ソフトウェア工学 58－3
（1988. 2. 4）

形式的仕様記述からのテストデータ生成

所　洋一　門倉敏夫　深沢良彰
早稲田大学理工学部

ソフトウェアの生産性・信頼性の向上のために、仕様記述を行なうことが重要視されてきている。しかし、その仕様記述から人手でプログラムを作成した場合には、仕様記述とプログラムとの間に、差異が生じる可能性がある。したがって、本システムは、形式的仕様記述から作成されたプログラムに対するテストデータを仕様記述を基に生成し、このテストデータを用いて、仕様記述とプログラムとの間の等価性を検証することができる。

本システムのテストデータ生成は、基本的に、プログラムの入力に対する同値分割、限界値分析の手法を用いている。さらに、テストデータ生成時に具体的な情報を得るためにテストデータ・ライブラリを用いている。

"Test Data Generation Based on a Formal Specification"

Youichi TOKORO, Toshio KADOKURA, Yoshiaki FUKAZAWA

School of Science and Engineering, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 160, Japan

The specification is regarded to be important for the improvement of the software productivity and reliability. In case of programming by hand from a given specification, however, there is a possibility to occur an inconsistency between the specification and the program. Based on a formal specification, this system generates a test data for the program which is written followed by the specification. This test data is utilized to detect the inconsistency. The test data is basically generated by equivalence partitioning and boundary-value analysis to the program input. Moreover, it uses the test data library to get concrete information on test data generation.

1．はじめに

ソフトウェアに関して、高い信頼性が得られない、開発予算がオーバーする、開発スケジュールが遅れるといった問題が、無視できなくなってきている。そこで、ソフトウェアの生産性・信頼性向上のために、仕様記述を行い、これを基にプログラムを作成することが重要視されてきている。

仕様記述からプログラムを作成するための手法としては、コンピュータによる自動生成とプログラマによる作成が考えられる。

自動生成は、理想的な手法であるが、ＡＩ的な生成手法等のアプローチの完全な実現可能性、生成されたプログラムの実行効率等を考えると、現時点では、プログラマが介入せざるを得ないと考えられる。

一方、プログラマによる作成においては、仕様記述と作成したプログラムとの間に差異が生じる可能性がある。そこで、何らかの方法により、作成されたプログラムの仕様記述に対する等価性を検証することが必要になる[4]。

等価性を検証するための一手法として、テストデータを用いたソフトウェアのテストがある。テストデータ生成のためには、その生成に要する時間、コスト、エラーの検出効率等を考慮しなければならない。そのなかでも、最も重要視すべき項目は、エラーの検出効率である。

ランダムにテストデータを生成する手法は、最も検出効率の低い手法である。そこで、テストデータ生成に対して、何らかの指針を持たなければならない。その指針として、ブラックボックス・テストとホワイトボックス・テストがある。ブラックボックス・テストに基づくテストデータ設計法には、同値分割、限界値分析などがある。ホワイトボックス・テストとしては、命令網羅、判定条件網羅、条件網羅などに基づくテストデータ設計法がある。一般には、エラーを発見する確率を向上させるために、ブラックボックス・テスト的手法を用いて、補足的にホワイトボックス・テスト的手法を用いることが考えられている[1][2]。

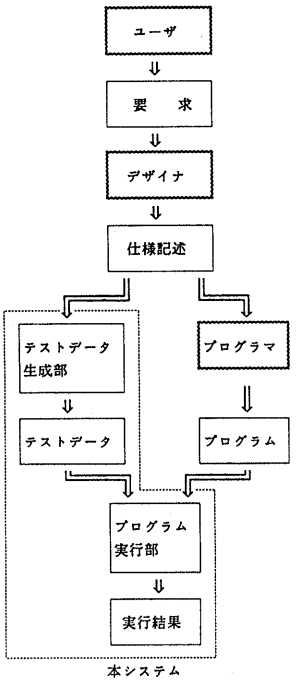
2．本システムの概要

本システムの目的は、プログラマにより作成されたプログラムが、デザイナが記述した仕様記述を満足していることを検証することである。この検証のために、仕様記述を基にテストデータを生成する。そして、そのテストデータを用いて、作成されたプログラムを実行し、その実行結果を判断することによって検証を行う。

本システムにおいて、要求からプログラムを作成する過程をFig.1 に示す。Fig.1 において、要求はユーザから提出される文書で、通常は、自然言語によって記述される。仕様記述は、デザイナが要求から仕様記述言語を用いて作成する。プログラムは、プログラマがプログラミング言語を用いて記述する。

本システムにおける仕様記述言語は、前提終了条件形式(pre- and post-conditions)を用い、抽象データタイプを基礎としている[5][6]。さらに、入力に対する詳細な条件を記述する機能が加えられている。

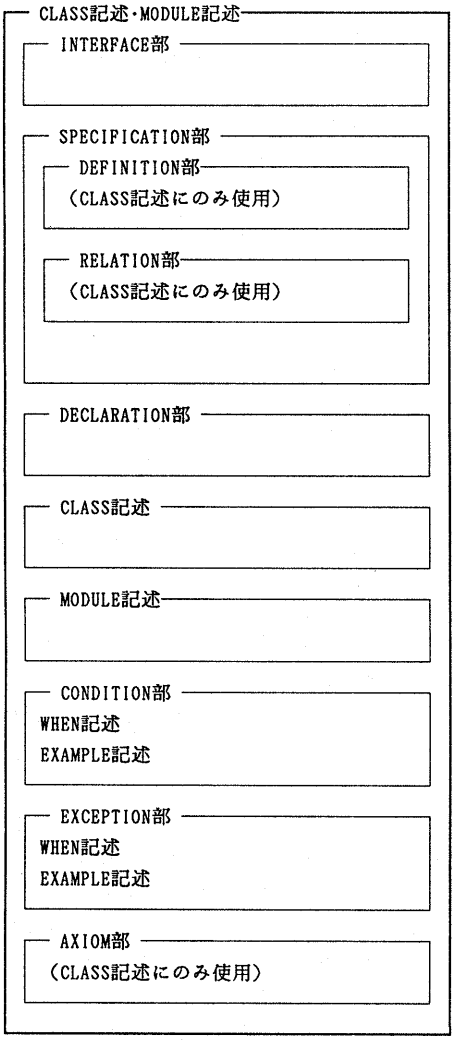
オペレーションの対象となるデータのデータタイプは、ブラックボックス・テストのためのテストデータを生成するために使用される。しかし、これらのみでは明らかに不十分である。そこで、ホワイトボックス・テストのためのテストデータを、入力に対する条件記述から生成する。これにより、本システムにおいては、ブラックボックス・テストとホワイトボックス・テストを融合したテストを行うためのテストデータを生成できる。



Ｆｉｇ．１　システム構成

3．仕様記述言語

本仕様記述言語は、抽象データタイプの定義を行うCLASS記述と、オペレーションの定義を行うMODULE記述から構成される。CLASS記述、MODULE記述内に、階層的にCLASS記述、MODULE記述を行うこともできる。Fig.2は本仕様記述言語の構成の概念図である。



Ｆｉｇ．２　仕様記述言語概念図

1)INTERFACE部は、外部とのインターフェースの宣言である。その宣言には、IMPORT宣言、EXPORT宣言、PARAMETER宣言の３種類がある。

・IMPORT宣言　他の仕様記述で定義されている概念を使用することを宣言する。

・EXPORT宣言　この仕様記述で定義している概念を他の仕様記述で使用できることを宣言する。

・PARAMETER宣言　抽象データタイプを定義した場合に、その要素のデータタイプをパラメータとして渡すための宣言である。この宣言はCLASS記述にのみ存在する。

2)SPECIFICATION部は、MODULE記述では、前提終了条件形式を用いた要求の記述であり、CLASS記述では、DEFINITION部、RELATION部から構成されている。

・DEFINITION部　抽象データタイプの定義において、その内部で定義された各オペレーションのオペランドのデータタイプの宣言をを行う。

・RELATION部　内部で定義された各オペレーション間の関係を等式を用いて定義する。すべてのオペレーションは、少なくとも一度、RELATION部において定義されなければならない。

3)DECLARATION部では、CLASS記述、MODULE記述の内部で使用している変数のうち、INTERFACE部で宣言されていない変数の宣言を行う。また、CLASS記述において、抽象データタイプの実現方法の宣言を行う。

4)CONDITION部では、オペレーションの対象となるデータに対して、処理が分岐する場合の条件式を集合の記法を用いて表現する。この記述には、WHEN記述、EXAMPLE記述の２種類がある。

・WHEN記述　条件式を完全な数学的関係式で表現できる場合には、これを用いる。

・EXAMPLE記述　条件式を完全な数学的関係式を用いて表現できない場合には、例を用いて表現する。例えば、あるオペレーションをテストする際に、変数'st'には、すべてが数字で構成されている文字列と、それ以外の文字列を用いなければならないという場合には、

```
st = {all digit, @otherwise};
```

等と記述することができる。この場合は、人間がこの記述を参照して、テストデータを作成するため、詳細な記述法は定義していない。

5)EXCEPTION部は、誤りとなる入力を表わす条件式の記述である。この条件式も、CONDITION部と同様に、WHEN記述、EXAMPLE記述の２種類があり、集合の形式を用いて表現する。

6)AXIOM部では、定義している抽象データタイプが必ず満たさなければならない条件を記述する。その際には、このデータタイプを構成している基本データタイプで表現する。

```
string : CLASS
  INTERFACE
    null, left, right, length, eval, substr,
    ins, del : EXPORT(OPERATION);
  ENDINTERFACE
  SPECIFICATION
    DEFINITION
      null   :                          -> *;
      substr : *, interger, integer     -> *;
               ⋮
      del    : *, integer               -> *;
    ENDDEFINITION
    RELATION
      length(null) = 0;
              ⋮
      length(del(st, ps)) = length(st) - 1;
    ENDRELATION
  ENDSPECIFICATION
  DECLARATION
    string : array[positive_integer] of
             character;
           ⋮
    ch : character;
  ENDDECLARATION
  AXIOM
    ALL i, j ^ string[i] == EOS ^
    string[j] == EOS -> i == j;
  ENDAXIOM

  null : MODULE
         ⋮
  ENDMODULE null
         ⋮
ENDCLASS string
```

Ｆｉｇ．３　ＣＬＡＳＳ記述例

CLASS記述の例として、文字列というデータタイプの記述をFig.3に、MODULE記述の例[7]として、手続きの記述をFig.4に示す。Fig.3において、SPECIFICATION部の中のDEFINITION部で用いられている'*'は、定義している抽象データタイプ自身を示す。

```
left : MODULE
  INTERFACE
    string : IMPORT(READ/WRITE);
    pos : IMPORT(READ) integer;
  ENDINTERFACE
  SPECIFICATION
    ALL i ^ i <= pos ->
        string'[i] = string[i]
        string'[pos+1] = EOS;
  ENDSPECIFICATION
  DECLARATION
    i : integer;
  ENDDECLARATION
  CONDITION
    WHEN pos {=<, >} length(string);
  ENDCONDITION
ENDMODULE left
```

Ｆｉｇ．４　ＭＯＤＵＬＥ記述例

４．テストデータ生成

テストデータ生成は、主として、CLASS記述、MODULE記述から境界値分析に関する情報を、テストデータ・ライブラリから同値分割に関する情報を得ることにより行う。これにより、網羅率を向上させることができる。

さらに、本システムは、テストデータ生成時に必要となるデータタイプの情報を得るために、テストデータ・ライブラリを使用する。テストデータ・ライブラリには、データタイプの型名、その構成、それを代表するテストデータ、各データタイプの隣接値の求め方の情報が含まれている。

データとして扱う領域が、無限集合、または、無限に近い集合の場合には、人手の介入が必要となる。この場合には、ある条件を満たさないという場合に、その値をシステムが自動的に決定できないからである。

以下に、テストデータ生成のアルゴリズムを示す。CLASS記述とMODULE記述に対しては、異なるアルゴリズムによりテストデータを生成する。

4．1　CLASS記述

1)DECLARATION部の記述より、CLASS-TYPE(そのCLASS自身で定義している抽象データタイプ)と、そのCLASS-TYPEを構成しているデータタイプを、テストデータ・ライブラリに登録する。

2)CLASS記述の内部でオペレーションを定義している各MODULE記述に対して、4.2節に示すMODULE記述に対するテストデータ生成のアルゴリズムを使用して、CLASS-TYPEのテストデータを生成する。

3)AXIOM部の記述により、生成されたテストデータの定義している抽象データタイプに対する妥当性を判断する。

4)各オペレーションにおいて、CLASS-TYPEのデータ以外の入力が存在する場合は、2)と同様にして、4.2節に示すMODULE記述に対するテストデータ生成のアルゴリズムにより、テストデータを生成する。

5)4)において生成されたテストデータのデータタイプがすでにテストデータ・ライブラリに登録されている場合には、生成されたテストデータに、テストデータ・ライブラリから得られるテストデータを加える。

6)生成されたテストデータの隣接値が計算できる場合には、さらに、それぞれの隣接値をテストデータに加える。

7)生成されたテストデータを組みあわせて、テストデータの集合を生成する。

8)生成されたテストデータの集合を用いて、各オペレーションを実行する。そして、その実行結果をRELATION部の等式に適用する。その等式の成立、不成立により、エラーの有無、及び、エラーの存在するオペレーションの判断を行う。

9)生成されたCLASS-TYPEのテストデータの中から、定義している抽象データタイプを代表するテストデータを選択して、テストデータ・ライブラリに登録する。

CLASS記述においては、テストデータ生成時の一部を除いて、ほぼ自動的に、テストを行い、結果を判断することができる。

4．2　MODULE記述

CONDITION部、EXCEPTION部のどちらに対しても、基本的に、同じアルゴリズムでテストデータの生成を行う。

1)WHEN記述、EXAMPLE記述に記述されている、変数のデータタイプを、そのMODULE記述のINTERFACE部、DECLARATION部により決定する

2)決定されたデータタイプを基に、テストデータ・ライブラリを参照して、そのデータタイプの変数に対応するテストデータを求める。

3.1)WHEN記述の場合、求められたテストデータに、WHEN記述から得られるデータをテストデータとして加える。

3.2)EXAMPLE記述の場合、求められてテストデータに、EXAMPLE記述を参照して人手で決定したデータをテストデータとして加える。

4)WHEN記述の条件式において、テストデータを生成した変数の比較の対象となっている変数の値を決定できる場合には、その値を計算して、その対象となっている変数のテストデータに加える。

5)テストデータ・ライブラリを参照して、隣接値が得られるテストデータに対しては、それぞれの隣接値もテストデータとして加える。

6)生成されたテストデータを組みあわせて、テストデータの集合を生成する。

7)生成したテストデータを用いて、人手で作成したプログラムのテストを行う。

MODULE記述においては、テストデータを用いたテスト、結果の判断は、人手で行わなければならない。

Fig.4 の仕様記述から生成されたテストデータをFig.5 に示す。この時に使用したテストデータ・ライブラリの一例をFig.6 に示す。

```
string.left =
  {[string = null, pos = -1],
   [string = null, pos = 0],
   [string = null, pos = 1],
   [string = null, pos = 100000],
   [string = "AAAAAAAAAA", pos = -1],
   [string = "AAAAAAAAAA", pos = 0],
   [string = "AAAAAAAAAA", pos = 9],
   [string = "AAAAAAAAAA", pos = 10],
   [string = "AAAAAAAAAA", pos = 11],
   [string = "AAAAAAAAAA", pos = 100000]
```

Ｆｉｇ．５　テストデータ例

型名	構　成	テストデータ	隣接値
integer	primitive	…, -1, 0, 1, …	+1, -1
array	primitive	max(subscript) = 10	------
character	primitive	A	------
string	array[positive_integer] of character	null	------
⋮	⋮	⋮	⋮

Ｆｉｇ．６　テストデータ・ライブラリ

５．本システムの評価

５．１　仕様記述言語の記述性

ここでは、記述性の評価として、ある機能に対して記述した仕様記述とプログラム（Ｃ言語により記述）とのサイズの比較を行った。ここでは、その基準として行数、文字数を評価した。その結果の一部をFig.7に示す。それぞれを合計して、比率を計算すると、

行数

プログラム：仕様記述＝１：１．０４

文字数

プログラム：仕様記述＝１：１．８７

となる。

この結果、多くの場合、仕様記述がプログラムよりも、行数、文字数に関して、多少大きくなっていることがわかる。しかし、これは、仕様記述におけるキーワードの存在にかなりの影響を受けていると考えられる。そこで、キーワードを除いて、文字数に関して、プログラムと仕様記述の比率を計算すると、

プログラム：仕様記述＝１：０．８７

となる。

以上の定量的評価により、本仕様記述言語による仕様記述と、プログラムには、ほとんど差がないという結果が得られる。

機能名	仕様記述	コード
alldig	17 566	13 305
balpar	14 497	22 336
bubble	12 355	16 245
char_count	10 331	8 57
compare	14 491	11 147
left	15 361	6 56
length	9 264	8 83
linecount	11 390	11 134
pop	13 434	11 189
push	13 423	13 198
right	15 372	11 133

※上段：行数、下段：文字数

Ｆｉｇ．７　サイズの比較

5．2　生成されるテストデータの評価

まず、ソフトウェアにおける実行時エラーを次の3種類に分類する[2]。

・サブケース・エラー：仕様に対応したプログラムの処理パスがない。

・ドメイン・エラー：対応する処理パスは存在するが条件判定が不正で制御が渡らない。

・コンピュテーション・エラー：パス内の演算が間違っている。

これらのエラーの具体的な例をFig.8に示す。

以上の様に分類されたエラーが、本システムによって生成されたテストデータによって検出できるかをカウントすることによって、テストデータの評価を行う。さらに、この評価をテストデータ生成時に、人手が介入しない場合、介入した場合に分ける。この場合、介入とは、仕様記述のCONDITION部、または、EXCEPTION部のEXAMPLE記述を参照してテストデータ生成を行っているかどうかである。

実際にこの方法で評価した結果をFig.9, Fig.10に示す。

この評価の対象としたのは、C言語で記述された文字列に関する各種のプログラムである。それらのプログラムに対して、3種類のエラーを、それぞれに100件想定して評価を行った。ただし、それぞれのプログラムには、外部変数の参照は存在しないという前提を置いている。

エラーの種類	検出可能	検出不可能
ｻﾌﾞｹｰｽ・ｴﾗｰ	78	22
ﾄﾞﾒｲﾝ・ｴﾗｰ	64	36
ｺﾝﾋﾟｭﾃｰｼｮﾝ・ｴﾗｰ	81	19

Fig. 9　検出結果（人手無）

エラーの種類	検出可能	検出不可能
ｻﾌﾞｹｰｽ・ｴﾗｰ	100	0
ﾄﾞﾒｲﾝ・ｴﾗｰ	76	24
ｺﾝﾋﾟｭﾃｰｼｮﾝ・ｴﾗｰ	100	0

Fig. 10　検出結果（人手有）

正しいプログラム)

```
if(c > 0) {
  a = 10 - c;
  b = 10 + c;
} else if(c == 0) {
  a = 20;
  b = 30;
} else {
  a = 10 + c;
  b = 10 - c;
}
```

ｻﾌﾞｹｰｽ・ｴﾗｰ例)

```
if(c > 0) {
  a = 10 - c;
  b = 10 + c;
} else {
  a = 10 + c;
  b = 10 - c;
}
```

ﾄﾞﾒｲﾝ・ｴﾗｰ例)

```
if(c > 1) {
  a = 10 - c;
  b = 10 + c;
} else if(c == 0) {
  a = 20;
  b = 30;
} else {
  a = 10 + c;
  b = 10 - c;
}
```

ｺﾝﾋﾟｭﾃｰｼｮﾝ・ｴﾗｰ例)

```
if(c > 1) {
  a = 10 - c;
  b = 10 + c;
} else if(c == 0) {
  a = 20;
  b = 30;
} else {
  a = 10 + c;
  b = 10 - c;
}
```

Fig. 8　3種類のエラーの例示

次に、詳細について考察する。

1)ｻﾌﾞｹｰｽ・ｴﾗｰ，ｺﾝﾋﾟｭﾃｰｼｮﾝ・ｴﾗｰ

検出できないエラーは、ある特定のデータが、ある特定の順序で与えられるときのみ発生する。本システムのテストデータ生成では、特に配列に対するデータにおいては、そこに出現するデータの組みあわせだけを決定する。しかし、そのデータがどのような順序で与えられるかということまでは決定しない。そこで、データの順序に依存するエラーの検出を行えるようにするためには、人手の介入が必要になる。

2)ﾄﾞﾒｲﾝ・ｴﾗｰ

検出できないエラーは、ある集合に属しているか、いないかによって、処理を分ける条件文に誤りがある場合である。例えば、前述の例の文字列'st'に対してテストデータを生成する場合、文字列の構成要素である文字に何を割り当てるかを決定する必要がある。本システムでは、そのために、テストデータ・ライブラリを参照する。しかし、そのエラーを発見するためのデータがテストデータ・ライブラリに存在しない場合は、そのエラーを発見できない。また、そのようなエラーを発見できるようにするため、すべてのデータを組み入れた場合、生成されるテストデータが指数関数的に増大するので実用的ではないと考えられる。

5．おわりに

以上の結果より、仕様記述を基に作成されたプログラムと、仕様記述との間の等価性を、本システムにより生成されたテストデータを用いて検証することができると考えられる。

今後の課題としては、以下の２点が挙げられる。

・入力変数の数が増加すると、生成されるテストデータの数が指数関数的に増大する。そこで、その生成されたテストデータの組みあわせに対して、同値分析が行えるようにする。

・ドメイン・エラーの検出効率を向上させるために、テストデータとして集合を扱えるようにする。

参考文献

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

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

[3] 野木兼六，古川善吾，保田勝通："ソフトウェアテスト項目作成支援システム"，日立評論 Vol.66，No.3（1984），pp.29-32

[4] M.Bidoit，B.Biebow，M.C.Gaudel："Exception Handling:Formal Specification and Systematic Program Construction"，IEEE Trans. on S.E. Vol.SE-11，No.3（1985），pp.242-252

[5] F.W.Beichter，O.Herzog，H.Petzsch："SLAN-4:A Software Specification and Design Language"，IEEE Trans. on S.E. Vol.SE-10，No.2（1984），pp.155-162

[6] F.W.Beichter，O.Herzog，H.Petzsch："SLAN-4:A Language for the Specification and Design of Large Software System"，IBM J.Res.Develop.，Vol.27，No.6（1983），pp.558-576

[7] B.W.Kernighan，P.J.Plauger 木村泉訳 "ソフトウェア作法"，共立出版（1981）