ソフトウェア工学 58－13
（１９８８．２．５）

コントロールフローに着目したデバッグ手法とその実現について

齊藤　明紀　　辻野　嘉宏　　都倉　信樹
大阪大学基礎工学部情報工学科

従来のデバッガではプログラムの静的な構造であるソースあるいはオブジェクトコードに基づいてデバッグ作業を行う．一方，バグによる異常動作は動的な存在である．

本報告ではプログラムの動的なコントロールフローに基づいてデバッグ作業を行うコントロールフロー指向デバッガを提案し，評価する．このデバッガは，利用者に動的イメージを表示することで，プログラムの内部の振舞いの理解を容易にする．また，プログラムコードに代えてプログラムの計算状況に対してブレークポイント設定することにより観察したい局面を一意に指定することを可能としている．

A proposal on Control Flow Oriented Debugger

Akinori SAITOH, Yoshihiro TSUJINO and Nobuki TOKURA
Department of information and computer sciences, Faculty of Engineering science, Osaka University
1-1, machikaneyama, toyonaka, Osaka, 560 Japan

Debugging processes with traditional debuggers are based on source or object program code, which is a static image of the target program execution. But bug behavior of the program is of dynamic nature.

In this paper, we propose and evaluate a debugger which is control flow oriented.

1. It displays a dynamic control flow graph of target program execution. This enables to understand the behavior of the target program easily.
2. It allows to access any scene in the target program execution by pointing a vertex, which corresponds to the scene uniquely, on the control flow graph.

1. はじめに

プログラム作成の過程においてデバッグはそれに費やされる労力という観点からみて大きな比重を占めている．そして試行錯誤に頼る部分が多く，作業を行うものの熟練度に大きく左右される作業でもある．

設計ミス，コーディングミスなどが原因となってプログラムコード（オブジェクトコードやソースコード）中に作り込まれた誤りを“バグの原因”，これによってプログラムの異常な動作が引き起こされることを“バグの発生”と呼ぶことにする．バグが発生すると，それ以降プログラムの正常な部分も“バグの影響”により異常な動作をする．そして，バグの影響が最終的に人間に認識できるかたちで現われることを“バグの発現”と呼ぶことにする．これらの中で，バグの原因はプログラムコードの中にあり，静的な存在である．残りの３つはプログラムの入力や動作環境に依存した動的なものである．そしてこれらの総称を“バグ”と呼ぶ．

デバッグ作業は一般に次のような手順を繰り返して行われる．

1) テストや実際の使用で，バグの発現を検出する．
2) バグの原因を捜す．
3) プログラムを修正する．あるいはプログラムの設計を見直す．
4) コンパイル．

この中で，いちばん手間がかかるのは2)のバグの原因の探索である．これは，バグの原因がソースプログラムのどこにあり，どのような条件で発現するのかを突き止める作業のことである．その結果，3)で修正すべきコーディングミスや，設計の誤りがどこにあるのかが判明する．デバッガは主にこのバグすなわち異常動作の原因究明のために使われる．

従来のデバッガでは，プログラムコードに基づいてデバッグ作業を行う．本論文では，従来のデバッガの問題点を指摘し，それを解決するコントロールフロー指向デバッガについて述べる．

2. 従来のデバッグ方法の特徴と問題点

2.1 従来のデバッグ方法

バグの原因の探索の手法は静的な解析と動的な解析の２つに大きく分類できる．

（Ａ） プログラムの静的な解析

ｌｉｎｔ［１］，クロスリファレンサのようなプログラム解析ツールを使ってバグを捜す．プログラマがソースプログラムを読むというのもこれに属する．コーディング終了直後に多発する構文エラーなどの単純なバグはこのような静的な解析で発見される．しかし，これには限界がある．静的解析で発見できるバグを取り尽くした後は，動的な解析手法を用いることになる．

（Ｂ） プログラムの動的な解析

ターゲットプログラムにバグが発現するような入力を与え，どのように動作するのかを調べる．

これはさらに二つに分類できる．

（Ｂ１） プログラムを少しづつ動かして状態の変化を調べ，バグの発現するありさまを観察する．

デバッガを用いたデバッグがこれである．デバッガを用いると，ターゲットプログラムの実行状況を非常に詳しく観察することができる．プログラムの実行全体を詳しく観察するのでは，手間がかかりすぎ，また全体のふるまいの概略を観察することは不得手である．バグの影響を受けている部分だけをうまく観察できるかどうかが作業効率を左右する．

（Ｂ２） プログラムの実行状況を出力させながら実行し，それを解析する．

プログラムに変数の値や手続きの起動，終了を示す診断メッセージを出力する文を埋め込む方法がこれである．この手法では，プログラムの実行全体を調べることができる．しかし，実行状況の詳しいことは分からない．詳しいデータを取ろうとすればするほど余分な診断メッセージの量も増大してしまい，解析が困難になってゆく．

2.2 従来のデバッガの問題点

デバッガで行うのはプログラムの動的な，つまりある特定の入力を与えたときのプログラムのふるまいの解析である．

デバッガの機能は大きく分けて，

1. 連続実行
2. ステップ実行
3. 停止（ブレークポイントによる）
4. プログラムの状態の観察（メモリ読みだし，変数値参照）
5. プログラムの状態の変更（メモリ書き込み，変数値設定）

に分類できる．これらの機能を使ってバグの原因の探索を行う．

この作業の手順の詳細は以下のようになる．

1. バグの影響が現れている時点はどこか推測する．
2. バグの影響が現れている時点（の直前）まで実行を進める．
3. そこを観察する．
4. その原因と思われる時点を推定する．
5. 実行を進めるかあるいは最初から実行して，その原因とおもわれる時点の直前で停止する．
6. そこでの実行状況を観察する．
7. 4での推定が誤りであったとわかったら 2か4へ，そこもバグの影響が現れているだけで原因ではないのならば4へ．

このように，バグの影響を受けている時点から因果関係をたどって原因を探索する．この手順は，

Ａ 推定する．
Ｂ 観察したい時点の直前で停止させる．
Ｃ 観察する．

という３種類の作業に分類できる．

まず，Ａであるが，これは残りの２種類の作業に依存している．つまり，推測するのに必要な情報を得るためには，観察が効率よく行えなければならないし，観察するためには観察したい時点の直前で停止させることが効率よく行えなければならない．従来のデバッガの問題点は，Ｂ，Ｃに関連して，

・プログラム全体の実行状況がうまく観察できない．
・意図する観察したい時点にうまくアクセスできない．

というところにある．

次にＢについて考える．従来のデバッガでプログラムの実行を停止するために使用できる機能はブレークポイントである．ところが，ブレークポイントが設定できる

対象は，ソースプログラムのある行や手続き，オブジェクトプログラムのある命令など，プログラムの静的な構造である．バグがあるのはたしかにソース（オブジェクト）プログラムのある部分だが，そこを実行したら必ずバグが発生するわけではない．またバグのある手続きを実行したからといって，バグのある行が実行されるとは限らない．つまり，ソースプログラムのある行，ある手続きは，プログラムの終了までに何回実行されるかわからないし，実行されたからといって，そこでバグが必ず発生したり，影響を受けていたりするわけではない．バグの原因があると推定した手続きがプログラムの実行中に何回も呼び出されるものであった場合，観察したいのはその中のある一回の呼び出しである．ところが，その観察したい手続きにブレークポイントを設定したのでは，いつまでたっても観察したい時点にたどりつけない．そのような場合には，呼び出し回数の少ない別の手続き（これを捜すのも困難）にブレークポイントを設定してある程度実行を進めてから，観察したい手続きにブレークポイントを設定し直すという間接的な方法を使わざるを得ない．そのため，ブレークポイントで停止したときには観察したい部分を通り過ぎていたり，ブレークポイントの実行が起きず，プログラムが終了してしまうといったこともよくある．

このように観察したい時点まで実行を進める手順はブレークポイントの設定や解除，実行再開を組み合わせた複雑なものになる．どこでどのようにブレークポイントの設定，解除を行うかは，ブレークした時点でのターゲットプログラムの状態を観察して決定する．このとき，ある程度の試行錯誤は避けられない．プログラムが比較的小さく，設計，コーディングをしたプログラマがデバッグも行う場合には試行錯誤はそれほど多くないと考えられる（バグのあるプログラムはプログラマの意図とは違うように動作しているのだから作成者自身も試行錯誤はする）．一般には，デバッグする者がプログラムの詳細な構造を完全に把握していることは期待できない．特にプログラムの移植，保守に際してはそうである．そのような場合は，観察したい時点を行き過ぎてしまい，最初からやり直すということが何回もおき，作業効率が非常に低下する．

このように，プログラムに対して設定する従来のデバッガのブレークポイントでは，デバッグ作業をしていて観察したいと思う時点を，一意に指定することができない．そのため観察したい時点にたどりつくための手間は大きくなり，しばしば観察作業そのものに要する手間より大きくなる．これは作業者への負担が大きい．

最後にＣであるが，従来のデバッガの最大の問題点は，バグを探索するためにプログラムの動きを解析したいのにも関わらず，プログラムに状態しか表示できないことである．よって，何カ所かの時点でプログラムの状態を観察し，そこからプログラムの動きを知ることになる．

ある局所的な実行状況はターゲットプログラムの状態表示と，ワンステップ実行を繰り返し行うことで観察できる．ソースコードデバッガでは，この作業はソースコードレベルで行うことができる．それだけでなく，ＶＩＰＳ［２］のように，ターゲットプログラムのデータ構造やスタックの状況を図示することによって，状態を視覚的に表現する機能を持つデバッガもある．このように，小さい範囲を詳しく観察することに関しては従来のデバッガで十分である．

しかし，バグの原因を推定するためには，プログラムの実行全体のおおまかな流れ，大局的な処理状況や，現在観察している時点がプログラムの実行のどのあたりなのかという情報も必要である．ある時点で停止しているターゲットプログラムの状態を調べて分かるのは，局所的な情報である．

従来のデバッガでは，プログラムがどこで停止しているのかは，ソースプログラムに対応づけて表示される．これでは，プログラム実行全体の中でどのあたりで停止しているのか分からない．つまり，プログラムの実行状況の一部分は詳しく観察できるものの，処理状況全体の中でのどの時点を観察しているのかが分からない．

３．　新しいデバッグ方式の提案

２節で述べたように，従来のデバッガの問題点は，プログラムの動的なふるまいの解析を，静的な構造に基づいて行うことにある．なんらかの方法で，プログラムの動的なイメージをうまく扱えるようにする必要がある．

３．１　コントロールフローへの注目

プログラムの動的な構造を表わすものとしてコントロールフローを考えてみる．しかし，プログラム実行の行単位，命令単位のコントロールフローはは膨大な量になり，また余りにも細かすぎて理解が困難であると考えられる．そこで，局所的なコントロールフローを省き，手続き，関数呼び出しの系列に注目してみる．時間順序のほかに，手続き呼び出しの深さを考えると，プログラムの実行全体を２次元的に表示することが可能になる．それぞれ手続き呼び出しを頂点，呼び出し関係を辺とすれば，プログラム（図３．１ａ）の実行全体はメインプログラムを根とする木として表示できる（図３．１ｂ）．これはクロスリファレンスの結果得られた手続きの呼び出し関係図（図３．１ｃ）に少し似ている．しかし，
①クロスリファレンスではプログラムテキストに書いて

```
main(argc,argv)
{

    while(--argc>0) {
        puts(*++argv);
        fputc(' ');
    }
}
```

図３．１（ａ）　プログラム

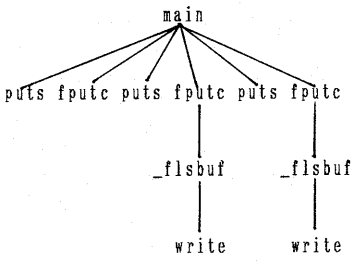

図３．１（ｂ）コントロールフロー

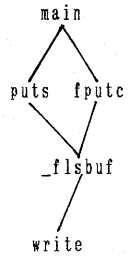

図３．１（ｃ）手続き呼び出し関係

係あるだけで呼び出し関係があるものとされるが，この木では実際に呼び出されたときに限り，呼び出し関係の枝が張られる．
②ある手続きに対応する頂点は，呼び出しが実際に起きた回数だけ存在する．
③入力等の条件を変えると，そのたびに違った木が得られる．
などの大きな違いがある．
このようにすると以下のような利点がある．
1.プログラムの実行状況全体を図示することができる．
2.この木の頂点，あるいは辺を使えば，ある条件のもとで実行したプログラムの実行のある時点を一意に指定することができる．

3.2 コントロールフロー指向デバッガ

ここで提案するコントロールフローに基づいたデバッガをコントロールフロー指向（Control Flow Oriented）デバッガ，略してＣＦＯデバッガと呼ぶことにする．ＣＦＯデバッガでは，コントロールフローを使用して動的な処理状況を作業者に提示し，また従来のソースプログラム上の位置に代えて，コントロールフロー図の上で観察したい時点を指定することができる．
ＣＦＯデバッガの機能の概略は，以下のようなものとなる（従来のデバッガの機能は含むものとする）．
①手続き呼び出しに注目したコントロールフローの記録を持っており，それを表示する．
②それを手続き名などをキーとして検索できる．
③表示したコントロールフローの上のある点を指定して，対応する手続き呼び出しあるいはリターンまで実行を進めることができる．すでに実行を終えた時点を指定した場合には，自動的に最初から実行しなおす．
④ワンステップ等で実行を進めたとき，そこが実行全体のどこにあたるのかをコントロールフローの図上で示す．

ＣＦＯデバッガには以下のような利点がある．
①プログラムの動作全体の概略がわかる．
手続きの動的な呼び出し関係は，プログラムの動作の概略そのものである．バグの影響で，コントロールフローがプログラマが意図しているものと異なっている場合は，これを調べることにより，バグの存在する手続きの候補が得られる．
②試行錯誤の低減．
観察したい場所を一意に指定することができるので，２節で述べたような，観察したい時点にたどりつくために手間をとられるということがなくなる．
ワンステップで少しずつ実行を進めながら観察していて手続き呼び出し文があった場合，その文をスキップするか，その中までワンステップで追って行くかの判断をしなくてはならない．従来のデバッガでは呼び出される手続きの名前と，せいぜい引き数しか分からない．これだけの情報ではその手続きの中でどれだけの処理が行なわれるのか見当がつかない．ワンステップによる観察は，手間がかかるので，余分な観察は省きたい．逆に，飛ばしてしまった手続きのなかに観察すべき時点があった場合には，相当な余分な手間がかかる．まず，問題の場所を飛ばしてしまったことがわかるのは，もうしばらく無駄な操作を続けた後であるし，ターゲットプログラムを終了させ，飛ばしてしまった手続き呼び出しの時点まで実行を進めるのにも手間がかかる．ＣＦＯデバッガでは，ある手続きがその中でさらにどの手続きを呼び出しているか，どの程度の量の処理が行なわれるかを推し量ることができる．
③呼び出される手続きが何か分かる．
手続きへのポインタなどを用いているプログラムでは，手続き呼び出しを行なっている部分のソースプログラムを見ただけでは，どの手続きが呼び出されるのか分からない．クロスリファレンサなどの静的プログラム解析ツールでも，このような間接的な呼び出し関係は分からない．この方式では，どれが呼び出されるのか，あるいは逆にどれから呼び出されたのかが分かる．
④二つの実行状況を比べることができる．
バグの探索の過程で，入力を変えてみてどう動きが変わるかを調べるのはよく行われることである．しかし，出力がどう違うかは容易にわかるが，内部の動作がどの様に異なるのかを知るのは従来のデバッガでは困難である．ＣＦＯデバッガでは条件を変えて実行し，処理状況がどう変わったかを比べることができる．
また，プログラムの移植を行なう場合，移植したプログラムのデバッグを行なうのは，移植者であって，元のプログラムの作成者とは異なるのが普通である．移植者のプログラムに対する理解は作成者ほど深くないため，デバッグ作業は困難である．しかし，もとの計算機上ではそのプログラムが正しく動作しているという利点もある．同じ言語であれば，手続き呼び出しのレベルで見る限り，移植したプログラムは元のプログラムと同じにふるまうはずである．オリジナルのプログラムのコントロールフローを記録し，移植したプログラムのコントロールフローと比べることにより，どこにバグがあるか検出することができる．

4. コントロールフローの表示法

コントロールフローを図示するうえでの問題点について考察する．

4.1 コントロールフローの乱れの取り扱い

実際のプログラムでは，手続きの呼び出し，リターン以外に，割り込みや大域脱出など，コントロールフローの乱れがある．これらの扱いは以下のようにすればよい．
①プログラムの終了
プログラムの終了と，メインルーチンの終了とは異なる．途中で異常終了するかもしれないし，メインルーチンまで戻らずに終了する（exit）かもしれない．ただ，終了が起きた場所はメインルーチンかさもなければ木の最も右のノードであるから，終了原因さえ作業者に知らせればよい．
②コントロールフローの乱れ
Ｃのsetjump()/longjump()[3]，ＰＡＳＣＡＬのgotoなど，手続きから戻るのではなく，飛び出す（大域脱出）ことがある．これもどこからどこへ起きたか，なんらかの方法で表示すればよい（図４.１）．
③割り込み
例えばＵＮＩＸでのsigvec[4]システムコールを使って，ユーザプログラムで割り込みが扱われている場合，割り込みで呼び出された手続き（及び，さらにそれから呼び出された手続き群）をどう扱うかという問題が生じる．これは，割り込まれた手続きが割り込み処理の手続きを呼び出したかのように木に接続すればよい．このとき，手続き呼び出しを表わす枝と割り込みによる呼び出しを表わす枝を区別できるように表示する（図４.１）．

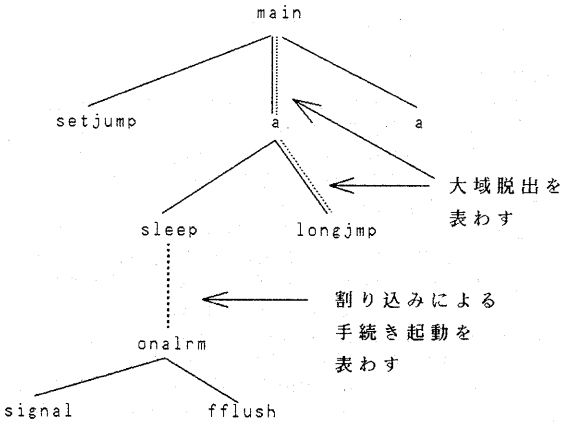

図４．１　例外的コントロールフローの表現

このようにすれば，プログラムの実行の履歴から一つの木を構成することができる．

４．２　必要な情報の抽出

コントロールフローを表わした木は一般には大きいと考えられるので，その全体を画面に納まるように表示することはできない．一部を表示するしかないわけだが，”一部”をどのように切り出すかが重要である．これにはいくつかの手法が考えられる．

（Ａ）入りきらない部分を切り捨てる方法

1)　全体が表示できるような大きな仮想画面を考え，ディスプレイはそれの一部を表示する窓であるとする．実際のプログラムでは，木全体は一度に表示できる画面の大きさに比べて非常に大きくなると考えられる．この方法では，たまたま近くにある頂点や枝が同じ画面に表示される．木全体が大きくなるほど，空白部分と枝が占める比率が多くなってしまう．

2)　ある頂点を考え，それを根とする部分木の根に近い部分を画面に納まるだけ表示する．

3)　ある頂点を考え，それが画面の中央にくるように，それを根とする部分木，祖先，兄弟を画面に納まるだけ表示する．1)と異なるのは，画面に表示された部分は連結であるという点である．

（Ｂ）元の木から作業者が注目する情報だけを含んだ木に縮約する方法

4)　注目する手続き（複数）を選び，それ以外の手続きに対応する頂点を縮約する．

5)　各ノードに対応する手続きを，手続き名を表示するのではなく頂点の形で区別する．表示できる（そして容易に区別できる）頂点の形はたかだか数種類と思われるので，いくつかの手続きを選び，それ以外の手続きに対応する頂点は同じ形で表示する（図４．２）．

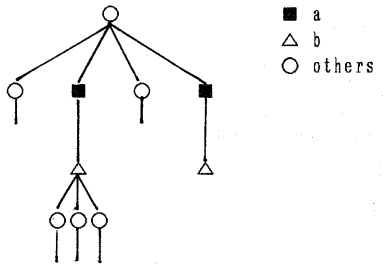

図４．２　手続きを頂点の形で表わした例

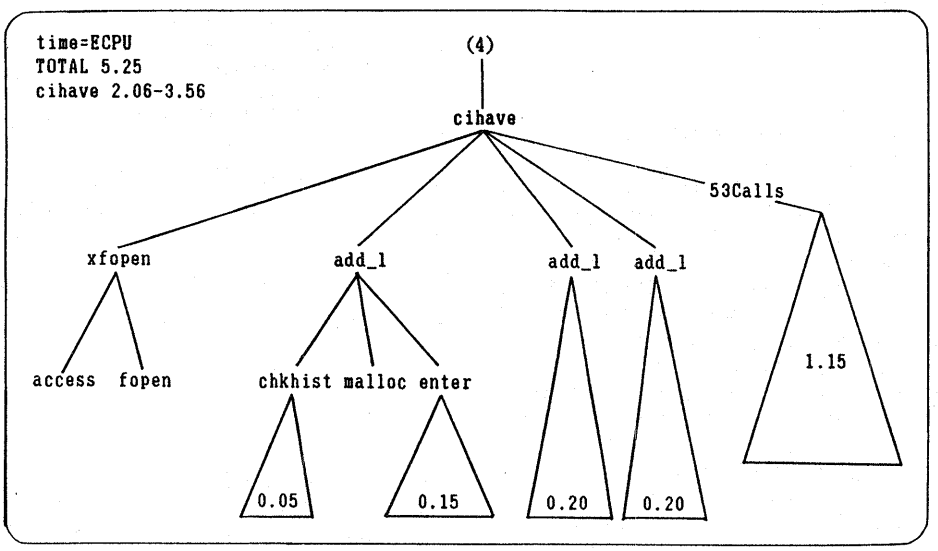

図４．３　省略部分の表示法

（Ｃ）一つのノードの子供が多すぎる場合の処置
6) 単に一部分だけを表示する（図４．３）．
7) 同じ手続きが何回も呼び出されているはずなので，何回も呼び出されている手続きは省略して１つだけ表示する．

（Ｄ）省略部分の表示方法
このような省略を行なった場合には，どこでどのような省略が行なわれたかを表示することも重要である．
8) 端にノードのない枝で部分木の省略が行われたことを示す（図４．２）．
9) ある部分木を表示したときに，メインルーチンからその部分木の根までの手続き呼び出しの深さを表示する（図４．３の中央上の“(4)”）．
10) 省略された部分木にあたる部分が消費したＣＰＵ時間を表示する（図４．３の三角形の中の数値）．

実際には，まず，４～７の手法で必要な情報だけを含んだ小さな木を作り，その一部を１～３の手法で切り出し，８～１０の手法で省略部分を示して作業者に提示すればよいと考えられる．

５． 従来型のデバッガとの比較

３，４節で述べた機能を実現したデバッガと従来型のデバッガを思考実験により比較した．

思考実験は，以下の条件で行なった．
①比較対象は ｄｂｘ[5]
②ターゲットプログラムは，これはBnews[6]という ネットワークニュースシステムのなかの，inews というプログラム．
③バグは，過去に実際に見つけたもの．
④入力したコマンド数で評価する．
バグ発現の状況：
ihaveコントロールメッセージの処理で，入力ファイルのデータ部分が空であったのと同じ動作をしてしまう．ihaveコントロールメッセージを受け付けたとログファイルに記録してinewsは正常終了する．

◎従来型のデバッガ（ｄｂｘ）
ihave コントロールメッセージの扱いがおかしいので，ihaveをキーとしてソースプログラムを調べると，c_ihave()という関数（図５．１）があることがわかる．そこで，c_ihave()は調べるべき候補にはなるが，これに特定はできない．これ以外の見当違いのところを調べてしまう可能性もある．実際，筆者がデバッグしたときは，inews 自身がまったく正常に終了し，また入力を受け付けたというログも残るので，c_ihave()を調べるのが後回しとなり，相当な無駄な労力を費やした．しかし，この点は定量的に評価するのは難しい．

さて，c_ihave()を調べようと思うと，ここにブレークポイントを設定して実行する．幸い，この関数は１回しか実行されないので，ｄｂｘでも２コマンド（stop in c_ihave ,run）で到達できる．そしてステップコマンド８回で３０行目のwhile文に到達する．そして，ステップコマンドを入力すると，while文の中をまったく実行せずに，３７行にいってしまう．この時点で，ftell(infp)かoutposの内容がおかしいことが分かる．outposとinposの値を調べてみると（２コマンド），outposの内容は０なので，ftell()かxfopen()がおかしいことがわかる．計１２コマンドである．

◎ＣＦＯデバッガ
図５．２(a)が，ihaveコントロールメッセージを受信したときの正常な動作記録の一部である．図５．２(b)は実際にあったバグを再現したものである．もし，正常に動作しているものと比較したなら，関数c_ihave()に問題がありそうな事がすぐに分かる．

まず，全体のコントロールフローを調べる．そうすると，c_ihave()の中から，xfopen()しか呼ばれていないことがわかる（Ｃライブラリ関数はトレース対象外）．xfopenは２７行のif文のelse節にしかないので，while文の中を一度も実行していないことが分かる．そこで，c_ihave()まで実行を進め，３０行で止めて（２コマンド），変数inpos,outposの内容を調べる（２コマンド）．計６コマンドで，ｄｂｘと同じくxfopenかftellがおかしいことが分かる．

このように少ないコマンド数で同じ情報が得られる．

```
 1 c_ihave(argc, argv)
 2 register char **argv;
 3 {

 9     if (argc < 2)
10         error("ihave: Too few arguments.");
11     if (STRNCMP(PATHSYSNAME, argv[argc - 1], SNLN) == 0)
12         return 0;
13     list[0] = '¥0';
14     if (argc > 2) {

22     } else {
23         register FILE *    outfp;
24         register long    outpos, inpos;
25         char        myid[256];
26
27         outfp = xfopen(INFILE, "a");
28         outpos = ftell(outfp);
29         inpos = ftell(infp);
30         while (ftell(infp) < outpos) {
31             if (fgets(myid, sizeof myid, infp) != myid)
32                 error("iline: Can't reread article");
33             myid[strlen(myid) - 1] = '¥0';
34             if (findhist(myid) == NULL)
35                 (void) fprintf(outfp, "%s¥n", myid);
36         }

42         (void) fclose(outfp);
```

図５．１ 関数 c_ihave() （部分）

６． 試作と実現上の問題点

試作したＣＦＯデバッガのプロトタイプについて述べ，次に実際にＣＦＯを実現する際に考えられる問題点について考察する．

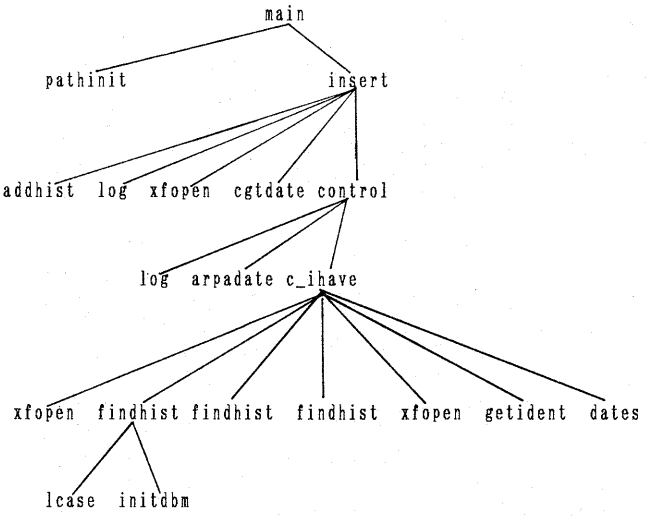

図５．２（ａ）　正常なコントロールフロー

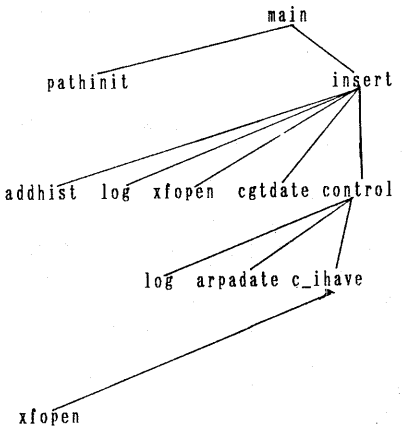

図５．２（ｂ）　バグの影響を受けている
コントロールフロー

６．１　プロトタイプの作成

ＣＦＯデバッガの実現可能性を示すため，基本的な機能を持ったプロトタイプを作成した．

６．１．１　プロトタイプの構成

対象言語はＣである．このプロトタイプは，ｄｂｘの機能に加えて，関数の呼び出し状況のデータを取り，表示する機能がある．ＳＵＮ２上で作成したが，バークレー系ＵＮＩＸならばどこでも容易に移植できよう．

作成，評価を短期間に行いたかったので，図６．１に示す内部構成図のようにｄｂｘを内蔵することにした．今回作成した部分は，ｄｂｘのフロントエンドとして動作する．疑似ターミナルを介してｄｂｘと接続し，ｄｂｘのコマンドはそのままｄｂｘに伝え，そのコマンドに対するｄｂｘからの出力はそのまま作業者側に渡す．今回追加したコマンドが入力されたら，トレースを取るためのｄｂｘのコマンド群をｄｂｘに与え，ｄｂｘの出力を解析して木をつくる．この間，作業者からの入力はターゲットプログラムに与え，ターゲットプログラムの出力は作業者のターミナルに出力する．

６．１．２　コントロールフローデータ採取機能の実現

関数の呼び出しと関数の終了が分かればよいので，ｄｂｘのトレース機能を使う．ｄｂｘでは，一度ブレークして実行を再開するあるいはトレースポイントを通るだけで，すべてのブレークポイントの解除／再設定が起きる．つまり，ブレークポイントは実行されなくても，設定されているだけで実行速度を引き下げる．このため，単にすべての手続き（関数）にトレースポイントを設定したのでは，時間がかかりすぎる．そのため，まずすべ

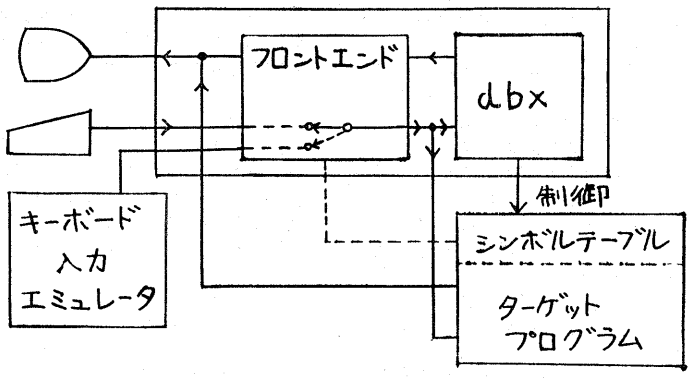

図６．１　試作したプロトタイプの内部構成

ての手続きにブレークポイントを設定して実行する．ブレークすると，そのブレークポイントを解除して実行を再開する．こうして実際に呼び出される関数を調べる．

次にそれらの関数にだけトレースポイントを設定してデータをとる．また，出現回数に制限を設けて，規定の回数以上，実行された手続きはトレースを停止することにした．

6．1．3　その他の機能

実行の度に毎回同じ動作をさせるためには，なんらかの初期化作業（作業ファイルのクリーンアップなど）が必要である．そこで，ターゲットプログラムの実行開始に先立って行なうべき初期化作業を定義できるようにした．

後で評価するために，このデバッガの入出力の記録を取れるようにした．

6．2　実現上の問題点

実用に耐えるＣＦＯデバッガを作成するには，解決しなければならない問題がある．それはＣＦＯデバッガのそれぞれの機能を現実的な効率で実現する事である．

6．2．1　コントロールフローの履歴情報の採取と表示

まず，手続きの呼び出しに関するコントロールフローの情報採取と，その情報から木を構成することに関しては，実際にプロトタイプを作成することで実現可能であることを示した．

次に，得たトレースデータの表示手法に関しては，プロトタイプでは何もなされていない．単に実行開始から順に全体をプリントアウトするだけである．これに関しては，４節で論じたような手法を実現すれば良いと考えられる．

４節では，プログラムの実行全体のトレースがあるものとして話を進めた．試作したものも，実際にプログラムに実行全体の関数トレースを取るものである．しかし，実際にはこれは非現実的である．トレースしながらの実行は（ハードウェアの助けをかりない限り）トレースなしの実行よりもかなり遅くなる．トレースデータを取るための時間が作業者が我慢できないほど長くては意味がない．また，実行時間の長いプログラムだと，トレースの量も取り込めないほど膨大なものとなり得る．

全体のトレースは時間がかかる．それならば，実際に全トレースを取らなくても全体のトレースがあるかのように，振舞えばそれでいい．作業者に表示を要求されるつど，その部分のトレースをとればよい．

まず，表示を要求されている部分のうちの根にいちばん近い頂点に対応する時点（まだまったくトレースをとってないときには，メインルーチンのはじめ）まで実行を進める．その部分の実行がすでに終わっていたら，最初から再実行する．次に，表示を要求されている手続きに，ブレークポイントを設定する．あとは，ブレークする度にアドレスとスタックを調べ，実行再開してゆけばよい．１画面分のデータがたまれば実行を停止してよい．一回のトレースでは一画面分のトレースデータを取ればそれでよいので，必要以上，深い手続き呼び出しが起きたなら，一画面にはいる深さに制御が戻ってくるまで，ブレークポイントを解除して実行する．こうすれば，停止／実行再開の回数を減らすことができる（処理の都合からいえば，ある手続きの呼び出しからリターンまで通して　－　つまり，部分木を　－　をトレースすることが望ましい）．

このようにして取ったトレースデータがすでに行なわれたトレースと同じ手続きを対象として行なわれたのならば，新たなトレースデータを単に併合してゆけばよい．異なる場合には，既にあるトレースデータのどこに接合するかを記録した上で別に管理する．よって，トレースデータは作業者には木として見せるが，林として管理する必要がある．

6．2．2　木の頂点に対するブレークポイント

内部的にはブレークポイントの設定変更と実行再開を何回か繰り返すことになる．まず，既に実行を終えた点であれば最初から実行しなおす必要がある．

1）　現在実行中の部分から，目的とする時点までのトレースが同じ条件で取られているとする．つまり，先に述べた林の，同じ木に”現在地点”と目的とする頂点がある場合である．手続きの実行トレースを取った時点で，木のノードに対応する手続きが何回目に呼ばれたものかは分かっている．よって，目的とする頂点に対応する手続きにブレークポイントを設定し，実行再開／ブレークポイントによる停止を何回かくりかえせば到達できる．

2）　トレース条件が異なる場合は，まず，目的とする頂点の祖先である頂点まで実行を進める．そして，木と木の接合点を順次，1)の方法でたどって行けばよい．

6．2．3　ターゲットプログラムの動作の再現性

何回も実行させてその結果をつなぎ合わせるためには，ターゲットプログラムが毎回同じ動作をしなければならない．再現性のない動作をするターゲットプログラムにはこのデバッグ方式は適用できない．このようなプログラムは従来の方式でもデバッグ困難であった．ただし，手続き呼び出しの発生の系列さえ同じなら，影響を受けない．また呼び出す手続きが多少異なっても，その異なる部分に注目しない限り，矛盾が起きることはない．

7．　まとめ

デバッグはプログラムのふるまいという動的なものを調べる作業である．従来のデバッガの操作はプログラムコードという静的なものを作業者に見せそれに基づいて行われていた．そこで，動的なプログラムの動作を作業者に提示し，またこれに対して作業することを許すコントロールフローに基づいたデバッグ手法を提案し，デバッグ効率が向上することを示した．

さらに，実現に際しての問題点を考察し，また基本的な機能を持ったプロトタイプを試作することでこの方式の実現可能性を示した．

参考文献

[1] S.C.Johnson: "Lint, a C Program Checker", 4.2 BSD UNIX PROGRAMMER'S MANUAL, (1983).
[2]S.Isoda et al, "VIPS: A Visual Debuger", IEEE SOFTWARE, Vol.4 No.3, pp8-19, (May 1987).
[3] D.M.Ricche et al: "The UNIX Programmer's Manual", Bell Laboratpries, (1978).
[4] W.Joy et al: "4.2BSD System Manual", 4.2BSD UNIX PROGRAMMER'S MANUAL, (1982).
[5] Bill Tuthill: "Debugging with dbx", 4.2BSD UNIX PROGRAMMER'S MANUAL, (1983).
[6] M.Horton et al:"INEWS(1)", 4.2BSD USER CON--TRIBUTED SOFTWARE SUPPLEMENTAL MANUAL, (1983).