ソフトウェア工学　52－17
（1987・2・14）

ユーザ定義に基づく
プログラムの変則構造の検出

長谷川哲夫[†]　門倉敏夫[†]　深沢良彰[††]
[†] 早稲田大学理工学部　[††] 相模工業大学工学部

プログラムからユーザが指定した静的な属性を検出するツールを作成した．
開発標準を設定してソフトウェアを開発していく場合，詳細の基準はプロジェクトにより異なることが多い．プログラムのコーディング時においては，プログラムの持つ静的な属性という観点から基準を表現することが可能であり，この静的属性を変則構造と呼ぶ．本ツールは，変則構造のユーザによる自由な設定を可能にすることにより，プロジェクト独自の基準に対する違反検出の自動化を目的とする．
本稿では，変則構造の定義方法，プログラムからの検出法，および，定義方法に対する評価について述べる．

"A Detection Method of User Defined Irregular Styles"

Tetsuo HASEGAWA[†], Toshio KADOKURA[†] and Yoshiaki FUKAZAWA[††]

[†] School of Science and Engineering, Waseda University, Tokyo 160 Japan
[††] Faculty of Engineering, Sagami Institute of Technology, Fujisawa, 251 Japan

A static program analyzing tool has been designed and developed. The specific feature of this tool is that a user can define target properties.

Even if a software system is developed according to a standard procedure, each project must have the variety of details in the standard procedure. This tool regards the coding standard as a set of static properties of user programs. In this tool, a target of detection, which is called Irregular Style, can be freely defined by a user.

This paper describes the notation of definition of Irregular Styles, the detection strategy, and the results of its evaluation.

1．はじめに

ソフトウェアの大規模化に対する，信頼性向上のための一手法として，開発過程の各段階での標準化が存在する．この実現のために，いくつか汎用的な開発標準が提唱されている[1],[2]．また，これらの開発標準に従ったソフトウェア開発を支援するための自動化ツールの試作も行われてきた．しかし，実際にユーザがシステムの開発を進めるには，汎用的な標準の他に，かなり詳細な基準をプロジェクト内で独自に設定する必要が生じることが多い[3]．これは汎用的な標準が，システムの目的，開発環境，蓄積されているノウハウなど，プロジェクトごとに異なっている要因を反映していないためである．したがって，プロジェクト独自の基準に対応可能な自動化ツールは，より実用的であり，その意義は非常に高い．

本ツールはソフトウェア開発段階のうち，製作段階，特にプログラムのコーディング時におけるプロジェクト独自の規約に対する違反検出の自動化を目的とする．

2．本ツールの概要

本ツールにおいては，コーディング時の様々な規約をプログラムの持つ静的な属性という観点から捉え，検出が望まれる静的属性をプログラムの変則構造と呼ぶ．ユーザは変則構造モデルを記述することにより，変則構造を定義する．本ツールはターゲット・プログラムを静的に解析し，変則構造モデルに相当する箇所を検出し，出力する．

本ツールで扱う変則構造は構文レベルで検出可能なプログラムの静的属性である．したがって，次のような規約は扱わない．

- ・コメントに関する規約，段付けなどの表記上の規約のように，解析木に展開した後では検出できない規約．
- ・変数名が変数の意味を忠実に表わしているかなどの，ソース・プログラムからでは判定できないような規約．

本ツールで用いている変則構造モデルの記述法は次の２点を重視している．

- ・ユーザの考えている変則構造をできるだけ素直に記述できること．
- ・変数の値の動きや相互関係も記述可能にし，より複雑な静的属性を表わせるようにすること．

この２点を実現するために

- ・モデルの記述内容を，構造記述と条件記述という２種類に分け，それぞれ最適と思われる記述法を用いて記述する．
- ・変数の値の動きを調べるために，部分的な記号実行[4]を行う．

という手法を取り入れた．これらの詳細については次節以降で述べる．

本ツールが対象とする言語は，構文の表現が容易で，変則構造モデルの記述に都合がよい，という理由からPascalを選んだ．

3．変則構造モデルの記述法

変則構造は，「プログラムのいくつかの部分が満たすべき条件」，「条件を満たす各部分の，プログラムの制御構造に対する相対的位置付け」の２点で定義できる．この２点をそれぞれ，「条件記述」，「構造記述」と呼び，これらに「表題」を加えて，１つの変則構造モデルを構成する．

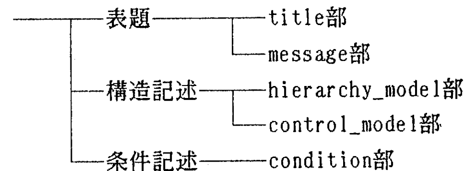

図１　変則構造モデルの構成

以下，変則構造モデルの各部分についての概説，及び，例に沿った解説を挙げる．

(1)表題

表題はtitle部とmessage部から構成される．それぞれ，変則構造モデルの名称，変則構造モデルが表わす変則構造がターゲット・プログラム中に検出された時に出力するメッセージを記述する．

(2)構造記述

構造記述は，変則構造がプログラムの制御構造に依存する場合，または，条件を満たすプログラムの各部分の相対的な位置関係を指定する必要がある場合に記述する．プログラムの構造は，手続き・関数の静的な階層構造と，連接，分岐，反復の３種基本

```
title    :        reassign_without_reference
message :         'reassign some value to a variable before reference it'
hierarchy_model :
        (
        )
control_model :
        (         1 : free
                  2 : free
        )
condition :
        (         (   ,  1   :              assign(#a,#_)    ) and
                  (   , 1~2  : noexist      varref(#a)       ) and
                  (   ,  2   :              assign(#a,#_)    )
        )
```

図２　変則構造「reassign_without_reference」のモデル記述

制御構造とによって記述できる．これらの構造を，それぞれ，hierarchy_model部，control_model部に記述する．構造の記述法はグラフ表現を基にしている．プログラムの構造を考える場合，図形的に考えることが多い．よって，構造を表わすには，図形的に理解できるグラフ表現が適している．

hierarchy_model部においては，ノードは，それぞれの手続き・関数を表わし，アークは，その親子関係を表わしている．記述内容は，各ノード番号，手続き・関数の種別，及び，下位階層に相当するノードへのアークである．

control_model部においては，ノードは，分岐，反復のような特定の構造，プログラム，手続きなどの入口，出口のような特殊な位置，及び，条件を満たすべきプログラムの部分を表す．また，プログラム中のある部分からある部分へ制御が流れる可能性がある時，その間にパスがあるとする．アークはノード間にパスが存在することを示す．この場合のパスは，複数の手続きや関数にわたるパスも考える．記述内容は，各ノード番号と，ノードが表わす構造の種別である．

(3)条件記述部

条件記述は，構造記述中の各ノードで満たさなくてはならない条件，および，構造に関係なく満たさなくてはならない条件を指定する．この条件を表わすために，述語表現を用いる．これは，述語表現は，意味を分りやすく正確に表現できる表現法だとされているからである．ここで用いる述語を，モデル述語と呼び，モデル述語の引数として用いられる変数をモデル変数と呼ぶ．モデル変数の働きはPrologにおける変数と同様で，１つの変則構造モデル内で同一名のモデル変数は同じものを表わす．そして，単一のモデル述語によって，あるいは，モデル述語をand/orによって組み合わせることによって，条件を表わす．ここで，モデル述語には「代入する」，「参照する」のような動作，「型」のような引数の持つ属性を表わすものなどがある．モデル述語は，解析木の特定のパターンに対して成立するものと，プログラム中の『変数』の値がある論理関係を満たした時成立するものがある．いくつかの変則構造モデルで利用するような一般的なモデル述語は，予め本ツール内に用意してあるが，特殊なモデル述語をユーザが追加することもできる．

図２～４に変則構造の例を挙げた，以下この例に沿って，変則構造がどのようにモデル化されるかを説明する．

図２は，「代入後参照されずに再代入される変数が存在する」という変則構造のモデルである．この場合，手続き・関数の階層関係とは無関係であるので，hierarchy_model部に対する記述はない．そして，control_model部での記述は，プログラム中にパスの存在する任意の２点をそれぞれノード１，２とすることを意味する．条件記述のcondition部では，ノード１，２で，同じ『変数』に代入が行われ，さらに２ノード間でその『変数』の参照が無いことを示している．この時，モデル述語「assign」は，代入文に対して成立し，第一引数には『変数』が，第二引数には『式』が返される．同様に「varref」は，『変数参照』に対して成立し，引数には『変数』が返される．また，「#a」はモデル変数であり，す

```
title    :     assign_global_variable
message  :     'assign to global variable in sub module'
hierarchy_model :
        (      1 : ( free )
        )
control_model :
        (      1 : mstart
               2 : free
               3 : mend
        )
condition :
        (      ( 1 , 2   :              assign(#a,#_) and var_no(#a,#no) ) and
               ( 1 , 1~3 : noexist ( var_def(#_,#no,#_)
                                     or fpara_def(#_,#no,#_,#_) )     )
        )
```

図３　変則構造「assign_grobal_variable」のモデル記述

```
title    :     same_condition
message  :     'inappropriate branch structure'
hierarchy_model :
        (
        )
control_model :
        (      1 : branch()
               2 : branch()
        )
condition :
        (      (   , 1 : br_cond( #a )                    ) and
               (   , 2 : br_cond( #b )                    ) and
               (   ,   : relation( #a = #b ) for_any_input      )
        )
```

図４　変則構造「same_condition」のモデル記述

べて同一の『変数』を表わす．ただし，「#_」は無名モデル変数であり，それぞれ別の変数とみなされる．

図3は，「手続き・関数内で大域的変数に対する代入がなされている」という変則構造を表わしている．hierarchy_model部において，手続き・関数の種別は任意とされている．また，同一手続き・関数内で成立すべき事なので，下位階層の記述はない．control_model部の「mstart」，「mend」はそれぞれ，手続き・関数の入口，及び，出口である．モデル述語「var_no」は，各『変数』に対し成立し，その変数の『変数番号』を返す．『変数番号』とは，プログラム中の『変数』に付けられた逐次番号である．「var_def」，「fpara_def」は，それぞれ，『変数定義』，『仮引数定義』で成立し，第2引数に『変数番号』を返すモデル述語である．この構造記述と条件記述の組合わせで，「ある手続き・関数内で，仮引数定義，または，変数定義がされていない変数に対する代入がある」ことを表わしている．

図4は，「あるパス上の2つの分岐条件式が，どのような入力値に対しても，等しい値を持つ」ことを表わしている．ここで用いている「relation」は記号実行により成否を判定するモデル述語で，引数としてモデル変数を含む式を取り，その式が真となる時成立する．さらにこのモデル述語が例では「どのような入力値に対しても成立する」制限を付加する「for_any_input」により修飾されている．

4．変則構造の検出

本ツールの処理の概略は以下の通りである．まず，control_model部の各ノード単位に，条件を満たすプログラム部分を検索し，その部分が他のノードとの位置関係を満たしているかを調べる．条件の成否は，組み合わされたモデル述語の成否で判定する．その後で，control_model部のノードとは無関係に指定されている条件の判定，hierarchy_modelの構

```
br_cond(1)
extent(target) :
    if_sta{
        1(1) : exp
        2    : free
        3    : free
    }
br_cond(1)
extent(target) :
    while_sta{
        1(1) : exp
        2    : free
    }
br_cond(1)
extent(target) :
    repeat_sta{
        1    : free
        2    : free
        3(1): exp
    }
```

（ａ）モデル述語「br_cond」の定義

```
var(2)
extent(target,1) :
    exp {
        1:simp_expl {
            1:term {
                1:factorl {
                    1(2): free
                }
                2 : = 0
                3 : free
            }
            2 : = 0
            3 : free
        }
        2 : = 0
        3 : free
        4 : free
    }
```

（ｂ）モデル述語「var」の定義の一部

```
varref(1)
extent(target) :
    exp {
        include(1):var
    }
```

（ｃ）モデル述語「varref」の定義

図５　モデル述語の記述例

造を満たしているかの判定を行う．以下にモデル述語の成否の判定，記号実行，及び，実行結果について述べる．

(1)モデル述語の成否

図５に，モデル述語とそのモデル述語が成立する解析木の特定のパターンとの対応を記述した例を挙げる．解析木はツリーとして表わされ，各構文要素がノードとなる．そして，特定の構文要素がどのような構文要素を子のノードとして持つかを記述することにより，モデル述語と解析木のパターンとの対応を表わす．図５(a)では，モデル述語「br_cond」は，構文要素の一種である『if文』，『while文』，または『repeat文』において常に成立し，モデル述語の第１引数にその分岐の分岐条件式である『式』が返されることを示している．さらに，図５(b)は，モデル述語「var」の一部で，プログラム中で単一の『変数』が，『式』として使われている場合に成立する．そして，図５(c)は，モデル述語「varref」が，『式』に含まれる総ての『変数』に対して成立し，その『変数』を第１引数として返すことを示している．

また，モデル述語「relation」は，引数として，モデル変数を含む式を取り，必要に応じてターゲット・プログラムに対して，後述の記号実行を行い，その成否を判定する．

(2)記号実行

変数の値の動きや相互関係を調べるために部分的な記号実行を実行する．本ツールで用いる記号実行の能力は，次のようなものである．

- 分岐においては，すべての分岐パスを実行する．
- 不定回数の反復は０回および，１回の実行とする．
- 配列変数などで，添字の値が不定の状態で配列変数の値が必要になった場合には，その値を不定とする．

(3)実行結果

図４が表わす変則構造を図６のターゲット・プログラム例から検出した結果を図７に挙げる．また，ターゲット・プログラムの制御構造をグラフ化した制御構造グラフと，プログラムの対応を図８に示す．現在は，構造記述の各ノードに相当するプログラム部分を，制御構造グラフのノード番号を用いて出力する．また，モデル変数は，解析木の一部の形で出力される．制御構造グラフ，及び，解析木にプログラムとの対応情報を持たることにより，ソース・プログラムのままの表現で出力することも可能となる．

```
program test(input);
var     a,b,c : real;
        procedure sub( x,y : real );
                begin
                        if ( x < 0 ) then y :=  x
                                     else y := -x
                end;
begin
        a := 1;
        b := 2;
        if ( a > b ) then sub(a,c) else sub(b,c);
        if ( a < 0 ) then c := -c;
        c := c*2
end.
```

図6　ターゲット・プログラム例

```
=========================================================================
message :        'inappropriate branch structure'
-------------------------------------------------------------------------
*** hno          cno     path    location in gr         ***
    free_        cn1     [9]     3-3      at [3,6,3,1,3,7,2,3,1,2]
    free_        cn2     []      13-13    at [3,7,2,3,4,2]
*** model variable ***
    #a   : [exp,[simp_exp1,[term,[factor3,[exp,[simp_exp1,[term,[factor1,
           [var,4,0,[]]],0,[]],0,[]],1,[rel_op,small],[simp_exp1,[term,
           [factor2,[number,0]],0,[]],0,[]]]],0,[]],0,[]],0,[],[]]
    #b   : [exp,[simp_exp1,[term,[factor3,[exp,[simp_exp1,[term,[factor1,
           [var,1,0,[]]],0,[]],0,[]],1,[rel_op,small],[simp_exp1,[term,
           [factor2,[number,0]],0,[]],0,[]]]],0,[]],0,[]],0,[],[]]
-------------------------------------------------------------------------
```

図7　出力結果例

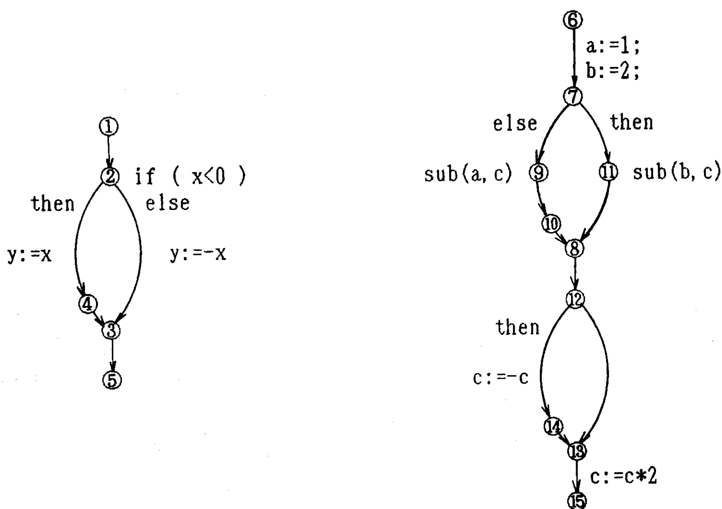

図8　プログラムの制御構造グラフとプログラムの対応

表1　目的別規約と記述可能性

規約の目的	規約数	記述可能な規約数	記述可能な規約数/規約数	記述不可能な規約の種類*				
				コメント	表記	機能	変数意味	その他
信頼性	8	7	.87	0	0	1	0	
簡潔性	9	6	.67	1	0	0	0	冗長性
完全性	8	5	.63	1	0	2	0	
無矛盾性	9	4	.44	1	1	0	3	
携帯性	10	4	.40	1	0	3	0	
構造性	12	4	.33	1	3	2	1	ｲﾝﾀｰﾌｪｰｽ規則有無
テスト容易性	15	4	.27	2	0	6	0	出力様式
理解性	21	3	.14	11	3	1	2	冗長性
保守性	16	2	.13	5	4	2	2	モジュール化
使用性	6	0	.00	0	0	3	2	出力様式
合計	114	39	.34	23	11	20	10	

* コメント：コメントの有無，コメントの内容に関する規約
表記　　：段付けなど表記上の規約
変数意味：変数名が変数の意味を忠実に表わすようにする，などの変数の意味に関する規約
機能　　：エラー処理，初期設定など，特定の機能に関する規約

5．評価

モデル記述の可能性，容易性，及び読解性の点からの評価を行った．

モデルの記述の可能性を確かめるために一般的なコーディング時の規約[5]を用いて，規約の目的別の記述可能性および，記述できない規約の種類を表1に挙げた．ここでは，本ツールが変則構造として扱うことを考えていない，コメントに関する規約や表記上の規約なども数に入れた．記述できない規約の中で目立つのは，特定の機能に関する規約である．ここでいう特定の機能とは，エラー処理，プリンタなどの周辺機器の初期設定など，抽象度の高い機能である．本ツールのモデル記述法では，これらの機能を表わすことが非常に難しい．特定の機能を構文レベルで記述しようという試み[6]もあるので，これを応用すればさらに記述性が高まる可能性がある．また，コメントに関する規約の中で，コメントの内容には無関係に，コメントの有無だけを問う規約に対しては，コメントが書かれている位置を解析木に反映させるような変更を加えれば，対応可能である．表1のコメントに関する規約のうち，コメントの有無のみに関する規約は6種（全体の3割弱）含まれている．

表2　使用したモデル述語の種類

使用した変則構造モデル数	予め用意されていたモデル述語	ユーザが追加したモデル述語	合計
0	47	-	47
1～2	7	8	15
3以上	14	0	14
合計	68	8	76

また，予め用意されているモデル述語の種類が十分なものであるかを調べるため，表1に挙げた規約を変則構造モデルとして記述する時に用いたモデル述語の種類を表2に挙げた．ユーザが新たに追加したモデル述語が，3種類以上の変則構造モデルで用いられることが無いことから，一般的なモデル述語としては，現在用意されているだけで十分であるといえる．

次に記述の容易性を確かめるために，一般のプログラマ・レベルの被験者が，指定する変則構造を変則構造モデルにより定義する作業を行った．この結

```
node1 : x := y+z;
node2 : u := x;
```

（ａ）プログラムの一部

```
node1 : assign(#a,#_)
node2 : assign(#_,#a)
```

（ｂ）誤ったモデル述語の記述

```
node1 : assign(#a,#_)
node2 : assign(#_,#b) and var(#b,#a)
```

（ｃ）正しいモデル述語の記述

図９　条件記述における誤りの例

果，次のようなことがわかった．

- 変則構造の意味から変則構造モデルのアウトラインを考える作業はほぼ確実にできる．
- 構造記述は問題無いが，条件記述で，構文解析に関する基本的な知識が必要であることが分った．

後者に関して，図９の例で述べる．図９(a) のように「ある変数に対する代入が行われた後，その変数を他の変数に代入する」ことを表わすためには，図９(b) のように記述する必要がある．これに対し，図９(c) のように記述してしまう例があった．解析木において，『代入文』の右辺はたとえ１つの『変数』のみであっても，『式』として扱われる．同様にモデル述語「assign」の第２引数も『式』を返すのに対し，『変数』が返ってくると勘違いするためである．

ここで，モデル述語「var」は，第１引数に『式』，または，『単純式』，『項』，『因子』を取り，それが実際には単一の『変数』のみである場合に成立して第２引数にその『変数』を返すモデル述語である．現在は，各モデル述語の引数に返される構文要素の種類を，同一モデル変数では統一しなくてはならないことを徹底し，『式』と『変数』のような間違えやすい例を例示することによって，対処している．

さらに，表題を除いた変則構造モデルから，定義されている変則構造の意味を読取る作業は，ほぼ９割程度の率で正解を得た．

結果として，ターゲット・プログラムのみから機能や意味を読取ることを試みていない現在の段階では，構文レベルの記述法は十分であり，一般のプログラマにとっても記述しやすい手法であるといえる．

６．おわりに

本ツールは，プロジェクト独自の規約に対する違反検出を自動化し，違反検出の作業に人間が拘束される時間の短縮，年限が違反検出した場合に起こりえる検出漏れの排除などを目的としているが，この他の応用例として，次のようなものが考えられる．

- プロジェクトに蓄積されたデバッグ上のノウハウを変則構造モデルで表現し，デバッガとして用いる．
- 移植において問題となる特定の構文のチェックなどの用途に利用する．
- プログラム教育において，問題が決っている時，問題を解くのに必要な機能の欠如のチェックをしたり，予想される誤りのチェックに用いる[7]．

また，今後の拡張としては，次のような方向が考えられる．

- 評価の節に述べた，ある程度高度な抽象度を持った機能の抽出手法を取り入れること．
- 毎回のプログラム解析の結果が以後の検出手法に反映されるような，なんらかの学習機能をつけること．

これらにより，より使いやすいツールへと発展していくと考えられる．

参考文献

［１］水野，東，"コンピュータソフトウェアの標準化"，日本経済新聞社 (1977).

［２］"SDEM : Software Development Engineer's Methodology", FACOMジャーナル, Vol. 3, No. 12, pp.25-31 (1977).

［３］白井，東，"事務システム標準化マニュアル"，日刊工業新聞社 (1986).

［４］玉井，福永，"記号実行システム"，情報処理，Vol.23, No.1, pp.18-28 (1982).

［５］宮本，"ソフトウェアエンジニアリング：現状と展望"，ＴＢＳ出版会 (1982).

［６］R.L.Sedlmeyer, "Knowledge-based Fault Localization in Debugging", Proc. of Software Engineering Symposium on High-Level Debugging, pp.25-31 (1983).

［７］W.L.Johnson, "PROUST:Knowledge_Based Program Understanding", IEEE, Trans.on Soft. Eng. Vol.SE-11, No.3, pp.267-275 (1985).