ソフトウェア工学　83－1
（１９９2.　2.　6）

ソフトウェア変更時の効率的なテストケース生成手法

古谷　信俊　　小野　康一　　深澤　良彰　　門倉　敏夫
早稲田大学　理工学部

変更を受けたソフトウェアに対し、変更された部分のみの効率的なテストをおこなうテストケースを生成するための一手法を提案する。一般にソフトウェアの変更では仕様の変更とプログラムの変更は独立して考えなければならない。われわれはテスト箇所を決定するために仕様とプログラムの両方の情報をつかい、それぞれの変更に対応している。テスト対象システムの仕様は形式的代数仕様記述で書かれていることを前提としている。テストケースは仕様の公理部から生成される。われわれの手法で得られたテストケースにより、全体的なテストを行うのに比べ、より少ないテストケースでより効果的なテストを行えることが確認できた。

An Efficient Test Case Genaration Method for Software Changing

Nobutoshi FURUYA, Koichi ONO, Yoshiaki FUKAZAWA, Toshio KADOKURA
School of Science and Engineering, Waseda University

We present a method for automatic test case generation to test a changed software effectively. With test cases generated by our method, only influenced parts by changing of the software are tested. In general, specification changing doesn't directly correspond to program changing. In order to cope with each changing, both specification and program infomations are utilized to decide where to be tested in the changed software. The former is assumed to be written in formal algebraic specification language. Test cases are generated from the axioms in the specification. By our method, effective testing is realized with far less test cases than wholly coverd ones.

1 序論

計算機システムに対するニーズは年々増大しており、また要求されるシステムも複雑化、大規模化の傾向にある。しかしこれに対して、ソフトウェア開発者の絶対数は不足してきている。これらの要因によってソフトウェアの開発費用はますます高価になり、これを減らすために既存のソフトウェアに変更を加えて再利用する場面が増えている。

ソフトウェアを変更する典型的な例として、ソフトウェア保守がある。一般的に一つのソフトウェアに対して、修正、適応、効率化といったさまざまな保守を施しながら利用していくことが多く、その過程ではさまざまな変更がおこなわれる。全コストの 60 % 以上が保守に費やされているという報告もある [1]。

ソフトウェア変更のもう一つの例は、ソフトウェア部品として使用することに伴う変更である。ソフトウェア部品は、再利用を目的として用意したプログラム単位であり、利用者は部品を単独あるいは適当に組み合わせて使用する。これらの部品の効果的な再利用のために、拡張、制限、引数化等の変更を加えることが多い。

これらの変更がおこなわれたソフトウェアに対しても、開発時同様テストがおこなわれる。しかし部分的な変更を受けたソフトウェアに対し、全体的なテストを繰り返すことは経済的でなく、変更により影響を受ける部分のみを効率的にテストすることが要求される。これによって変更に伴うコストの大幅な削減が期待できる。

本研究では、テスト対象システムにおいて変更の影響を受ける部分を仕様とプログラムの両情報より判断する。その後、仕様の情報から実際のテストケースの生成を行なう。この意味で、本研究の提案する手法はホワイトボックステストとブラックボックステストを融合したものであるということもできる。したがってシステムの仕様が正しいことが前提である。

これまで代数的仕様に基づくテストケース生成法はいくつか研究されている [2][3]。本研究で基盤とした Bouge の戦略 [3] では仕様の公理に着目し、公理の変数を具体化したテストケースが生成される。

本研究では仕様を CESP[4] に変換することを試み、仕様で定義される代数系を反映した処理を可能にした。またテストもれを防ぐために、いくつかの工夫をしている。

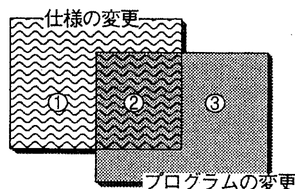
本稿では、まずソフトウェアの変更と、テスト対象システムの仕様記述について述べ、次に本システムの構成、およびテストケース生成法のアルゴリズムを示し、最後に生成されるテストケースの評価を行なう。

2 準備

2.1 ソフトウェアの変更

ソフトウェアの変更には様々なものがあるが、以下の三つに大きく分類することができる。

- **仕様のみの変更**
  読解性の向上のための変更など。
- **プログラムのみの変更**
  環境の変更に伴う修正、よりよいアルゴリズムへの変更など。
- **仕様とプログラムの変更**
  ソフトウェアの機能の追加や変更など。



①仕様だけが変更

②仕様とプログラムが変更

③プログラムだけが変更

図 1: 変更の種類

例えばプログラムのみ変更した場合、仕様だけからどの部分をテストすべきか判断することはできない。逆の場合も同様である。従ってさまざまな変更に対応するためには、ソフトウェア中のテストすべき箇所を特定する場合、仕様とプログラムの両方の情報が必要である。

2.2 構成的仕様記述法

本研究では仕様の機械的な操作を可能にするため、テスト対象システムは形式的代数仕様言語によって記述されているものとする。ここではその仕様記述法について述べる。

代数的仕様記述の中に、構成的仕様 [5] と呼ばれる記述法がある。構成的仕様記述では仕様の直接実行が可能で、プロトタイピングにも応用されている。構成的仕様の定義を簡単に以下に示す。

- 演算記号のうち標準形をつくるものを特に構成子と呼び、 他の演算記号と区別する。
- 公理の左辺は、その公理を含む代数系で定義する 非構成子で始まり、それ以外の項は構成子と 変数のみによって形成されている。
- 左辺に複数の同一変数が含まれない。
- 公理の右辺に含まれる変数は、必ず左辺にも登場する。
- 条件的公理 (a => b = c) はつかえないが、代わりに if 文 (b = if a then c else a endif) を使用できる。

- 非構成子を定義する公理の左辺が、 その定義域を厳密に一度カバーする。
- 公理を右向きの項書き換えシステムとしたとき、 変数を含まない任意の項の書き換えは必ず停止する。

しかし以上の制限の強さにより、その記述能力には限界がある。図2に集合の例を示す。

```
/* 整数集合 */
module int_set;
sort int_set;
use INT, BOOL;
ops
  cstr
    empty: -> int_set;
    add: int * int_set -> int_set;
  non_cstr
    member: int * int_set -> bool;
eqns
  add(X,add(X,Y)) = add(X,Y);              ...(*)
  add(X,add(Y,Z)) = add(Y,add(X,Z));       ...(*)

  member(X,empty) = false;
  member(X,add(Y,Z)) =
      if eql_int(X,Y)
        then true
        else member(X,Z)
      endif;
end;
```

図 2: 整数集合の代数的仕様

図2における (*) の公理は、構成的仕様では禁止されている構成子で始まっている。しかし、このような公理をつかわないで集合型を定義することはできない。

この問題を解決するために、構成的仕様の制限を若干ゆるめた半構成的仕様が提案されている [5]。しかし半構成的仕様では、仕様の停止性を満たすことが難しく、直接実行する場合にはかなり複雑な機構が必要である。

これに対し本研究では別のアプローチをとる。構成的仕様は始代数意味論をつかうことが前提になっていたが、終代数意味論による定義も可能とする仕様記述法を新たに提案する。始代数意味論と終代数意味論は、二つの抽象データの同値性を判断する方法が異なっている。

始代数意味論 : 公理から等価性を導けるときのみ同値

終代数意味論 : 公理から異なっていることが導けないかぎり同値

従って、終代数意味論を用いれば、(*) の公理は冗長な定義となり必要なくなる。

終代数意味論によってあるソートを定義をする場合、既に定義されているソートが必要である。従ってすべてを終代数意味論で定義することは不可能である。このため始代数・終代数をひとつのシステム定義で混在させる必要がある。本研究では以下のように、どちらの意味論をつかって仕様記述しているか明記させてこれに対処する。

```
module int_set: final;
        ⋮
```

3 テストケース生成法

3.1 テスト対象システム

本研究でテスト対象とするシステムでは、仕様とソースプログラムがともに揃っていることが必要である。

- 仕様
  2.2で示した、終代数意味論を含む構成的仕様で 記述されていることを前提とする。
- プログラム
  仕様中にあらわれる演算記号に相当する同名の関数や手続きが モジュールとして定義されていなければならない。 また、仕様で定義するソート (型) はプログラミング言語で用意 されているものか、ユーザ定義されているものとする。

3.2 システム概要

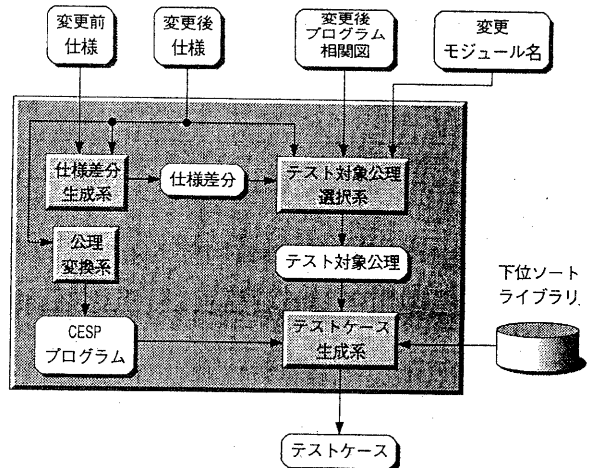


図 3: システム概要図

2.1で述べたようにテストすべき箇所を特定するためには仕様とプログラムの両方の変更情報が必要となる。

本システムでは仕様情報として、テスト対象システムの変更前後の仕様をつかう。

システムに入力されるプログラム情報は、変更後のプログラムにおけるモジュール相関図と、直接変更を受けたモジュール名の二つである。ただしこれらの情報はそれぞれ、変更前後のプログラムから自動的に生成および抽出される。

本システムは仕様差分生成系、公理変換系、テスト対象公理の選択系、テストケース生成系の四つの処理系から構成される。

仕様差分系は、テスト箇所を決定する際に利用する変更前後の仕様差分を取り出す。公理変換系では仕様をCESPのプログラムに変換する。変換したプログラムはテストケース生成系でつかわれる。テスト対象公理選択系は仕様中のテストすべき公理を選択する。これは本研究が公理ベースのテストケース生成法を採用するためである。テストケース生成系ではテストすべき公理中の変数へ、変数を含まない項を割り当てた形のテストケースを生成する。実際のテストは、テストケースの左辺と右辺の操作を独立に行い、それぞれの結果を比較することによって行う。図4にテストケースの例を示す。

```
(仕様)
  ops
  non_cstr
    half: posint -> posint;
          :
  eqns
    half(X) = div(X,2);
          :
(テストケース)
  half(0) = div(0,2);
  half(1) = div(1,2);
  half(32767) = div(32767,2);
          :
```

図 4: テストケース例

3.3 テスト対象公理選択系

(1) 仕様の情報による選択

本システムで用いる仕様は構成的であるため、公理は左辺の最左の演算記号の機能を定義するものと見なすことができる。従って、公理が変更されたということはその演算記号の定義が変更されたことを意味する。これより、変更された公理の左辺の最左の演算記号を含む公理が変更の影響を直接受けると考えることができる。また、こうして選ばれた公理によって定義されている演算記号を含む公理もテスト対象公理とする。これは仕様とプログラムの演算間の依存関係が必ずしも一致していないためである。

以上の操作を、新しくテスト対象公理としてつけ加えるものがなくなるまで繰り返す。

(2) プログラムの情報による選択

あるモジュールに手を加えた場合、それを呼び出しているモジュールにその変更の影響が波及すると考える。本システムでは、変更されたモジュールから相関図を上にたどって行き、仕様の演算記号と対応しているモジュールを全て選ぶ。最後に、それらのモジュールに対応している演算記号を含む公理をテスト対象公理とする。

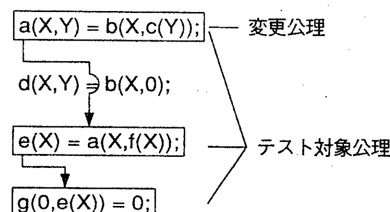
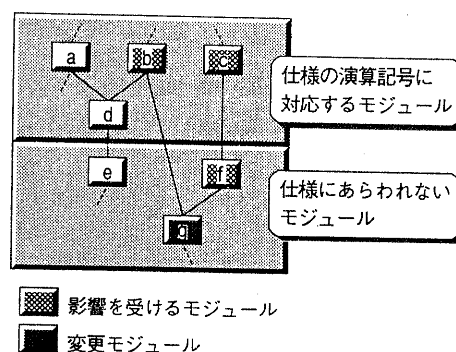


図 5: 仕様変更によるテスト箇所



a(X,Y) = d(X,1)+Y;

d(X,0) = 1;

c(X) = d(0,X);

b(X) = b(X-1);

テスト対象公理

図 6: プログラム変更によるテスト箇所

3.4 仕様変換系

(1) 代数的仕様から CESP への変換

Bougeは代数的仕様を論理型言語に変換して、公理中の条件部を満たす項を探すためにつかった。Bougeは論理型言語にPrologやMetalogを用いたが、本研究ではこの変換法をCESPに拡張する。

オブジェクト指向の概念を取り入れたCESPでは、オブジェクトの状態を保持するスロットと呼ばれる変数を個々のオブジェクトが持つことができる[4]。従って、状態遷移モデルを基盤とする前提 - 終了条件記法などの仕様記述法からのCESPへの実現が一般的に自然であると思われる。

本手法では代数的仕様をCESPに変換することを試みた。CESPのクラスは本来オブジェクト、つまり抽象データであるが、ここでは仕様の定義する代数系に対応させる。Prologでは複数の代数系から構成される仕様をプロ

グラムに反映させることは困難であったが、自然な形でそれを行うことができる。

クラスを代数系に対応させた場合、データが属するソート(型)の情報を反映させることは難しい。そのためソートに関する処理を一括して行うクラス sort をあらかじめ定義しておき、各クラスの演算記号の定義の最後で、クラス sort を通してソートのチェックを行わせるようにした。クラス sort は仕様毎に異なった演算記号のデータベースを持たなくてはならないが、これは事前に他のツールより生成される。以上より多重定義された演算記号も容易に扱える。ただし演算記号を呼び出す毎にソートをチェックするため処理の効率は現在のところ期待できない。

CESP への変換規則を示す。実際の変換系は C と CESP のマクロをつかって構築している。

<仕様の use 節の変換>

```
(仕様) module mod0;
       use mod1,...,modn;

(CESP) local
         find_st(A,B) :-
         :find(#sort,[mod0,mod1,...,modn],A,B);
```

<演算定義部の変換>

```
(仕様) non_cstr
         p: x1*...*xn -> y;

(CESP) クラス述語定義部
       :p(O,X1,...,Xn,Y) :-
         p(X1,...,Xn,Y),
         :sort(#sort,X1,x1),...,
         :sort(#sort,Xn,xn),
         :sort(#sort,Y,y);
```

<if 文の変換>

```
(仕様) a(x) = if t(x)
                then b(x)
                else c(x)
              endif;

(CESP) a(x) = b(x) :- t(x,bl_true), !;
       a(x) = c(x) :- !;
```

<等式 - ホーン節変換>

<規則 1>

```
... g(x1,...,xn) = y ... <=>
... g(x1,...,xn,y) ...
```

<規則 2>

```
p(g(x1,...,xn),z2,...,zm) :- ... <=>
p(y,z2,...,zm) :- g(x1,...,xn) = y ...
```

<規則 3>

```
... :- p(g(x1,...,xn),z2,...,zm) ... <=>
... :- g(x1,...,xn) = y, p(y,z2,...,zm) ...
```

<演算記号の変換>

```
... :- ... p(x1,...,xn) ... <=>
... :- ... find_st(p,O), :p(O,x1,...,xn) ...
```

図 7に先の集合型の仕様を変換した例の一部を示す。

```
/* 整数集合 */
class int_set has
  :member(O,A,B,C) :- member(A,B,C),
      :sort(#sort,A,int),
      :sort(#sort,B,int_set),
      :sort(#sort,C,bool);
local
  find_st(A,B) :-
   :find(#sort,[int_set, int, bool],A,B);
  member(X,empty,bl_flase);
  member(X,add(Y,Z),bl_true) :-
      :find_st(eql,O),
      :eql(O,X,Y,bl_true), !;
  member(X,add(Y,Z),A) :-
      :find_st(member,O);
      :member(O,X,Z,A), !;
end.
```

図 7: 整数集合の CESP プログラム

(2) ユーザ定義の等価関数

等価関数も他の演算記号と同様に仕様の上で定義する必要がある。ここで文字型の等価関数を考える。帰納的推論が許される仕様では、以下の定義で十分であった。

```
eql(X,X) = true;
eql(X,Y) = eql(Y,X);
imply(and(eql(X,Y),eql(Y,Z)),eql(X,Z)) = true;
```

しかし帰納的推論が許されていない構成的仕様では、次のように膨大な数の公理が必要である。

```
eql(a,a) = true;
eql(a,b) = false;
eql(a,c) = false;
     :
```

これは、列挙型のようなソートを定義する場合は必至である。必要な公理が高々有限個で済むことは保証されているが、定数演算記号の数に対し指数的に増えるので実用的でない。したがって、等価関数 eql を定義する公理がないソートでは次のプログラムが用意されるという暗黙のルールをおいて対処する。これは始代数意味論より得られる合同関係に等しい。

```
(仕様)
  ops
  non_cstr
    eql: s * s -> bool;
```

```
        :
(CESP)
  class ... has
    :eql(0,A,B,C) :- eql(A,B,C),
         :sort(#sort,A,s),
         :sort(#sort,B,s),
         :sort(#sort,C,bool);
        :
  local
    eql(A,A,true):-!;
    eql(_,_,false);
        :
```

3.5 テストケース生成系

テスト対象として選択された公理について、テストケースを生成する。テストケース生成のアルゴリズムは対象とする公理の形から二つに分かれる。

3.5.1 if 文を含まない公理のテストケース

t(x1,...,xm) = t'(x1,...,xm) の形をした公理が対象の場合のアルゴリズムを以下に示す。

- 公理中の変数のうち その公理を含む代数系で定義しているソートのものを、最大ネスト数内で演算記号を組み合わせてできる 全ての項をつかって置き換える。
- 残った下位ソートの変数を、そのソートの適当な定数項で 置き換える。

本システムでは、より効果的なテストケースを生成するために、以下のような工夫を行なっている。

(1) 下位ソートのテストケース

Bouge の戦略では、if 条件部に関与しない下位ソート変数へ割り当てる値をランダムに選択している。本システムでは下位ソートライブラリを使って、あらかじめ用意された各ソートの境界値を変数に割り当てる。扱うデータの境界値付近でエラーの発生する頻度が高いため、ランダムな値の割当てに較べてより有効なテストが期待できる。

表 1: 下位ソートの境界値

ソート	境界値
int	-32768,-1,0,1,32767
posint	0,1,32767
char	a,Z
string	a,ZZZZZZZZZZ

(2) 両辺がブール型を返す公理について

以下の公理に対して、本システムでは $(X) = \{0, 32767\}$ というテストケースを生成する。

```
f(X) = div(X,2) ;
```

また、次の公理を上と同様な等式的公理として扱うと、同じテストケースが生成される。

```
f(X) = eql(X,1) ;
```

この場合、公理の両辺が TRUE になるテストケースが含まれていないため不十分である。従って、両辺が TRUE、FALSE 以外のブール型の演算記号から形成されている場合は、以下のような等価な if 文に変換し 3.5.2で述べるアルゴリズムにより TRUE、FALSE の両方の結果が得られるテストケースを生成する。

```
f(X) = if eql(X,1)
          then true
          else false
       endif ;
```

3.5.2 if 文を含む公理のテストケース

t(x) = if a(x) then t1(x) else t2(x) endif の形をした公理が対象の場合のアルゴリズムを以下に示す。

- CESP によって、a(x) が true を返す x の項 i(x) および、a(x) が false を返す x の項 j(x) を 最大ネスト数の範囲で探す。
- t(i(x)) = t1(i(x)) と t(j(x)) = t2(jx)) に if 文を含まない公理の場合のアルゴリズムを 使ってテストケースを生成する。

本システムでは、この過程において以下の工夫を行ない、テストの効果を上げている。

(1) 組合せ条件部のテストケース

次の公理は、ボールゲームのスコア管理の仕様の一部であり、15 点先取のルールをデュース制に変更したことを示している。

```
iswin(red,score(R,W)) =
    if eql(R,15)
        then true
        else false
    endif
        ⇓
iswin(red,score(R,W)) =
    if R >= 15 and R >= W+2
        then true
        else false
    endif
```

Bouge の方法では、if 文による分岐の両側を網羅するようにテストケースを生成する。従って、変更後の公理に対して生成されたテストケースにおいて、変数 R、W への値の割当ては、例えば $(R,W) = \{(15,0),(0,0)\}$ となる。しかし、これは変更前の公理に対するテストケースと同じであり、デュース制によって変更された部分のテストにはなっていない。これより本システムでは、条件部が複数の条件の複合形である場合には、各条件の結果の組合せを全て網羅するようにして、テストケースの範囲を広げた。これは、ホワイトボックステストの手法で、

分岐網羅から、条件網羅に変更したことに相当する。上の例では、本システムの生成するテストケースは、$(R,W) = \{(15,0),(15,13),(2,0),(0,0)\}$ という割当てを含む。ただし、条件の組合せによっては解が得られない場合がある。このため、演算記号の最大のネスト数をあらかじめ与えておき、その範囲内で解が見つからない場合は解なしとして計算を停止させる。

本手法では、条件部の細かい条件の数が増えるにつれ組合せの数が指数的に増加する。しかし、記述の経験から、抽象度の高い仕様において条件部がそれほど複雑になることは少なく、たとえテストケースの数が多少増えてもテストケース生成からテスト判定のプロセスまでを自動化することにより十分対処できる。

(2) 比較演算のテストケース

これまでの戦略では、条件公理に対して、一つの条件に対して一つテストケースを生成していた。従って次のようなバグを見落してしまう。

今、引数が1より大きいかどうかを調べる演算記号 ge_one を考える。

```
ge_one(X) = (X >= 1);
```

この演算 ge_one のインプリメントを以下のように誤ってしまったとする。

```
bool ge_one(int x)
{
  if (x == 1)
    return true;
  else
    return false;
}
```

公理(1)に対して、両辺が true、false の場合を一つずつ、以下のテストケースが生成される。

```
ge_one(1) = (1 >= 1);
ge_one(0) = (0 >= 1);
```

しかし、このテストケースを上記の誤ったインプリメント上で試しても正しい結果が得られ、バグの存在を確認できない。

この問題を解決するために、比較演算記号 >=、<=、>、<= に関する条件部に対しては、複数のテストケースを生成するというルールを設ける。具体的にいくつ生成するかは、実験を重ね経験的に定義するつもりであるが、現在のところ二つで十分であると思われる。

4 評価

プログラム中でテストすべき箇所を特定し、それらの箇所について生成テストケースが満たす分岐カバレージを調べ、テストケースの有効性を示す。テストすべき箇所はプログラム中の直接変更を加えた部分とした。

表2に本システムが生成したテストケースと、Bouge のオリジナルの方法によって生成したテストケースによるカバレージをそれぞれ示す。表2より、組合せ条件部に

表 2: 各テストケースによるカバレージ

システム	Bouge	本システム
ボールゲームスコア	11/13 (85%)	13/13 (100%)
エディタ	7/7 (100%)	7/7 (100%)
図書館データベース	17/18 (94%)	18/18 (100%)

対する処理の改良による効果は明らかである。また、上記の基準で選んだ箇所に関して本システムでは100%の分岐カバレージを満たしていることも評価できる。

次に、変更後のソフトウェア全体に対するテストケースを生成した場合に比べ、テストケースの数をどの程度削減することができたかを調べる。表3は各テスト対象システムの変更状況であり、表中の変更率とは、全体のステートメント数に対する、変更されたステートメント数の割合を意味している。

表 3: テストシステムの変更状況

システム例	仕様変更率	プログラム変更率
ボールゲームスコア	19/42 (45.2%)	21/224 (9.4%)
エディタ	9/53 (17.0%)	8/311 (2.6%)
図書館データベース	29/309 (9.4%)	81/1869 (4.3%)

テストケース数の減少率を表4に示す。ただし減少率とは、変更後のソフトウェア全体を対象として生成したテストケースの数に対する、本システムが生成したテストケースの数の割合を意味している。ボールゲーム得点システムの例では小規模なシステムに対して変更箇所が大きかったためテストケース数はそれほど減少しなかったが、残りの二つの例ではかなりの減少が確認できた。

表 4: テストケース数の減少率

システム例	減少率
ボールゲームスコア	16/18 (89%)
エディタ	91/247 (37%)
図書館データベース	60/248 (24%)

最後にバグの種類別にテストケースの性能評価を行う。ボールゲーム得点システムとエディタを例に、システムの一部を変更し、その過程でさまざまなバグを埋め込み、本研究のテストケースでどの程度発見できるか調べた。

符号、等号、不等号のエラーは、演算記号のテストケース生成法を改良したことにより0にすることができた。

表 5: 分類別のエラー発見数

エラーの種類	未発見 エラー数 / 埋め込み エラー数	
機能抜け・未変更	1 /	3
演算エラー	0 /	6
制御構造エラー	0 /	3
データ初期化抜け	0 /	1
データ出力エラー	0 /	2
データ範囲の未カバー	0 /	2
符号、等号、不等号のエラー	2 → 0 /	12
その他	1 /	1

その他のエラーで発見できなかったものはプログラムの書式が標準と異なっているというバグで、機能的には問題がないものである。このように、論理的な影響を与えないバグは本システムのテストケースでは対象としていない。

機能抜け、未変更の未発見エラーは、仕様とプログラムの構造が異なっていたために発見することができなかったものである。ボールゲーム得点システムを例に、この種類の未発見エラーについて考察する。

いま、勝敗の判定を以下のようにジュース制によって行っているとする。

```
red_win(R,W) = (R >= 15) and (R >= W+2);
white_win(R,W) = (W >= 15) and (W >= R+2);
gameset(R,W) = red_win(R,W) or white_win(R,W);
```

これに対し、勝敗の判定を 15 点先取と誤って、演算 gameset がインプリメントされたとする。

```
Bool gameset(r,w)
int r, w;
{
  if ((r == 15) || (w == 15))
    return true;
  else
    return false;
}
```

gameset を定義する三番目の公理に対し、テストケースを生成すると、

```
gameset(15,0)=red_win(15,0) or white_win(15,0);
gameset(0,15)=red_win(0,15) or white_win(0,15);
gameset(0,0)=red_win(0,0) or white_win(0,0);
```

となる。明らかにこのテストケースでは、バグを発見できない。

このようなバグを見落してしまうのは、仕様の演算記号間の呼び出し関係と、プログラムのモジュールの呼び出し関係が一致していないためである。従って gameset をテストする場合、gameset を通して red_win、white_win のテストを再度しなければならない。しかし階層的仕様記述によって、呼び出しの階層が深くなることは必至で、それに伴ってテストケースの数は指数的に増大してしまい、実用的でない。

モジュール階層図から自動判断させる方法も考えられるが、複雑に絡みあったモジュール階層図と演算の呼び出し関係の対応をとることは非常に困難である。

テストケースの総数とテストの質のトレードオフを考えれば、これまでの方法が妥当であると思われる。以上より、このようなバグを防ぐためには、できるだけ仕様の演算記号の呼び出し関係に忠実な実現を奨励することが唯一の手段であると考えている。

5 結び

変更が行われたソフトウェアに対し、良質かつ無駄の少ないテストケースを生成する手法を与えた。

本研究では、仕様から変換されたプログラムが正しく動作するように構成的仕様を採用した。これに対し、制約論理プログラミング言語を用いることによって、より一般的な代数的仕様への応用が期待できる [3] [6]。

また、対象システムをオブジェクト指向プログラムに限定することによって、テスト範囲をより厳密に決定することも検討中である。

現在 Sun-4 の CESP シェル上で、CESP と C をつかってインプリメントを行っている。

今後は、テストの自動実行系、テスト結果の自動判定系も含めた統合的なテスト環境を構築していく予定である。

参考文献

[1] G. Parikh. ソフトウェア・メンテナンス・ハンドブック. 啓学出版, 1989.

[2] P. Jalote and M. G. Caballero. Automated testcase generation for data abstraction. In *Proc. COMPSAC 88. Chicago*, IL, Oct 1988.

[3] L.Bouge, N.Choquet, L.Fribourg, and M.-C.Gaudel. Test sets generation from algebraic specifications using logic programming. *J.System and Software*, 2:343–360, 1986.

[4] *CESP* 言語 *2.0* 版. (株)AI 言語研究所, 1990.

[5] I. V. Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering.* Spring-Verlag, 1989.

[6] R. Denney and S. W. Services. Test-case generation from prolog-based specifications. *IEEE SOFTWARE*, pages 49–57, March 1991.