ソフトウェア工学 49－3
（１９８６．７．３０）

ＦＯＲＴＲＡＮプログラムの論理的なエラーに関するデバッグ支援ツール

畠下　章　　永田　守男

（慶応義塾大学　理工学部　管理工学科）

現在、コンパイル時や実行時のエラーの発見と修正に対しては優れているデバッガも多い。しかし、プログラマの意図しない出力が得られるときには、メモリダンプやトレースなどを利用しているにすぎない。このようなデバッグは人間の負荷が大きいと考えられるので、本研究ではそのようなエラーを対象としたデバッグ支援ツールを設計し製作する。すなわち、プログラマが混乱しがちなプログラム中の様々な論理的な条件などを整理することを通じてデバッグの支援をするツールを考える。

本システムは、ＦＯＲＴＲＡＮ言語で記述され、ＧＯＴＯのないメインルーチンのみのプログラムを対象としている。そして、プログラマの意図しない出力が得られるエラーに対して、デバッグのための３つのモードを用意している。それらは、プログラムの場所を指定して条件を導くモード、逆に条件を与えてそれによってプログラムの実行される場所を示すモード、それに動的な情報を表示するモードの３つである。

本システムの特徴は、ＦＯＲＴＲＡＮプログラムにおけるＤＯ文とＩＦ文の混在、入れ子などによって生じるような論理的なエラーに対して、利用者に整理した情報を提供している点にある。この特徴を生かせば、これまでに研究や開発の進んでいるデバッガなどと組合せることによって、強力なサポートツールになると考えられる。

An Interactive Debugging Tool for Correcting Logical Errors in Fortran Programs

Akira Hatashita and Morio Nagata

Dept. of Administration Engineering, Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Yokohama 223, JAPAN

A programmer is often confused by inspecting complex logical conditions to be held in places of his or her programs during correcting logical errors in the debugging process. Thus, we propose an interactive tool displaying some logical conditions and corresponding places of the program written in the Fortran language without GOTO statements. Our tool provides following three facilities. First, if the user of our tool specifies a certain place of a program, then it shows the logical condition to be held there; Second, when a certain condition is given, our tool indicates the corresponding place of the program; Third, it shows the dynamic information of the user's program. It is expected that a powerful support tool is built by combining facilties of our tool and existing debuggers.

1．はじめに

ソフトウェアの開発には、依然として人手と手間がかかっているのが現状である。そのようになっていることの主たる原因の1つにプログラムのデバッグ作業におけるプログラマの負担が挙げられる。現在では、ＦＯＲＴＲＡＮのような高水準言語を例にとって考えると、コンパイル時に見つかるエラーや実行時のエラーの発見や修正に対しては優れたデバッグ機能を備えた処理系も多い。しかし、それらを取り除いた後で生じる論理的なエラーについてのデバッグは依然として負担が大きいと考えられる。

また、プログラムのテスト技術や正当性を検証する技法の研究も多く報告されてきた［1，2］。しかし、それらはバグのないことを示すのに役立つだけである。ある特定の型の言語のプログラムについては、積極的にデバッグの情報を与えるものもある。しかし広く使われているプログラミング言語でのプログラムのデバッグに対して有効な情報は、これまでの研究成果を使っても十分には、得られない。

このような背景のもとでは、デバッグにおける労力がより軽減されるシステムの提供が望まれる。そこで、本研究では、プログラマの意図しない出力が得られるような論理的なエラーを対象とし、そのデバッグ作業に対して有効なサポートをするシステムを設計し製作する。

対象とする言語は普及率の高いＦＯＲＴＲＡＮを取り上げる。ここでシステムとして製作する機能は、バグのありそうな場所の発見に役立つ情報を整理して、利用者に提示しようとするものである。すなわち、プログラム中のある場所とその場所で成立している論理的な条件との関係が本システムで与えられる情報の中心になる。これは、従来のデバッガには直接備わっていないと考えられ、しかもデバッグに有効な機能であると考えられる。この機能は、ＦＯＲＴＲＡＮプログラムの構造によってエラーがどのように生じるのかを考慮した結果、考案されたものである。

本研究の目的は、このような機能の実現を通じて今後のデバッガへの一つの基礎づけを与えようとするところにある。ここで示すツールと従来のデバッガ等を有機的に組合せることで、デバッグにおけるプログラマの負担を減らすことが期待できる。

2．デバッグの手法とデバッガ

2．1　デバッグの一般的手法

デバッグの方法は論理的なエラーを対象とした場合に、その手段とアプローチの仕方とでそれぞれ分けて考えることができる。

デバッグの一般的な手段は、次の4つに大きく分けることができる。

（1）ソースリストと実行結果を用いる方法
（2）プログラム内に印字命令を挿入する方法
（3）簡単なテストケースを実行させる方法
（4）自動デバッグツール（デバッガ）
　　を利用する方法

これらは一応区別はしているが、実際は組合せて用いられる場合が多い。

一方、デバッグのアプローチの仕方として代表的なものは次の2つである。

1）プログラムをトップダウン的、すなわち、その流れに沿って論理的に正しいかどうかを調べていく。
2）プログラムを正しくない結果からボトムアップ的、すなわち、流れとは逆方向に論理が誤っている点まで調べる。

この両者は対照的であるが、実際のデバッグの過程では無意識のうちに併用している場合も多い。

プログラマは自らの経験やその時の状況に応じて、先に述べた手段を選び、このアプローチによってデバッグをしていくのである。従って、どの方法が良いかは一概には言えない。但し、一般的に言えることは、手段の（1）のみではプログラマの負担は大きいであろう。

（2）を行なえば新なエラーを生じる可能性がある。（3）ではテストケースを考えること自体が負担である。（4）の場合はその利用法さえ習得していれば、余計な心配はしなくてもよい。しかし、その場合のデバッグとしての機能が、どの程度役立つものであるかが重要である。次に従来のデバッガの機能の特徴について述べる。

2．2　従来のデバッガ

現在、プログラムの解析・評価ツールは様々なものが発表されている［1］。それらは主に静的、動的な2つのツール［5］に分かれるが、プログラムのテストを目的とするものが多い。デバッグを目的とする従来のデバッガの主な機能には、次に示すようなものがある［7］。

・プログラムの実行順序の確認（トレース）
・プログラムの実行の中断（ブレーク）
・プログラムの1命令ずつの実行（ステップ）

・変数の内容の確認
・環境の変化による実行
（変数の内容を変えて実行するなど）

これらは、メインプログラムのみでプログラムが作成されている時のものである。サブルーチンや各処理系について考えれば、さらに多くの機能が挙げられるが、本研究ではメインプログラムだけのレベルで話を進める。

ここで示したデバッグ機能の特徴について考え、これらの欠点を指摘して本システム作製の動機について述べる。デバッグを行なう利用者は、プログラムの実行過程の任意の位置で自分が必要とする機能を働かせることができる。つまり利用者の自由度は大きいわけである。しかし、これらの機能を用いたデバッグは変数の内容を確認するなどの動的なものが中心となっていて静的なデバッグのサポートには向いていない。しかも基本的には、１ステップずつモニターしていく機能であるため、プログラム内のまとまった命令群の状況や分岐条件などが他の命令とどのような関係になっているのかを確認することは容易ではない。そこで、本システムではこの点を考慮してデバッグの機能を考えることにした。

３．プログラムのブロック化

３．１　ＦＯＲＴＲＡＮプログラムの論理的なエラー

ＦＯＲＴＲＡＮが７７バージョンに改訂されて、ブロックＩＦ文が導入されてからは、プログラムを構造的に記述することが可能となった。そのことにも関係して、ここではＧＯＴＯ文を除いて考える。メインプログラムだけのレベルで考えると、プログラムの制御に関係している主な実行文は、ＩＦ文やＤＯ文などである。これらの実行文を使って作られるプログラムの構造は、およそ次の３つに分けることができる。

（１）ブロックＩＦ文などによるブロック構造
（２）ＤＯ文などによるループ構造と
　　その入れ子構造
（３）（１），（２）が混在した場合
　　の複雑な構造

（３）のような複雑な構造に対しては、プログラムを記述する側にとっての負担が大きいであろう。こうした複雑な構造内では、考えている実行経路で成り立つ条件を常に把握していなければならないからである。従って、そのような場合には論理的なエラーが生じ易いと考えられる。しかも、そうしたエラーをデバッグするには、再びプログラムの構造を確認しなければならない。

そこで、こうした複雑な構造によって生じる論理的なエラーを対象としたデバッグ機能を考えるために、次に示すプログラムのブロック化を導入する。

３．２　ブロック化の導入

プログラムをブロックに分けることによって、論理的なエラーに対するデバッグ機能を設けることができる。

ここで扱うブロックとは、サブルーチンを含まないメインプログラムのレベルにおいて、ＤＯ文やＩＦ文のような制御の分岐に関する実行文を除いた逐次実行文の集合を指す。図１にブロックの例を示す。

ブロック化の概念を取り入れたのは主に次に示す理由からである。

・ブロック内における実行の流れは単純なものであり、ブロックの内部のみで考えると制御条件に悩まされる心配がない。
・ブロック内で実行される処理は、そのブロックを取り巻く条件のもとで意味を持っている。従って処理と条件を区別することにより、プログラムの構造に対して整理された情報を利用者に提供することができる。

```
      INTEGER I,J,WORK,UP(5)
      DO 10 I=1,5
        READ(5,*) UP(I)        1命令のブロック
10    CONTINUE
      DO 20 I=1,4
        DO 30 J=I+1,5
          IF(UP(I) .GT. UP(J)) THEN
            WORK  = UP(I)      3命令のブロック
            UP(I) = UP(J)
            UP(J) = WORK
          ENDIF
30      CONTINUE
20    CONTINUE
      DO 40 I=1,5
        WRITE(6,*) UP(I)       1命令のブロック
40    CONTINUE
      STOP                     2命令のブロック
      END
```

図１　ブロックの例（昇順に並べ換えるプログラム）

これらの理由から、ブロックをもとにしたデバッグの機能をシステムとして実現した。

すなわち、あるブロックを実行するときの条件を示す機能、ある論理式からこれに対応するブロックを示す機能、プログラムを実行した時のあるブロックに関しての動的な情報を示す機能を組み込んだシステムを作製した。

なお、ここで指すブロックは本来の基本ブロック［４］の考え方とは必ずしも一致していない。制御条件なども基本ブロックとなるが、ここでのブロックには条件などは全く含まれないのである。

４．システムの概要

４．１　デバッグ機能

システムの製作にあたっては、我々の考えたデバッグ機能を具体化することが第一であるので、本質的ではない部分を取り去ってシステムの扱えるプログラムの範囲を制限した。

読み込まれるプログラムはＦＯＲＴＲＡＮ言語で記述され、コンパイル時のエラーや実行時のエラーが既に取り除かれたものとする。また扱う文法は基本演算を中心とする簡単なものとする。

システムは会話型デバッガとしての機能が要求される。また記号処理を中心にして論理式等も扱うことになる。従って本システムはＬｉｓｐを用いて記述した。ＬｉｓｐはＣｏｍｍｏｎ　Ｌｉｓｐ［８］のひとつであるＶＡＸ　ＬＩＳＰ［９］を利用した。

デバッガとしての各モードは図２に示す通りである。以下、各モードの機能について説明する。

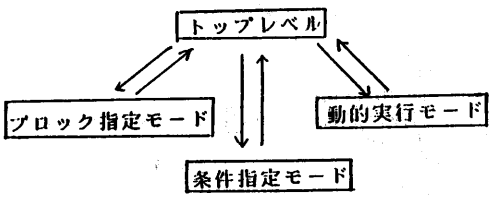

図２　システムの各モード

（１）ブロック指定モード

このモードは、利用者がブロックを指定することによって、そのブロックに制御を移すための条件をわかりやすく提供するためのものである。ブロックが実行されるための条件は、不等号などを用いて表示される静的なものである。なお、このモードの中では、利用者にとって分かりやすいように、次のような簡約化を行なって条件を表示する。

・条件の簡約化

条件が幾つも存在する場合に、可能であればそれらの条件を簡約する。図３に条件の簡約化の例を示す。この図は、ブロック１の実行は$6 \leqq I \leqq 10$のときであってここでＩが２ずつ増加していることを、ブロック２では$2 \leqq I \leqq 4$で、Ｉがやはり２ずつ増えることを示している。

```
DO 10 I=2,10,2          ＜ブロック１を指定＞
  IF(I.GT.5) THEN
                          6 ＜ I ＜ 10   I+2=>I
  ブロック１                =    =

  ELSE                  ＜ブロック２を指定＞

  ブロック２              2 ＜ I ＜ 4    I+2=>I
                            =    =
  ENDIF
10 CONTINUE
```

図３　条件の簡約化

（２）条件指定モード

このモードは、ブロック指定モードとは反対に、利用者が条件を指定することによってどのブロックに制御が移るかを表示する。コーディングした流れに沿って制御の確認を行なえるのが特徴である。

（３）動的実行モード

（１），（２）のモードが静的な解析結果を利用しているのに対して、このモードでは動的な情報を提供することができる。機能としては、先ずプログラムを実際に実行させる。そして、利用者の指定したブロックを中心に実行過程における変数の内容や、ブロックに対する条件の値などを動的に表現する。

４．２　システムの構成

システムの主な流れは図４に示す通りである。次に各構成要素について述べる。

１）プログラム解釈部
読み込まれたプログラムをまとめて解釈し、実行しやすいようにＬｉｓｐの内部形に変換する。

２）デバッグ情報作成部
各モードで使用するための各種デバッグ情報を作成する。情報としては、ブロック、条件、表示用・簡約化条件、動的実行などに関する情報である。

３）デバッガ制御部
図２で示される各モードとトップレベルを統括する。動的実行モードの選択によっては、プログラム実行部に移る。

４）プログラム実行部
動的実行モードの指定によって起動する。指定されたブロックの実行に移るとその動的表現、及び条件を表示する。

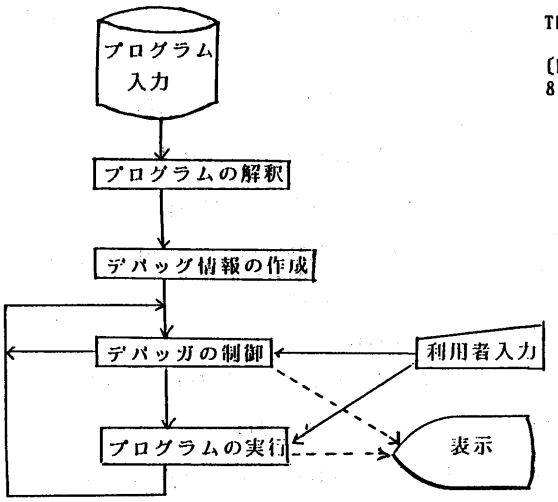

図４　システムの主な流れ

５．実行例

図５で示される幾つかのエラーを含むプログラムを考える。このプログラムは点数（８０以上，８０未満６０以上、６０未満）によってＡ，Ｂ，Ｃの３つのクラスに人数を求めて出力することを意図する簡単なものである。このプログラムを用いて各モードによるデバッグ機能の実例を示すことにする。

本システムは会話的に使えるようになっていて、メニュー方式で動く。

（１）ブロック指定モードによるデバッグ

本システムでは、ソースプログラムとそのブロック化を行なったものをコマンド入力によって表示させることができる。図５（ｂ）がブロック表示の例である。

図５（ａ）で示されるプログラムに対して、このモードを利用すると次のようになる。この中での利用者の入力は「＝＞」のあとの３という数字だけである。なお、ＢＬＩＳＴは図５（ｂ）で示されるブロックを表示し、ＬＩＳＴは図５（ａ）で示されるプログラムリストを表示するコマンドを、ＥＸＩＴはモードから抜け出すコマンドを表す。

```
(*BLOCK*) Input block number,BLIST,LIST,EXIT =)3

** CONDITION **
1 ( I ( 5  I+1 =)I
  =    =
TEN(I) ( 80
       =
(BLOCK. 3)
8              A = A+1
```

これは、Ａクラスに場合分けされる時の条件が８０点以下であることをはっきり示している。従って、Ａクラスに場合分けする条件を再確認することによってエラーを発見し、図５（ｃ）のように修正する。

（２）条件指定モードによるデバッグ

図５（ｃ）のように修正したプログラムに関して条件指定モードを適用する。

このモードでは選択条件が次のように用意される。

```
** CONDITION **
( 1) I=1,5
( 2) (TEN(I) .GE. 80)
( 3) (TEN(I) .GE. 60)
```

この条件を選択することによって思った通りのブロックに制御が移るかを確かめる。

```
(*COND*) Input condition number,CLIST,LIST,EXIT =)1,2,3
Input logical value. (TEN(I) .GE. 80) =)F
Input logical value. (TEN(I) .GE. 60) =)F

** CONDITION **
I=1,5
(TEN(I) .GE. 80) =)F
(TEN(I) .GE. 60) =)F

No block with this condition!
```

思い通りのブロックに制御が移らないことがわかる。従ってこのブロックに関する条件を再確認することによってエラーを見つけ、図５（ｄ）のように大幅な修正を行う。

（３）動的実行モードによるデバッグ

図５（ｄ）のように修正したプログラムに対してＢクラスに分けるブロックを中心に動的実行モードを適用する。

```
(*RUN*) Input block number,RUN,BLIST,LIST,EXIT =)4
Input start,end value :I =)1,5
?
100
?
30
?
90
?
60
 I = 4 (60 .GE. 80)=)F (60 .GE. 60)=)T
10      B = 0+4
?
70
 I = 5 (70 .GE. 80)=)F (70 .GE. 60)=)T
10      B = 4+5
A =   2   B =   9   C =   1

Dynamic execute ended.
```

人数は１人ずつ加算されるべきであるが、実行結果より明らかにこの加算にエラーのあることがわかる。図５（ｅ）のように修正する。

以上に示した実行例は、説明のための簡単な小さなプログラムである。しかし、このような機能は、むしろ複雑に条件の絡み合う大きなプログラムに対して有効なものになろう。但し、その場合にはサブルーチン等の扱いが問題となる。

```
1           INTEGER A,B,C,TEN(5),I
2           A=0
3           B=0
4           C=0
5           DO 10 I=1,5
6             READ(5,*) TEN(I)
7             IF(80 .GE. TEN(I)) THEN
8               A = A+1
9             ELSE
10               IF(TEN(I) .GE. 60) B = B+I
11                                  C = C+1
12            ENDIF
13    10      CONTINUE
14            WRITE(6,*) 'A = ',A,' B = ',B,' C = ',C
15            END
```

（ａ）エラーを含む初期のプログラム

```
(BLOCK. 1)
2           A=0
3           B=0
4           C=0
(BLOCK. 2)
6             READ(5,*) TEN(I)
(BLOCK. 3)
8               A = A+1
(BLOCK. 4)
10                                B=B+I
(BLOCK. 5)
11                                 C = C+1
(BLOCK. 6)
14          WRITE(6,*) 'A = ',A,' B = ',B,' C = ',C
15          END
```

（ｂ）プログラムのブロック表示

```
 7            IF(TEN(I) .GE. 80) THEN
```

（ｃ）モード１により７行目を修正

```
9           ELSE IF(TEN(I) .GE. 60) THEN
10                    B = B+I
11                ELSE
12                  C = C+1
13          ENDIF
```

（ｄ）モード２により場合分けを修正

```
10                  B = B+1
```

（ｅ）モード３により１０行目を修正

図５　サンプルプログラムとデバッグの過程

６．おわりに

本システムはプログラム解析における静的、動的の両者の立場を取り入れている。また、ブロック指定、条件指定のモードはデバッグにおけるボトムアップ、トップダウンの方法のどちらでも使えるようになっている。

プログラムのブロック化という考え方は、プログラムの構造を考慮した上で取り上げたものである。そして、そのブロックと制御条件を区別することにより本システムのデバッグ機能が製作されたわけである。結論的に言えることは、従来のデバッガが扱わないような整理された情報を本システムは利用者に提供できるということである。本研究のアイデアと共通した考え方を持つ既存のものとして、デシジョンテーブルがあるが、動的な会話型のデバッギングツールにまとめ上げるには、ここで示した手法のほうが便利である。

本システムは従来とは異なった観点で機能を考案したものであるが、まだまだ改善の余地はある。例えば、モード２において条件としてプログラム上の記述をそのまま用いるのではなく、利用者が変数の値の範囲を任意に指定することができるようにすることなどが挙げられる。

今後はさらに本システムと従来のデバッガの長所を取り込んだシステムの開発が必要であろう。そのためには評価実験なども行う必要があるが、それ以前に本システムが扱える文法的制約を減らして少しでも実用的なものにする必要がある。

これから開発されるデバッガとして一般的に要求されることは、プログラムの内容把握を促すための何らかの推論機能、プログラム全体を見渡すための全体的な情報の提供などが挙げられるであろう。

［参考文献］

［１］Ｇ．Ｊ．Myers著
長尾真　監訳，松尾正信　訳：
ソフトウェア・テストの技法，
近代科学社　(1980)

［２］Ｚ．Manna著：プログラムの理論，
日本コンピュータ協会　(1975)

［３］Nagata,M．, Akiyama,T．
and　Fujikake,Y．：
An　Interactive　Supporting
System　for　Functional
Recursive　Programming,
Information　Processing 80,
North-Holland, pp.263-268 (1980)

［４］松本吉弘：
ソフトウェアの考え方・作り方，
電気書院　(1981)

［５］宮本勲：
プログラム・テスト支援ツール，
情報処理，Vol.20，No.8，
pp.688-693 (1979)

［６］牛島和夫，原田賢一：
ソフトウェアの解析と評価ツール，
情報処理，Vol.20，No.8，
pp.703-710 (1979)

［７］VAX-11 ｼﾝﾎﾞﾘｯｸ･ﾃﾞﾊﾞｯｶﾞ ﾚﾌｧﾚﾝｽ ﾏﾆｭｱﾙ，
Digital　Equipment 社 (1982)

［８］Ｇ．Ｌ．Steele Jr.他著，
井田昌之　訳：
Common ＬＩＳＰ，共立出版(1985)

［９］ＶＡＸ　ＬＩＳＰ　User's　Guide，
Digital　Equipment 社 (1984)