているが、これをアルゴリズミックに行なうことが考えられる。入出力の間違った部分もしくは間違ったサブプロセスをユーザが指摘していくうちに、どの定義節のどの部分が間違っているかをデバッガが示すことができるであろう。

8 おわりに

現在、HyperDEBU は、UNIX 上の疑似並列処理系において開発中であり、x-window インターフェースを用いて Fleng 自身で記述されている。これは、我々の研究室で開発中の並列推論エンジン PIE64 上に実装される予定である。なお、本研究は文部省特別推進研究 No.6265002 による。

参考文献

[1] 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 ”, 第 40 回情報処理全国大会.

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


```
a([A|B],C,D) :- D = [A|E], a(B,C,E).
a([],C,D) :- C = D.
Top Goal: a([1,2],[3,4],R)
Gskel:a(X,Y,Z)
Output of Z : Oz
 |
-+-< X = [A1|B1] / [1,2] >
 |- Z = [A1|E1]
   -+-< B1 = [A2|B2] / [2] >
    |-E1 = [A2|E2]
      -+-< B2 = [] / [] >
       |- E2 = Y
```

図 1: I/O tree の例

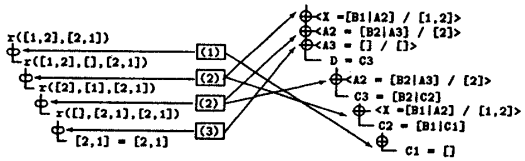

図 2: I/O tree と計算木の対応

```
a([f(X)|L]) :- X = true.
b(L,R) :- L = [f(X)|L1], c(X,R).
c(true,R) :- R = true.
t(R) :- a(L), b(L,R).
Gskel: t(R)
Output of R: Or
  |
 -+-< A = true / true >
  |  <- [f(A)|B] = L
  |      -+-< L = [f(A)|B] / [f(true)|_] >
  |       |- A = true
  |- R = true.
     (abstraction =>)
  |
 -+-< A = true / true >
  |  <- [f(A)|B] = L
  |       |- a(L)
  |- R = true.
```

図 3: I/O tree の簡略化例

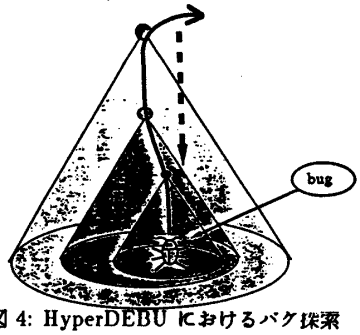

図 4: HyperDEBU におけるバグ探索