ソフトウェア工学 51－2
（１９８６．１１．２６）

構造化プログラムへの変換とその応用

青山　幸也

日本アイ・ビー・エム株式会社　製品保証

本報告では、非構造化ソースプログラムを、論理的に等価な構造化ソースプログラムに変換するアルゴリズムと、それにもとづく試作ツール、さらにそれらの構造テストへの応用について紹介する。

変換アルゴリズムは二進木の考え方を用いており、非構造化プログラムを二進木構造に変換し、それに対していくつかの規則を適用することにより構造化変換を行なう。ART-SP(Automatic Restructuring Tool to Structured Programs)は、このアルゴリズムにもとづく試作ツールであり、内部処理方法について簡単な例をもとに紹介する。

構造化変換はそれ自体ではそれほど実用性がないが、ソースプログラムの制御構造を把握しているため、単体テスト段階での構造テストへの応用が考えられる。単体テストは、機能テストと構造テストに分けられ、構造テストはさらに経路テストと分岐テストに分けられる。経路テストはコストの点で問題があり実施が困難だが、分岐テストではテスト網羅率が不十分である。そこで分岐テストの範囲を越えた追加テストを提言するための支援として、構造化変換アルゴリズムの一部を改良して、そのモジュールに含まれる経路テストの範囲の全てのパスを順序番号の形で出力する支援ツールＵＳＴ(Unit test Support Tool)の開発を試みた。しかし対象とするプログラムの構造によっては膨大なパスが出力されることがある。このことから構造テストにおいて、経路テストと同等のバグ発見効果をもち、テストパス数を最小にする、テストパス選択の最適化の可能性について言及する。

AN APPROACH TO RESTRUCTURING OF NON-STRUCTURED PROGRAMMING AND THE APPLICATION (in Japanese)

Yukiya AOYAMA

Product Assurance Laboratory, IBM Japan, Ltd.
1, Kirihara-cho, Fujisawa-shi, Kanagawa-ken 252 Japan

An algorithm which converts 'non-structured source programs' to logically equivalent 'structured source programs' is introduced in this paper. The algorithm is based on the 'binary tree' concept. ART-SP(Automatic Restructuring Tool to Structured Programs), is the prototype of the above algorithm, is also introduced.

Although the conversion by the alogrithm does not seem to be practical, it can be applied to Unit testing as a structured testing approach, because the control structure in the source program can be grasped by the algorithm.

Structured testing can be classified as two types of testing, Path and Branch. Path testing is hard to execute due to cost. On the contrary, Branch testing has test coverage problems. To find and recommend additional test paths beyond the range of Branch testing, we developed a supporting tool, UST (Unit Test Support Tool), using improved conversion algorithm. The UST outputs all paths(within the rangeof branch testing) included in the objective module as lines of sequence numbers. As a result of using UST, a large number of paths were output as expected. Based on this result, a new range which has the same testing coverage as path testing and has a minimum number of test paths is proposed in structure testing.

1. はじめに

近年、Dijkstra[1]に始まる構造化プログラミング技法は定着しつつあり、またフローチャートに代わるいくつかの木構造チャートが提案、実用化されている。[2] 本稿では、構造化されていないソースプログラムを、論理的に等価な構造化プログラムに変換する二進木を用いた構造化変換アルゴリズムと、それにもとづき試作されたツールART-SPを紹介する。また応用としてプログラムの単体テスト(構造テスト)に焦点をあて、ART-SPの機能の一部を改良したツールＵＳＴの、単体テスト検査への適用例について述べる。さらに適用結果に対する考察にもとづき、構造テストのテストパス選択の最適化に関する提案をおこなう。

2. 構造化変換アルゴリズムとART-SP

２．１　構造化変換アルゴリズム

本節では、構造化変換アルゴリズムについて紹介し、それにもとづく試作ツールART-SPを２．２節で紹介する。

プログラムに含まれる各命令は以下のように分類できる。

・開始命令 ； プログラムの先頭の命令(例えばPL/IではPROC)。以下、記号Ｓで表す。

・終了命令 ； プログラムの最後の命令(PL/Iではﾌﾟﾛｸﾞﾗﾑの最後のEND)。以下、記号Ｅで表す。

・分岐命令 ； プログラムの実行順序を決定するための判断を行なう命令。以下、記号Ｊn (n=1,2‥) で表す。

・制御命令 ； プログラムの実行順序を制御する命令。制御命令は記号化しない。

・実行命令 ； 上記以外の各命令(演算命令、入出力命令、宣言文など)。以下、記号Ｉn (n=1,2‥) で表す。

2.1.1 構造化展開

非構造化プログラムは、それと等価な制御構造をもつ木構造(二進木)で表現できる(SELECT文などもIF文の複合と考える)。以下に(非構造化)プログラムのフローチャート表現と、それに対応する木構造表現の例を示す。(図中の各Ｊnの左側は真、右側は偽とする。)

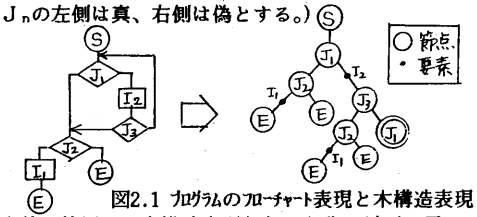


図2.1 ﾌﾛｸﾞﾗﾑのﾌﾛｰﾁｬｰﾄ表現と木構造表現

本稿で使用する木構造表現各部の名称を以下に示す。

・Ｓ，Ｊn，Ｅを節点、Ｉnを要素とよぶ。

・木構造の節点の上から下へ連結したつながりのうち、上位にある節点を親節点、下位の節点を子節点とよぶ。

・同一木構造中に同一節点が二度以上現れることがある(図2.1ではＪ１，Ｊ２)。このうち、任意の節点の子節点に同一の節点が現れた場合、子節点を二重丸で表し重複節点とよぶ(図2.1のＪ１)。

・任意の節点の子節点のうち、最上位にある節点を、その節点が親節点の真の側にある場合は、真の最上位子節点、偽の側にある場合は、偽の最上位子節点とよぶ。(図2.1でＪ１の真の最上位子節点はＪ２、偽の最上位子節点はＪ３)

・木構造の節点の上から下へ連結したつながりのうち、最下位にある節点を葉とよぶ。木構造中のそれぞれの葉は、Ｅまたは重複節点である。またＳは葉に、Ｅは親節点になることはできない。

2.1.2 木構造表現の集約

木構造表現は、(非構造化)プログラムの制御構造を、開始命令から終了命令までを縦に展開した等価表現であり、葉が全てＥならば構造化されているが、葉に重複節点を含む場合は、その部分は構造化されていない。また構造化されている部分についても、単に非構造化プログラムの制御構造を展開しただけであり、冗長な構造をもっている。これらを以下に述べる規則によって IF～END および、DO WHILE～END の形に集約し、図2.2のようにＳとＥの間に要素、または要素となった結合が直列に接続されたら構造化は終了する。これによって、元の非構造化プログラムと等価な、集約された構造化プログラムが作成される。

規則１ 分岐結合

任意の親節点の、真と偽の最上位子節点が同一節点の場合、分岐結合として集約する。親節点と、両子節点の間に同一の要素が含まれている場合には、その要素は分岐結合の外に出す。また分岐結合はそれ自体、要素になったとみなす。

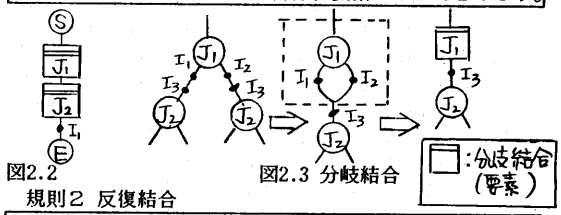


図2.2　　図2.3 分岐結合

規則２ 反復結合

任意の親節点の、真(または偽)の最上位子節点が、親節点の重複節点で、偽(または真)の最上位子節点が、親節点と異なる場合、反復結合として集約する。反復結合は要素になったとみなす。

(注)真と偽の最上位子節点がともに親節点の重複節点の場合、無限ループを意味する。

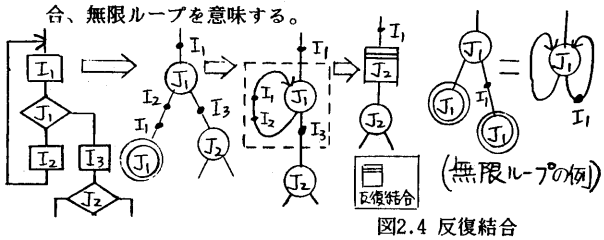


図2.4 反復結合

規則３

任意の親節点に、最上位でない重複子節点が存在し、両節点の間に親節点とは異なる節点がＮ個存在する場合、このままではこれ以上構造化できないので、ダミーフラグを付加して集約する。この結合も要素になったとみなす。

Ｎ＝１の場合を以下に示す。

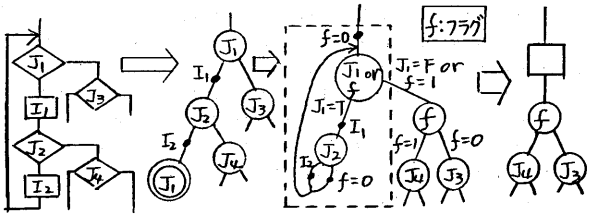
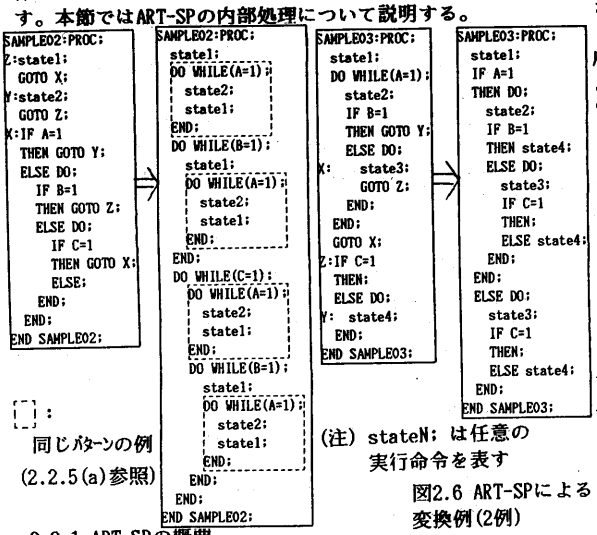


図2.5

2．2 ART－SP

ART-SP(Automatic Restructuring Tool to Structured Programs)は、２．１でのべたアルゴリズムにもとづく試作ツールである。ART-SPによる構造化変換例を図2.6に示す。本節ではART-SPの内部処理について説明する。



(注) stateN; は任意の実行命令を表す

図2.6 ART-SPによる変換例(2例)

2.2.1 ART-SPの概要

ART-SPは、PL/I言語で記述された約1000ステップのプログラムで、以下の３部分から構成されている。

(a) 構文解析部；入力ファイルから非構造化ソースプログラムを読みこみ、以下の表を作成する；

・命令対応表；ソースプログラムを構成する各命令を記号化し、両者の対応を示した表。

・制御フロー表；ソースプログラムの各命令の制御フロー(分岐命令間の対応)を記述した表。

(b) 変換部；制御フロー表の内容を、いくつかの規則を適用して構造化変換する。

(c) 復元部；命令対応表と、構造化変換された制御フロー表から、構造化されたソースプログラムを作成し、出力ファイルに書きだす。

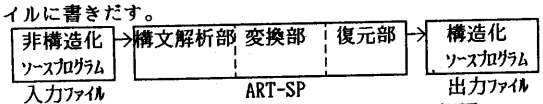


図2.7 ART-SP概要

以下の節でART-SPの各部分の処理を説明する。なお、ART-SPは試作品のため、機能的に次の制限がある。

・分岐命令としてIF, DO WHILEのみを扱う。

・制御命令としてTHEN (DO), ELSE (DO), END, GOTO のみを扱う。

・２．１で述べた規則３の機能は含まれていない。

2.2.2 構文解析部

構文解析部では、入力ファイルから非構造化ソースプログラムを読みこみ、命令対応表と制御フロー表を作成する。

(a) 命令対応表；ART-SPでの処理を簡単にするため、ソースプログラムに含まれる各命令を、２．１で示したように分類、記号化し、元の命令と記号の対応を命令対応表に作成する。例を図2.8の(1)に示す。(なお図2.8はART-SPによる処理の一連の流れを示しており、以下の各節の説明中で参照される。)

(b) 制御フロー表；制御フロー表は、開始命令Ｓとその最上位子節点、および各分岐命令J_nとその真と偽の最上位子節点との対応を示す表である。従って制御フロー表全体で、2.1.1で述べたプログラムの木構造を表現している。

以下に制御フロー表の作成方法を示す。(図2.9)

i) 各分岐命令ごとに、真と偽それぞれの側の、次に実行される命令にフラグＴとＦをつける。

ii) 命令の通常の実行順序(上から下への実行)と異なる順序の部分に、その実行順序に従ってポインターを付加する。

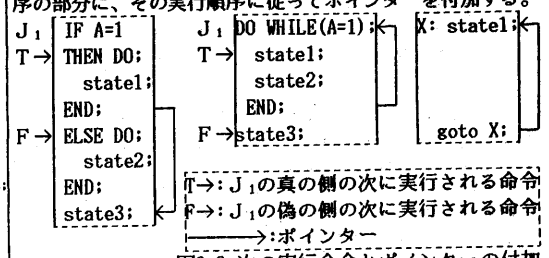
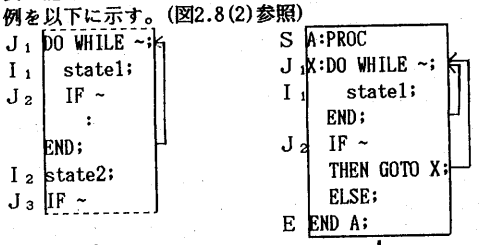


図2.9 次の実行命令とポインターの付加

iii) 各分岐命令ごとに、ii)でフラグを付けた、真と偽の次に実行される命令から、命令の実行順序に従って、最初の分岐命令または終了命令(最上位子節点)に到達するまで探索する。(この場合、元の分岐命令と次の分岐命令が同一の可能性もある。) 分岐命令間の対応づけの結果とともに、両分岐命令の途中に含まれる実行命令を、制御フロー表に記録する。開始命令についても同様の処理を行なう。例を以下に示す。(図2.8(2)参照)



親節点	節点間の要素	最上位子節点
J_1 T	I_1	J_2
F	I_2	J_3

T:親節点の真の側
F:親節点の真の側

親節点	節点間の要素	最上位子節点
S		J_1
J_1 T	I_1	J_1
F		J_2
J_2 T		J_1
F		E

図2.10 作成された制御フロー表の例

2.2.3 変換部

変換部では、2.1.2で述べた木構造表現の集約をおこなう。具体的には2.1.2の規則１と規則２を、前節で作成された制御フロー表の各親節点に対して繰り返し適用する。図2.8での適用例を以下にしめす。(以下の説明中の番号は、図2.8内の番号に対応する。)

(a)規則１の適用；各親節点に対し、以下の操作を行なう。

(3) 同一の親節点の、真と偽の最上位子節点が同一のものをさがす。図2.8では親節点がJ_2の場合が該当する。

(4) J_2全体を分岐結合とする。

(5) (4)で分岐結合になった親節点(図2.8ではJ_2)を最上位子節点とする組合せをさがす。図2.8では、親節点がJ_1の、真の最上位子節点J_2が該当する。

(6) 分岐結合となった節点(J_2)を要素に変化したとみなし、(5)を満足する組合せの、節点間の要素の欄に移す。

(7) (5)を満足する組合せの、真の最上位子節点の欄に、

分岐結合の最上位子節点(図2.8では、親節点J_2の最上位子節点であるJ_1)を移す。

(b)規則2の適用：各親節点に対し、以下の操作を行なう。

(8) 同一の親節点の、真または偽の最上位子節点が、親節点と同一のものをさがす。図2.8では親節点がJ_1の場合が該当する。(真の最上位節点もJ_1なので)

(9) J_1全体を反復結合とする。

(10) 分岐結合または反復結合になっていない部分のうち(9)で分岐結合になった親節点(図2.8ではJ_1)を最上位子節点にもつ組合せをさがす。図2.8では、親節点がSの最上位子節点J_1が該当する。

(11) 反復結合となった節点(図2.8ではJ_1)を、要素に変化したとみなし、(10)を満足する組合せの節点間の要素の欄に移す。(これは(6)と同様の処理である。)

(12) (10)を満足する組合せの最上位子節点の欄に、分岐結合の親節点とは異なる最上位子節点を移す。(図2.8では、親節点J_1の偽の最上位子節点E)((7)と同様の処理。)

規則1と規則2を繰り返し適用し、親節点Sの最上位子節点がEとなった時点で、変換部の動作は停止する。

(注) 2.1.2の規則3の形態が現れた場合は、規則3の処理を行なう必要があるがART-SPではこの機能はない。

2.2.4 復元部

復元部では、命令対応表を使って、変換部で集約された制御フロー表の記号列を、構造化されたソースプログラムの形に復元する。この過程を図2.8の例で説明する。(以下の説明中の番号は、図2.8内の番号に対応する。)

(13) 親節点がSの行を縦に並べる。

(14) SとEの間に、J_nが存在しなくなるまで以下の処理を行なう。

(14.1) J_nが制御フロー表上で反復結合の場合

まず DO WHILE(J_n); と END; を縦に並べ、その間に、制御フロー表を参照して、J_nのT側の節点間の要素の欄に含まれているJ_pとS_qを縦に並べる。また、END; の下に、J_nのF側の節点間の要素の欄に含まれているJ_pとS_qを縦に並べる。最後に元のJ_nを、以上の処理で作成した記号列で置き換える。

(14.2) J_nが制御フロー上で分岐結合の場合

IF(J_n)、THEN、ELSEを縦に並べる。制御フロー表を参照して、THENにはJ_nのT側の、節点間の要素の欄に含まれているJ_pまたはS_qを付加する。(ただしJ_p、S_qが複数存在する場合は、THENの代わりにTHEN DO;、END;を縦に並べ、その間にJ_p、S_qを縦に並べる。) ELSEについても同様の処理をする。最後に元のJ_nを、以上で作成した記号列で置き換える。

(15) (14)の処理が終了したら、作成された記号列に含まれるS、E、S_nと、DO WHILEとIFに付加されたJ_nを、命令対応表を用いて実際の命令に置き換え、結果を出力ファイルに書きだす。

2.3 ART-SPの応用について

以下にART-SPの応用の可能性について述べる。

(a)構造化変換機能

・過去に作成されたかなり複雑な非構造化プログラムでも構造化プログラムに変換することにより、内部構造を把握することができる。また復元モジュールを改造することによって、構造化された形式で、フローチャートや木構造チャートの図形出力をすることが可能である。

・前述の規則2に示した無限ループがプログラムに含まれている場合は、変換の過程でその発見が可能である。

・実行順序が決定している言語であれば、どのような言語で書かれたプログラムでも構造化変換は可能である。

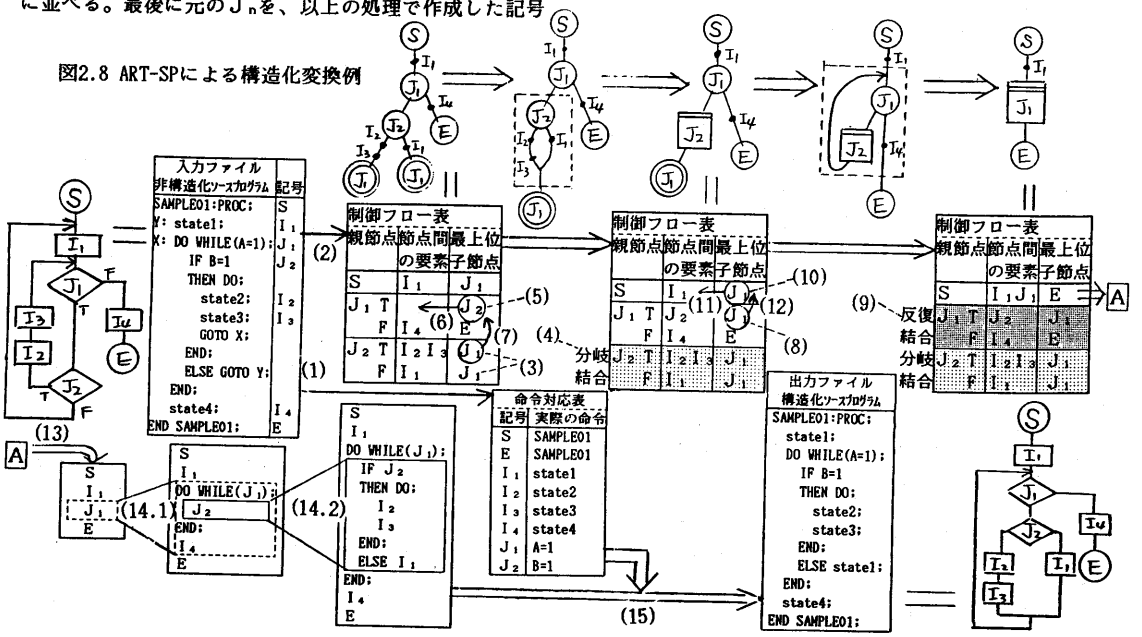
・応用ではないが、ART-SPによる変換では、変換後に同じパターンの命令列が複数出現することがある。(図2.6) 復元モジュールを改造して、これらをまとめてサブルーチン化することは可能である。

しかし現実には、構造化変換機能はそれ自体ではそれほど実用性はないと考えられる。

(b)プログラムテストへの応用

プログラムの制御構造を、制御フロー表として把握しているので、プログラムテスト、特に単体テストの構造テストへの応用が考えられる。これについては3章で述べる。

図2.8 ART-SPによる構造化変換例



3. ART-SPの単体テストへの応用

プログラムのモジュール単位のテストとして、機能(ブラックボックス)テストと構造(ホワイトボックス)テストがある。単体テスト段階では、まず機能テストを行ない、テスト終了後にテスト網羅率を支援ツールを使用して測定し、その結果に基づき、機能テストで未網羅の部分について構造テストの観点からテストを行ない、次第にテスト網羅率を向上していく方法が一般的である。[3] 構造テストはさらに (a) 経路テスト；対象モジュールに含まれる論理的に可能な全てのパスのテスト。(b) 分岐テスト；対象モジュール内に含まれる各分岐の、真と偽の両方を少なくとも一度通過させるテスト。(つまりモジュール内の各命令を少なくとも一度実行させる。)の二通りの方法に分類される。経路テストは、対象モジュール内の分岐の構造によってはテストパスの数は膨大になる可能性があるので全て実行するのは難しい。これに対し、分岐テストではテスト網羅率の面で問題が残る。このように、両テストはそれぞれ一長一短があるが、一般的に単体テストでは、分岐テストを中心にテストが行なわれることが多い。

(注) 正確には分岐テストはさらに、IF文における THEN または ELSE命令がない場合、その経路はテストパスとはみなさないC$_0$メジャーと、テストパスとみなすC$_1$メジャーに分類される。[3]

3.1 単体テスト検査支援ツール(UST)開発の背景

プログラムの単体テストで、テスト終了後に次の観点から、テスト網羅率の検査を行なうという必要性が生じた。

i) 実行されたテストケースが、単体テスト終了基準(分岐テストの完了)を達成しているかどうかのチェック。(USTとは別の支援ツールAによってこの基準の検証を自動的に行なった。)

ii) 実行されたテストケースで十分かどうかのチェック。分岐テストを達成するだけでは、テストとして不十分な場合がある。例えば図3.1で、分岐テストでは、xとyのケースを実行すればよいので、BからCに至るパスが、通常の処理形態にもかかわらずテストされない。このようなテストケースを、追加ケースとして指摘するために、試みとして、さらに次の順序で検証を行なった。(図3.2)

(a) すでに実行されたテストケースの把握；これは支援ツールAの実行結果のデータを使用することにした。

(b) 実行可能な全てのテストパス(経路テストの範囲)の把握；このデータを得るため、以下に述べるART-SPを応用した支援ツール(UST:Unit test Support Tool)を開発した。

(c) 実行可能でかつテストされていないパス((a)と(b)の差)の把握；今回は手作業にて比較した。

(d) (c)のうち、追加したほうが望ましいテストケースの判断；この判断の自動化は難しいため、テスト対象モジュールのソースプログラムリストにもとづいて、機能の点から手作業で判断し、追加テストケースを選択した。

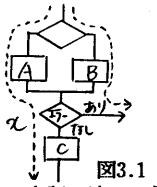


図3.1

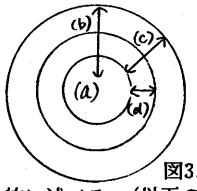


図3.2

上記の流れを具体的に述べる。(以下の説明中の番号は図3.3内の番号に対応する。)

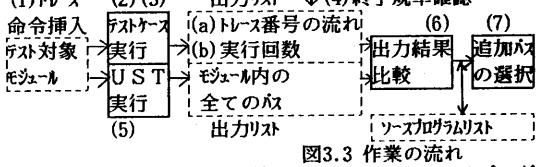


図3.3 作業の流れ

(1) あらかじめ、テスト対象モジュールのソースプログラム(ただし、今回のソースプログラムはあらかじめ構造化プログラムの形式でコーディングされていた)に含まれている各分岐命令(IF, DOなど)の真と偽の両側に、順序番号付きのトレース命令を挿入する。(図3.4)

(2) 次に、そのモジュールに対して用意された、全てのテストケースを実行する。

(3) その結果、テスト終了基準確認のための支援ツールAによって次のデータが出力される。

(a) 各テストケースごとに、テストで通過したパス上に存在する、トレース命令の順序番号(以後トレース番号)の一連の流れ

(b) 各トレース番号ごとに、実行された回数。

例えば図3.4のテスト対象モジュールでは

(テストケース1) トレース番号 14 -> 15 -> SUBEXIT

(テストケース2) トレース番号 14 -> 16 -> 18 -> SUBEXIT

の2つのテストケースを実行したとき、図3.5に示すテスト結果リストが出力される。

(4) 上記(b)を検討し、全てのトレース番号が1回以上実行されていれば、そのモジュールに対し分岐テストが完了した事になる。

(5) 上記の作業とは別に、テスト対象モジュールのソースプログラム自身を入力として支援ツールUSTを実行する。その結果そのモジュール内に存在する可能な全てのパス(経路テストの範囲のパス)がトレース番号の一連の流

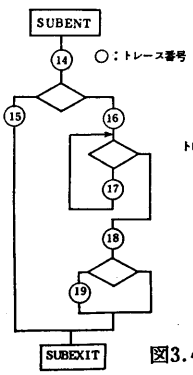


図3.4

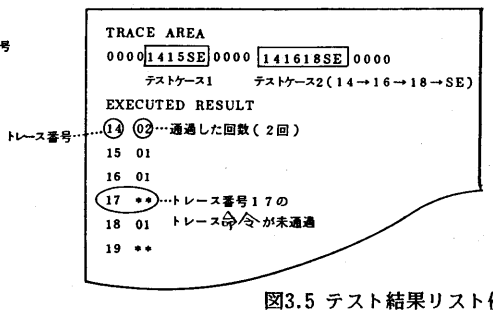


図3.5 テスト結果リスト例

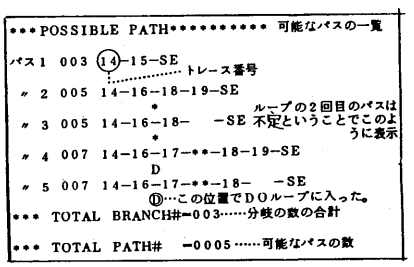


図3.6 USTの出力例

れとして出力される。(図3.6)
(6) 図3.5と図3.6の出力結果を手作業で比較し、テストされていないテストパスをチェックする。
(7) それらのパスのなかで、論理的に意味があり追加テストケースとした方が良いと思われるパスを、ソースプログラムリストを参照して手作業で選択する。

3．2 ＵＳＴの使用結果と問題点

ＵＳＴはART-SPの構文解析モジュールを、作成した制御フロー表を展開して、開始命令から終了命令までの各パスを順序番号の流れとして出力するように改造したツールである。ただしソースプログラムにループが存在する場合はループを一回まわし、ループした範囲をＵＳＴの出力結果に表示する(図3.6)。また論理的にありえないパスの除去は行なっていない。(図3.7) (注)類似ﾂｰﾙとして､文献4)5)6)がある。

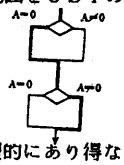


図3.7 論理的にあり得ないﾊﾟｽの例

以下にＵＳＴの使用結果と問題点を示す。
・実際のテスト結果とＵＳＴで出力されたテストパスの比較を、手作業で行なったので時間がかかった。ただしこの作業はツールによる自動化が可能である。
・ＵＳＴでは、対象モジュールの経路テストの範囲のパスが出力されるので、予想どおりモジュール構造によっては著しく多くのパスが出力された。ところが３．１の(7)の手作業によるパス選択の段階で、これらの経路テストの範囲のパスの大部分は、ソースプログラムの論理構造から見て、テストケースとしては相互に冗長であると思われた。このことから、経路テストあるいは分岐テストに置き変わる新たなテストパスの選択基準があるのではないかと考えられた。これについて３．３節で考察する。
・対象プログラムが大規模であったので、支援ツールの利用によって検査効率は向上した。ただし指摘した追加テストケース自体がどの程度有効であったかについては、単体テストでのバグの発生状況が記録されていないため調査していない。ただし分岐テストの範囲を越えたテストを行なったことにより、ある程度の効果はあったと考えられる。[4)]
・今回のように、テストを実行した後、その結果に基づいて追加のテストケースを指摘するためにＵＳＴなどの支援ツールを使用するよりも、テストケース作成の補助手段として使用する方が有効であろう。

3．3 構造テストにおけるテストパス選択の最適化

3.3.1 最適化の目的と方針

３．２節での考察のように、経路テストの範囲では、モジュールに含まれる可能なテストパスの数は、モジュールの構造によっては膨大になり、支援ツールによる効率向上には限界がある。そこで本節では、３．１で述べた、経路テストと分岐テストの持つそれぞれの問題点を解決するため、次のような属性を持つ新しいテスト区分について検討する。なお以下の方法はツールによる処理を前提としているが、実際のツールは作成していない。

現行のﾃｽﾄ区分	問題点	新しいﾃｽﾄ区分の持つべき属性
分岐テスト	テスト網羅率	経路テストと同等のバグ発見効果（テスト網羅率の確保）
経路テスト	コスト	テストパスの数の最小化（コストの低減）

新しいテスト区分は、テストパスと、それによって発見されるバグ範囲の対応を考えると次のようになる。

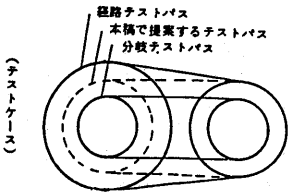


図3.8

まず若干の用語の定義を行なう。
・プログラムは一般に、開始命令Ｓ，、終了命令Ｅおよび各分岐命令J_nを節点とし、J_n間の制御フローを辺とする有向グラフで表現され、これを制御フローグラフと呼ぶ。
・各実行命令および、分岐命令に含まれる条件式を命令とよびInで表わす。(2章での定義と若干異なることに注意)
・隣り合うJ_n間の制御フローをCnで表す。Cnは途中に含まれるInの集合である。
・ＳからＥまでの制御フローのつながりをテストパスと呼びTnで表わす。Tnは途中に含まれるCnの集合である。

3.3.2 経路ﾃｽﾄと同等のﾊﾞｸﾞ発見効果(ﾃｽﾄ網羅率の確保)

新しいテスト区分が、経路テストと同等のバグ発見効果をあげるためには、次の二つの条件を満足する必要がある。

（ａ）分岐テストは少なくとも実行する

経路テストと同等のバグ発見効果をあげるために、全ての命令は一度は実行させる必要があることは自明であり、従って少なくとも分岐テストは満足する必要がある。

（ｂ）冗長でないテストパスの選択

経路テストの範囲のテストパスは、ソースプログラムの構造によっては膨大な数になることもあるが、これらの中には相互に冗長なパスが数多くあると考えられるので、冗長性を除くために、実行すべきテストパスを次のような観点から選択する。

テストパス選択の条件

固有のテストパスでしか発見出来ないバグが存在する可能性のあるテストパスは、かならず選択する。

例えば次のようなプログラムで、テストパスはT_1~T_4の4通り考えられる。ここで図中×で示す部分にバグがあるとする。このバグはその性質により、次の二通りの発見のされ方が考えられる。(図3.9)
(ｉ) T_2とT_4のどちらのテストパスを実行しても発見できる場合
(ii) T_2のみで発見できて、T_4では発見できない場合(あるいはその逆の場合)

(ｉ)のようなバグは、T_2とT_4のいずれのテストパスを設定しても発見できるので、分岐テスト、即ち前述の(ａ)の条件が満足されれば発見できるので問題はない。ところが(ii)のようなバグは、この例の場合、テストパスT_2を必ず選択しないと発見できない。従ってこのような、固有のテストパスでしか発見出来ないバグが存在する可能性のあるテストパスは、必ず選択する必要がある。

さて、このような性質を持つテストパスは、具体的には何を意味するのだろうか。筆者はこれを、そのテストパスを構成する各制御フロー上に存在する命令間に、実行順序の観点からの『関係』がある場合を意味すると考えた。(図3.10) この定義により、前述の『テストパス選択の条件』は次のように置き換えられる。

任意の複数の制御フロー上に存在する、少なくとも一組の任意の命令間に関係がある場合、それらの制御フローにまたがるテストパスを必ず一つは選択する。

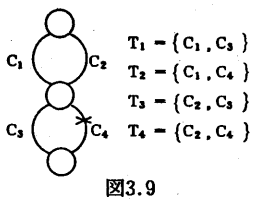


図3.9

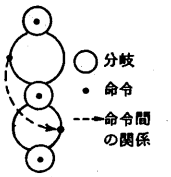


図3.10

以下に、命令間の関係を定義する。

・実行命令Inにおいて左辺の変数を更新変数と呼びM(In)、右辺の変数を参照変数と呼びR(In)で表わす。(実際は各命令の参照変数は2個以上の場合もあるが、以下では簡単のため1個とする。)(図3.11)

・分岐命令の条件式Inでは、左辺の変数、右辺の変数ともに参照変数と呼び、R(In)で表わす。(図3.11)

・命令IpとIqがこの順序で実行され、IpとIqの途中に存在する任意の命令Irとの間に

$$(M(Ip)=M(Iq)) \wedge (M(Ip)\neq M(Ir))$$

が成立するとき、命令Ipの次の命令から命令Iqまでの各命令を、命令Ipの有効範囲と呼びA(Ip)={Ip+1,・・,Ir,・・,Iq}で表す。ただし命令が分岐命令の条件式の場合は、更新変数を持たないので有効範囲を持つことはない。

・任意の命令IpとIrが(Ir∈A(Ip)) ∧ (M(Ip)=R(Ir))を満足するとき、命令IpからIrへの命令間の関係(以下、命令フロー)が存在すると呼び、Ip→Irで表す。ただし、命令が分岐命令の条件式の場合は、有効範囲を持たないので命令フローは発生しない。(図3.12)

以上の定義により、プログラム内の命令は各命令を節点、命令フローを有向辺とする有向グラフ(命令フローグラフ)で表現される。従ってプログラムは制御フローグラフと命令フローグラフの二つのグラフで表現される。(図3.13)

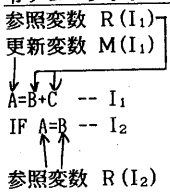


図3.11

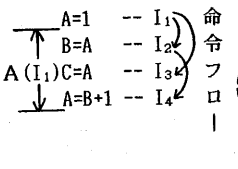


図3.12

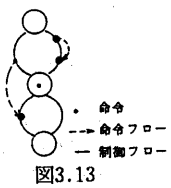


図3.13

3.3.3 テストパスの数の最少化(コストの低減)

本節では、3.3節の目的である、プログラムに含まれる全てのテストパスTn(経路テストの範囲のパス)のうち、冗長性を省いた、数の最も少ないテストパスの集合の決定方法について考察する。まず用語の定義を行なう。

・命令Inから発生する命令フローの数を$d^+(In)$、命令Inへ到達する命令フローの数を$d^-(In)$としたとき、$d^-(In)=0$を満足するInを始点命令、$d^+(In)=0$を満足するInを終点命令と呼ぶ。(図3.14)

・Ipが始点命令、Iqが終点命令で、かつIpとIqが同一方向の命令フローで連結されているとき、IpとIqの間に存在する一連の命令を命令パスと呼び、Pn={Ip,・・・・,Iq}で表す。また終点命令がIqである全ての命令パスの集合を、Ψ(Iq) = {Pm,・・・・,Pn}と表す。(図3.15)

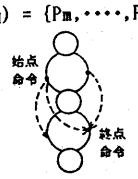


図3.14

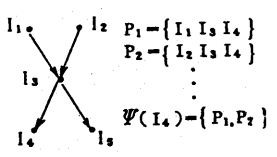


図3.15

つぎに準備として次の三つの操作を行う。(これらの操作はツールを想定したものであり、本論と直接関係ない。)

準備1．命令パスの要素の変換 (図3.16)

テストパスTnと、命令パスPnを比較するため、PnとTnの要素の属性を同一にする必要がある(Tnの要素属性は制御フローCn, Pnの要素属性は命令Inであるため、両方ともCnにする)。そこで命令パスPnの要素を、命令Inから、Inの存在している制御フローCnに変換し、変換後の命令パスをP'nで表す。従って、PnとP'nは次の条件を満足する。

(a) 任意のIp∈Pnに対し、Ip∈Cq∈P'n を満足するCqが存在する。

(b) 任意のCq∈P'nに対し、Ip∈(Pn∩Cq)を満足するIpが存在する。

準備2．分岐命令の条件式の移動 (図3.17)

分岐命令の条件式は制御フロー上には含まれないが、その分岐命令の真と偽の両方の制御フローの前提条件となるので、その条件式に関する命令フローを引き継いで、真と偽の両側に移動する操作を行う。(この操作は、分岐命令の条件式を、一般の命令と同等に扱うために行ない、実際に支援ツールとして具体化する際に必要となる。)

準備3．対称命令の付加

図3.18のような場合、C_1とC_5を通るテストパスとして、T_1={C_1,C_3,C_5} T_2={C_1,C_4,C_5}の2本のパスが考えられ、テストパスT_1ではC_3上の命令I_1によって変数Bの更新が行なわれているので、T_1上に命令パス$I_1 \rightarrow I_3 \rightarrow I_4$が存在している。ここで、テストパス$T_2$においても、$C_4$上でB＝Bという更新を伴わない命令が実行されたと考えると、T_2のテストパスもテストする必要がある。つまり命令I_4を検証するため、I_3においてBの更新がされた場合の他に、Bの更新がされなかった場合の検証が必要となる。

以後の操作でこのパスが選択されるようにするため、C_4上にI'_3という仮の命令を付加する。この命令を対称命令と呼ぶ。この操作を一般化すると次のようになる。

ある分岐の両側の制御フローをそれぞれCp,Cqとし、その下位に存在する任意の制御フローをCrとしたとき、(Ip∈Cp) ∧ (Ip→Ir∈Cr) を満足するIpが存在し、かつ (Iq∈Cq) ∧ (Iq→Ir∈Cr) を満足するIqが存在しなければ、Cq上にIpに対応する対称命令I'pを付加する。

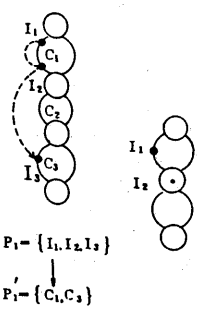


図3.16

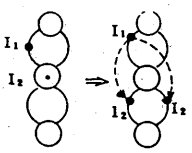


図3.17

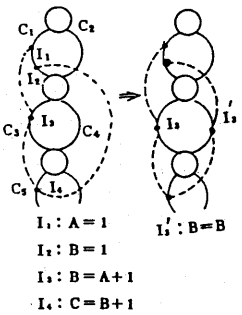


図3.18

これらの準備の後、テストパスと命令パスを比較して、異なる制御フロー間を渡る命令パスが存在するテストパスは必ず選択するような操作をおこなう。例えば図3.19では次のようになる。

i) テストパスT_1={C_1,C_3}は必ず実行されるべき

ii) テストパスT_1={C_1,C_3,C_5} または T_2={C_1,C_4,C_5}のうち、少なくとも一つは必ず実行されるべき

iii) テストパスT_1={C_1,C_4,C_5}は必ず実行されるべき

このように、異なる制御フロー間にまたがる各命令パスそれぞれにおいて、必ず実行すべきテストパスが存在する。また、いくつかの命令パスを複合したときにも、テストパスが存在することもある。従ってこれらのテストパスの中から、次に示すように冗長性を省いて最少の数のパスを選択する必要がある。

まず終点命令が同一の命令パスの集合を考えると、この集合のベキ集合の各要素、即ち命令パスの組合せによって、それに対応するテストパスが実際に存在する場合と、存在しない場合がある。例えば図3.20で、終点命令はI_5であり

$P(I_5)=\{P'_1,P'_2,P'_3\}$

$P'_1=\{C_1,C_4,C_6\}$

$P'_2=\{C_2,C_6\}$

$P'_3=\{C_3,C_6\}$

であるが、このうちP'_1とP'_2を組合せた命令パスを通過するテストパスは、制御フローC_1とC_2は独立なのでありえない。これに対しP'_2とP'_3を組合せた命令パスを通過するテストパスは、$C_2\rightarrow C_3\rightarrow C_6$を通る場合が考えられる。

このように、同一の終点命令を持つ命令パスの集合ごとに、そのベキ集合の要素それぞれについて、テストパスが存在するかどうか検討して、存在する場合にはそれらのテストパスを選択する操作を行なう。これを一般化すると次のようになる。

あるプログラム内のテストパスの集合を$T=\{T_1,\cdots,Tm\}$，任意の終点命令Ipの命令パスの集合を$\Psi(Ip)$とする。このとき$\Psi(Ip)$の任意の部分集合ϕにおいて、$Ti\subset\phi$を満足する全てのTiを含むTの部分集合を考え、それらのうち要素数の最も少ない部分集合が、冗長性を省いたテストパスの集合となる。

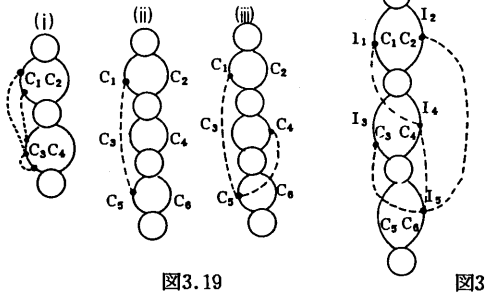


図3.19

図3.20

3.3.4 テストパス選択による効果の把握

以上述べたテストパス最適化の方法は、ツールによる処理を前提としているが、実際にツールを作成していないので、当初の目的(経路テストと同等のバグ発見効果、パスの数の低減)がこの方法で達成できるか未知数である。参考までに、あるプログラムについて、どの程度パス数が減少するかを机上で調査したので、その結果を示す。ただし、パス数の減少の割合については、対象プログラムの論理構造に依存するので、一概に論ずることはできない。

対象ﾌﾟﾛｸﾞﾗﾑｻｲｽﾞ	；２００ｽﾃｯﾌﾟ (PL/I)
分岐ﾃｽﾄﾊﾟｽの数	；５本
経路ﾃｽﾄﾊﾟｽの数	；７５０本
新ﾃｽﾄ区分のﾃｽﾄの数	；約１００本

3．4 単体(構造)テスト支援ツールの今後の展望

(1) テストケースデータの作成

今回のようなトレース番号の表示を一歩進めて、パスがツールなどにより一意的に決定した場合、各パスの実際のテストケース入力データの作成が考えられる。ただしプログラムの制御構造の他に、各命令自体の持つ意味の認識が必要となる。

(2) 新しいテスト区分にもとづくツールの作成

・新しいテスト区分にもとづくツールを作成した場合、使用方法として、単体テストで機能テストを行なった後で、そこで実行されたテストパスの情報を入力として、新しいテストケース区分に基づいた追加テストケースを選択するという方法が考えられる。

・現在、テスト網羅率のツールによる測定が広く行なわれている。[5)6)] しかしC_1メジャーは、テスト網羅率の尺度としては疑問が残り、C_1メジャーに置きかわる新しい尺度として、命令フローがどの程度テストされたかという尺度と、その測定用のツールが考えられる。

しかし、次のような問題点もある。

・論理的にありえないテストパスの除去について；この問題を解決するためには、前述のようにツールによる各命令の意味の認識が必要となる。

・ループ処理；ループ回数は、プログラム内部で動的に変化する可能性があり、また固定されている場合でもループ回数が多い場合には、その扱いが難しい。ただし、ループの回数を例えば２回と考えた場合、命令間の対応づけは可能である。(図3.21)

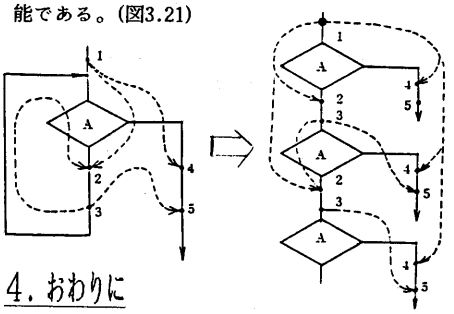


図3.21

4．おわりに

本稿では構造化変換ツールART-SPをとりあげ、そのアルゴリズムを紹介した。変換ツールとしての機能は実際にはあまり需要がないと思われるが、ART-SPのプログラムの制御構造の把握という観点から、モジュールの制御構造に着目したテストである構造テストへの応用が考えられ、ＵＳＴを作成した。さらにその適用結果から、パス選択の最適化の可能性について述べた。しかしいずれも試作または理論の範囲にとどまっているため、ソフトウェアの生産性、信頼性の向上の点からの応用分野を見出すとともに、効果的、実際的なツールに発展させる必要がある。

なお本論の内容は、日本ｱｲ･ﾋﾞｰ･ｴﾑ(株)を代表する意見ではなく、筆者個人の意見であることをお断わりする。

[参考文献]

1) Dijkstra E W et al.『Structured Programming』Academic Press, 1972

2) 林『木構造ﾁｬｰﾄ』日経ｺﾝﾋﾟｭｰﾀ (1984.1.9)

3) 田中 他『手工業的手法からの脱皮を目指すｿﾌﾄｳｪｱﾃｽﾄ』日経ｺﾝﾋﾟｭｰﾀ (1982.3.15)

4) 黒田 他『ﾌﾛｸﾞﾗﾑのﾊﾟｽに基づくﾃｽﾄ法の有効性評価』情報処理学会第27回全国大会 (1983)

5) 福原光一『ｵﾝﾗｲﾝﾌﾟﾛｸﾞﾗﾑのﾃｽﾄ手法の改善』第5回ｿﾌﾄｳｪｱ生産における品質管理ｼﾝﾎﾟｼﾞｳﾑ発表報文集 (1985.9.18,19)

6)『DYANA説明書』Software Research Associates, Inc.