VDM++仕様を用いた テストケース自動生成ツールBWDMにおける if-then-else式の構造認識手法の提案

立山 博基[1,a)]　片山 徹郎[1,b)]

概要：形式手法を用いたソフトウェア開発は，仕様の曖昧さを防ぐ．我々は，形式手法 VDM を用いたソフトウェア開発におけるテスト工程の効率化を目的として，テストケース自動生成ツール BWDM を試作した．既存の BWDM の問題点として，VDM++仕様中の if-then-else 式からテストケース生成を行う際に，ネストや else if などの条件式同士の構造中にある戻り値に対するテストケース生成を行えない点が挙げれられる．本稿では，この BWDM における，テストケース生成処理の問題点を解決するために，if-then-else 式の構造認識に基づいたテストケース生成手法を提案する．提案手法を BWDM 上に実装し，VDM++仕様に適用した結果，従来の BWDM ではテストケース生成を行えなかった if-then-else 式中の戻り値に対する，テストケースの生成が可能なことを確認した．このことから，BWDM に提案手法を実装することによって，既存のテストケース生成処理の問題点を解決できたため，BWDM の有用性が向上したと言える．

キーワード：形式手法，VDM++，ソフトウェアテスト，境界値分析，テストケース自動生成

1. はじめに

自然言語は元来，曖昧さを含んでいる [1]．そのため，実装者が，仕様書上の表記を，仕様の作成者が本来意図していない意味で捉えてしまうことが起こる．実装者が，仕様書の本来の意図から外れた認識に基づいて実装を行った結果，ソフトウェアにバグが混入されてしまう．

この問題を解決するための 1 つの手段として，形式手法 (Formal Methods)[2] が提案されている．形式手法を用いた開発では，まず，数理論理学を基盤とした形式仕様記述言語 (Formal Specification Language) により，開発対象が持つ特性を仕様として記述する．形式手法による仕様は，記述した内容に関する数学的な定理証明や機械的な検査が可能であり，自然言語の持つ曖昧さを排除した，厳密な仕様を作成することが可能となる．

一方，ソフトウェアの信頼性向上のために，テストも重要な作業である．テストを行うためには，テストケースの設計作業が必要であるが，人手によるテストケースの設計には手間と時間がかかる．そのため，テストケースの設計作業を効率よく行うことで，テスト工程を効率化できる．また，バグが潜みうる箇所に的を絞ったテストケースの設計により，バグを効率的に発見できる．これらの要求を満たすテスト技法として，境界値分析が一般的に知られている [3]．境界値分析は，ループの終了条件や判定条件などの「境界」付近に不具合が混入しやすいという経験則に基づいたテスト設計技法である．

以上の背景から，形式仕様を基にしたテストケース自動生成ツール BWDM の試作を行った [4], [5], [6]．BWDM は，形式仕様記述言語 VDM++で記述した仕様とデシジョンテーブルを入力とし，VDM++仕様を基に境界値分析を行い，テストケースを自動生成する．デシジョンテーブルとは，ソフトウェアの入力に対する出力を表形式でまとめたものであり，テストの際に，ソフトウェアの期待出力を確認する際に有用である [3]．BWDM の入力に用いるデシジョンテーブルは，本研究室で開発したデシジョンテーブル生成支援ツールを用いて生成する [7]．このツールは，VDM++仕様を入力として，デシジョンテーブルを出力する．そのため，ユーザーは VDM++仕様のみを用意するだけでよい．

BWDM の問題点として，ネストや else if などの if-then-else 式同士の構造の中にある戻り値に対して，テストケース生成を行えない点が挙げられる．BWDM が出力したテ

1　宮崎大学
University of Miyazaki
a)　tachiyama@earth.cs.miyazaki-u.ac.jp
b)　kat@cs.miyazaki-u.ac.jp

©2017 Information Processing Society of Japan

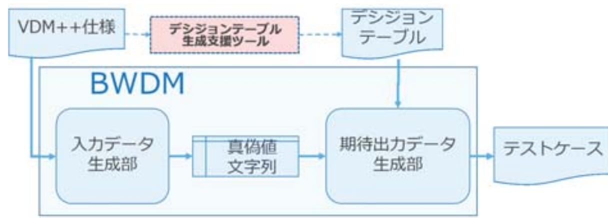

図 1　BWDM の構成
Fig. 1　The structure of BWDM

ストケースを用いることで，境界値テストを行うことが可能となるが，テストを行う上では，パスカバレッジも重要となる．既存の BWDM は，仕様中の if 条件式と引数型に対して，個々に境界値分析を行い，境界値を求めている．そのため，if-then-else 式がネストなどの構造をとっている場合，現状の境界値分析によって生成したテストケースのみでは，C1 カバレッジ [3] を 100%にすることができない．

そこで本稿では，この問題の解決を目的として，if-then-else 式の構造認識に基づいたテストケース生成手法を提案する．具体的には，if-then-else 式の構造を認識し，全ての戻り値毎に，入力データが満たすべき条件式とその真偽値を導出する．それを基に，C1 カバレッジが 100%となるテストケースを生成する．なお，本研究でのテストケース生成には，乱数生成を用いる．

本論文の構成は次の通りである．第 2 章では，既存の BWDM と問題点を示す．第 3 章では，提案手法と実装，及び適用例を示す．第 4 章では，ツールと人手によるテストケース生成効率の比較，及び関連研究について述べる．第 5 章では，本研究のまとめと今後の課題を示す．

2.　テストケース自動生成ツール BWDM

本章では，既存のテストケース自動生成ツール BWDM の構成，機能，処理，入出力，及び，問題点について記す．BWDM とは，Boundary Value と Vienna Development Method の頭字語である．W は Value と Vienna の 2 つの V を意味する．

2.1　BWDM の構成・機能・処理

図 1 に，BWDM の構成を示す．BWDM は入力データ生成部と期待出力データ生成部によって構成しており，それぞれの役割は以下の通りである．

- 入力データ生成部
  - VDM++仕様読み込み
  - 構文解析
  - データ抽出
  - 境界値分析
- 期待出力データ生成部
  - デシジョンテーブル読み込み
  - 期待出力データ導出
  - テストケース生成

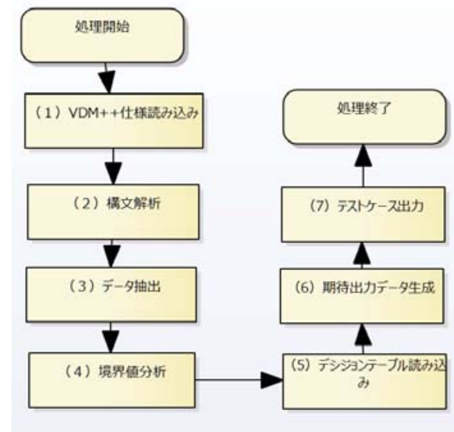

図 2　BWDM の処理の流れ
Fig. 2　The flow of BWDM

図 2 に，BWDM の処理のフローチャートを示す．処理は以下の流れで行う．

(1) 指定された VDM++仕様ファイルを読み込む
(2) VDM++仕様ファイルを構文解析する
(3) 関数定義中の引数型と if-then-else 式中の条件式に関する情報を抽出する
(4) 抽出した情報に対して境界値分析を行う
(5) デシジョンテーブルを読み込む
(6) 境界値とデシジョンテーブルから期待出力データを導出する
(7) 境界値と期待出力データをテストケースとして併せて出力する

構文解析は，Nick Battle 氏の開発した VDM 静的解析ツール VDMJ を用いて実現している [8]．境界値分析は，抽出した引数型，if-then-else 式中の条件式のそれぞれに対して行う．

BWDM は期待出力データの導出を，境界値とデシジョンテーブルの照らし合わせによって行う．この際に使用するデシジョンテーブルは，本研究室で開発したデシジョンテーブル生成支援ツール [7] によって生成したものであり，csv 形式 (comma-separeted values) で出力する．最終的なテストケースも，csv 形式でファイルに列挙して出力する．

BWDM は現状，整数値の境界値生成にのみ対応しているため，以降の説明で登場する条件式の変数と生成する境界値の型は全て整数型に限定する．また，テストケース生成処理は，関数定義中の引数型と if-then-else 式のみを対象に行い，現在は関数の引数を 2 つまでに限定している．

2.2　BWDM の入出力

入出力に関係するファイルを，以下に示す．

- 入力
  - VDM++仕様ファイル (テキスト形式)

©2017 Information Processing Society of Japan

```
1 class SampleClass
2
3 functions
4
5 sampleFunction : int*nat − > seq of char
6   sampleFunction(a, b) ==
7   if(a <= 10) then
8     if(-1 > a) then
9       "a <= 10 and -1 > a"
10    else if(b > 0) then
11      "a <= 10 and !(-1 > a) and b > 0"
12    else
13      "a <= 10 and !(-1 > a) and !(b > 0)"
14  else
15    if(b <= -3) then
16      "!(a <= 10) and b <= -3"
17    else if(a < 20) then
18      "!(a <= 10) and !(b <= -3) and a < 20"
19    else
20      "!(a <= 10) and !(b <= -3) and !(a < 20)";
21
22 end SampleClass
```

図 **3** 入力例：VDM++仕様

Fig. 3 An input example of a VDM++ specification file

引数の個数:2						
引数型:	第1引数:in	第2引数:nat				
テストケースNo.	入力データ		-->	期待出力データ		
No.1	intMin-1	natMin-1	-->	"Undefined Action"		
No.2	intMin-1	natMin	-->	"Undefined Action"		
No.3	intMin-1	natMax	-->	"Undefined Action"		
No.4	intMin-1	natMax+1	-->	"Undefined Action"		
No.5	[illegible]	1	[illegible]	[illegible]		
No.66	20	natMax	-->	"a > 10 and b > -3 and a >= 20"		
No.67	20	natMax+1	-->	"Undefined Action"		
No.68	20	1	-->	"a > 10 and b > -3 and a >= 20"		
No.69	20	-3	-->	"a > 10 and b <= -3"		
No.70	20	-2	-->	"a > 10 and b > -3 and a >= 20"		

図 **4** 出力例：テストケース

Fig. 4 An output example of testcases

– デシジョンテーブルファイル (csv 形式)

- 出力

– テストケースファイル (csv 形式)

VDM++仕様ファイルの例を図 **3** に，そのファイルから生成したテストケースの例を図 **4** に，それぞれ示す．図 4 の入力データ中に存在する $intMin, intMin - 1, intMax, intMax + 1$ などは，図 3 中の関数定義における引数型 (int) に対して境界値分析を行い，生成した境界値である．それぞれ,「引数型 int の最小値 (Minimum)」,「引数型 int の最小値より 1 小さい値」,「引数型 int の最大値 (Maximum)」「引数型 int の最大値より 1 大きい値 (Maximum)」を表す．引数型の範囲外の値 (*Min-1, *Max+1) は，桁溢れにより，プログラムの動作が予測不可能であるため，期待出力データは Undefined Action としている．

2.3 BWDM の問題点

既存の BWDM は，構文解析により VDM++仕様中の

```
1 class ProblemClass
2
3 functions
4
5 problemFunction : nat -> seq of char
6   problemFunction(a) ==
7     if(a mod 4 = 0) then
8       if(a > 92) then
9         "a mod 4 = 0 and a > 92"
10      else
11        "a mod 4 = 0 and !(a > 92)"
12    else
13      "!(a mod 4 = 0)";
14
15 end ProblemClass
```

図 **5** 現状の課題：ネストした if-then-else 式

Fig. 5 Current issue : if-then-else statements which have nest structure

引数の個数:1					
引数型:	第1引数:nat				
テストケースNo.	入力データ	-->	期待出力データ		
No.1	natMin-1	-->	"Undefined Action"		
No.2	natMin	-->	"a mod 4 = 0 and !(a > 92)"		
No.3	natMax	-->	"!(a mod 4 = 0)"		
No.4	natMax+1	-->	"Undefined Action"		
No.5	4	-->	"a mod 4 = 0 and !(a > 92)"		
No.6	3	-->	"!(a mod 4 = 0)"		
No.7	5	-->	"!(a mod 4 = 0)"		
No.8	93	-->	"!(a mod 4 = 0)"		
No.9	92	-->	"a mod 4 = 0 and !(a > 92)"		

図 **6** 図 5 から従来の BWDM が生成したテストケース

Fig. 6 Testcases generated by BWDM from Fig. 5

個々の if-then-else 式の条件式の抽出及び境界値分析を行う．しかし，ネストや else if などの構造を持つ if-then-else 式が仕様中に存在する場合も，構造を無視して個々の条件式として抽出し，それら 1 つ 1 つに対して境界値分析を行っている．そのため，複数の if 条件式が関わりあった構造の中にある戻り値に対する境界値の生成は行えないので，C1 カバレッジを 100%にすることができない．

図 **5** と図 **6** に，現在の BWDM で問題となるネストした if-then-else 式，及び，そこから生成したテストケースを示す．図 5 には，整数値 a に関する 2 つの条件式，a mod 5 = 0 と a > 92 がネストしている．戻り値は文字列である．なお，戻り値の文字列として，その戻り値を返す際に a が満たすべき条件式を記述したが，これはツールの処理や出力には関係しない．

図 6 の期待出力データ中に文字列 "a mod 4 = 0 and a > 92" が存在しないことが確認できる．BWDM は境界値分析により，1 つ目の a mod 4 = 0 からは 3, 4, 5 を生成し，2 つ目の a > 92 からは 92, 93 を境界値として生成するが，その中には，文字列 "a mod 4 = 0 and a > 92" を出力する制御フローへの入力が存在していない．文字列 "a mod 4 = 0 and a > 92" を出力するためには，a mod 4 = 0 が True となり，かつ，a > 92 が True となる入力を生成

©2017 Information Processing Society of Japan

表 1　図 5 における戻り値及びその際満たすべき条件式と真偽値
Table 1　The return value and the conditional expression and the boolean value of Fig. 5

戻り値	条件式：真偽値
"a mod 4 = 0 and a > 92"	a mod 4 = 0 : True, a > 92 : True
"a mod 4 = 0 and !(a > 92)"	a mod 4 = 0 : True, a > 92 : False
"!(a mod 4 = 0)"	a mod 4 = 0 : False

する必要がある．このような if-then-else 式による構造関係を認識し，その先の戻り値に対する入力を生成できるように，すなわち，C1 カバレッジが 100%となるテストケースを生成できるように BWDM を改良する必要がある．

本研究では，if-then-else 式の制御フローを解析し，木構造を作成することで，この問題を解決する．なお，本研究の範囲では，制御フローの解析に主眼を置き，入力データの生成には乱数生成を用いた．

以下に，本研究の提案手法で生成するテストケースを定義する．

(1) if-then-else 式の制御フローから条件式を導き，それを満たす入力データである
(2) 入力データ生成には，乱数生成を用いる

3. if-then-else 式の構造認識及びテストケース生成

本章では，2.3 節で示した問題点を改良するための手法，及び，BWDM 上でその処理を行う実装について述べる．

3.1 提案手法

複数の if-then-else 式による構造の中の戻り値に対する入力データを生成するために，制御フロー解析による木構造を生成する．**表 1** に，2.3 節で示した図 5 における，戻り値を決定する要素に至る条件式とその際の真偽値を示す．1 つの戻り値に対して，満たすべき条件式と真偽値の組み合わせは複数個存在しうる．if-then-else 式のネストの中の戻り値への制御フローでは，複数回の条件判定が行われるからである．

このような入力データを，人手で作成するのであれば，まず図 5 内の if-then-else 式を読み解き，表 1 を導出する必要がある．

すなわち，9 行目の戻り値 "a mod 4 = 0 and a > 92" を出力するためには，7 行目の a mod 4 = 0 と 8 行目の a > 92 の両方が True である必要がある．

同様に，11 行目の戻り値 "a mod 4 = 0 and !(a > 92)" を出力するためには，7 行目の a mod 4 = 0 が true かつ 8 行目の a > 92 が False である必要がある．

また，13 行目の戻り値 "!(a mod 4 = 0)" を出力するためには，7 行目の a mod 4 = 0 が False である必要がある．

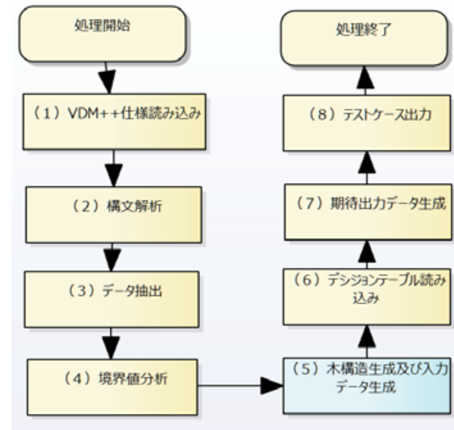

図 7　提案手法による処理を加えた BWDM の処理の流れ
Fig. 7　The flow of BWDM with proposed method

このように，満たすべき条件式と真偽値を導出し，そして，その条件を満たす整数値を 1 つずつ挙げていけばよい．

このような処理を BWDM 上で実現するために，今回は if-then-else 式の制御フローから木構造を生成し，必要な条件式と真偽値を導出するために利用する．入力データ生成には乱数生成を用い，生成した乱数が条件式とその真偽値を満たす場合に，入力データとして採用する．

なお，提案手法の実装は，既存の BWDM に機能追加の形で行い，既存の BWDM の境界値分析の実装及び処理とは全く関係しない．BDWM は，従来の境界値分析の終了後に，今回の手法による処理を行う．そのため，最終的に BWDM が出力するテストケースは，従来の境界値分析によるテストケースと，本研究による制御フローから生成した木構造によるテストケースから構成し，それにより C1 カバレッジを 100%にすることを目指す．

図 7 に，提案手法の処理を加えた BWDM の処理の流れを示す．次節では，提案手法を実現するために実装した処理について説明する．

3.2 実装

実装した処理の手順を以下に示す．また，手順のうち (1)，(2)，(3) を，**図 8** に示す．

(1) VDM++ファイルから if-then-else 式を抽出
(2) 抽出した if-then-else 式から木構造を生成
(3) 末端の各戻り値のノードからルートまで辿り，条件式と真偽値を取得
(4) 乱数を生成し，条件式と真偽値を満たしたものを戻り値と併せてテストケースとして出力

なお，一般的なプログラミング言語とは異なり，VDM++ の関数定義では，if 式は必ず then 節と else 節を伴うことが文法で規定されている．この点を前提に，本研究では実装を行っている．

©2017 Information Processing Society of Japan

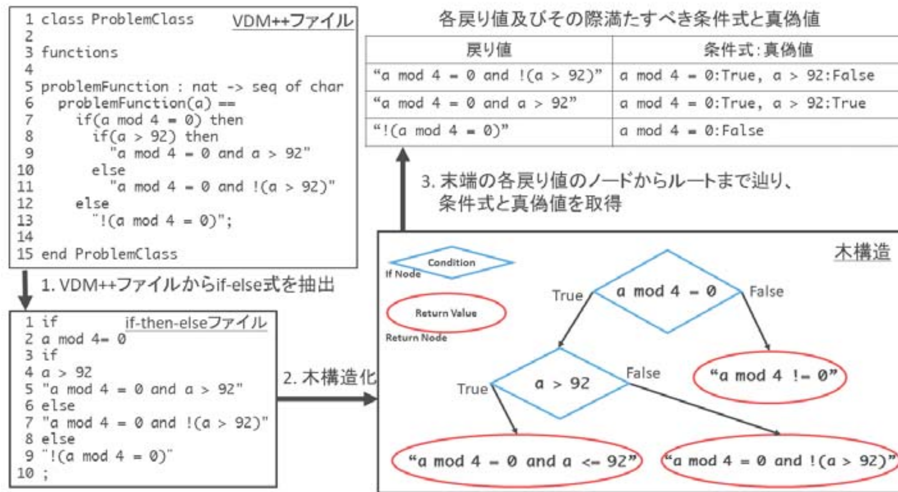

図 8　提案手法の処理の流れ
Fig. 8　The flow of the proposed method

3.2.1　VDM++ファイルから if-then-else 式を抽出

VDM++ファイルを読み込み，if-then-else 式とその終わりのセミコロンまでの部分のトークンを抽出する．クラス名や関数名などの if-then-else 式以外のトークン，及び，if-then-else 式中に存在する（）や then などのトークンを取り除く．また，本来の条件式は，左辺，演算子，右辺などのようにそれぞれ別のトークンからなるが，今回抽出する際は，1 つにまとめて条件式のトークンとして扱う．

抽出したトークンは図 8 の左下に示すように，1 行に 1 つずつ並べ，if-then-else ファイルとして出力する．

3.2.2　抽出した if-then-else 式から木構造を生成

3.2.1 節で出力した if-then-else ファイルから，図 8 の右下に示すように，木構造を生成する．木構造の生成は，if-then-else ファイルから 1 行（= 1 トークン）ずつ読み込みながら行う．木構造は If ノードと Return ノードからなる．If ノードと Return ノードそれぞれが持つ情報を，以下に示す．

- If ノード
  - String conditionString … 条件式
  - Node parentNode … 親ノード
  - Node trueNode … 条件式が True だった場合（then 節）の子ノード
  - Node falseNode … 条件式が False だった場合（else 節）の子ノード
  - Boolean isIfNode … 自身が If ノードか否か
  - Boolean isTrueNode … 自身が親ノードの trueNode か否か
- Return ノード
  - String returnValue … 戻り値
  - Node parentNode … 親ノード
  - Boolean isIfNode … 自身が If ノードか否か
  - Boolean isTrueNode … 自身が親ノードの trueNode か否か

どちらのノードも，自身が If ノードか否かを表す isIfNode と，自身の親ノードを参照する parentNode を持つ．これらは，木構造生成後にノードを辿る時と，その際に入力データ生成に使用する条件式とその真偽値を得るために用いる．

If ノードは更に，自身の条件式である conditionString と，条件式が True の場合の then 節に該当する trueNode と，条件式が False の場合の else 節に該当する falseNode を子ノードに持つ．

Return ノードは，自身の戻り値である returnValue を持つ．

以下に，木構造生成のアルゴリズムを示す．最初にルートとして If ノードを生成し，以降の処理を再帰的に行う．なお，ルートの parentNode は Null，isIfNode は True，isTrueNode は Null とする．

(1) If ノード生成
  (a) 次行の条件式トークンを conditionString へ代入
  (b) 親ノードを設定
  (c) isIfNode に True を代入
(2) 子ノードの trueNode を生成
  (a) もしも次のトークンが if ならば，If ノードを生成するため（1）に戻る
  (b) もしも次のトークンが戻り値ならば，Return ノードを生成し，戻り値のトークンを returnValue に代入
  (c) 生成したノードの parentNode に If ノードを代入，isTrueNode に True を代入
    (i) もしも生成したノードが If ノードならば，isIfNode に True を代入

(ii) もしも生成したノードが Return ノードならば，isIfNode に False を代入

(3) 子ノードの falseNode を生成

(a) もしも次のトークンが if ならば，If ノードを生成するため（1）に戻る

(b) もしも次のトークンが戻り値ならば，Return ノードを生成し，戻り値のトークンを returnValue に代入

(c) 生成したノードの parentNode に If ノードを代入，isTrueNode に False を代入

(i) もしも生成したノードが If ノードならば，isIfNode に True を代入

(ii) もしも生成したノードが Return ノードならば，isIfNode に False を代入

3.2.3 末端の各戻り値のノードからルートまで辿り，条件式と真偽値を取得

3.2.2 節で生成した木構造を辿り，図 8 の右上に示すように，戻り値とその条件式と真偽値の組み合わせを得る．木構造を辿る際は，ルートから順に子ノードへと読み込む．

If ノードを読み込んだ場合，その trueNode と falseNode を読み込む．

Return ノードを読み込んだ場合，そこから親のノード方向への読み込みを開始し，ルートまで読み込む．あるノードに至るための条件式と真偽値は，その親の conditionString の条件式（文字列）と，そのノード自身の isTrueNode の真偽値で決まる．例えば，あるノードの isTrueNode が True だった場合，そのノードは親ノードの trueNode，if-then-else 式の then 節に該当するノードである．そのため，そのノードに至るためには，親の conditionString に格納されている条件式が True になればよい．

これを末端の Return ノードからルートまで繰り返すことで，図 8 の右上に示すような，その Return ノードに至るための条件式と真偽値を得ることができる．表 1 と見比べると，実装した処理によって同じ表を得られていることを確認できる．

3.2.4 乱数を生成し，条件式と真偽値を満たしたものを戻り値と併せてテストケースとして出力

3.2.3 節で得た，条件式と真偽値と戻り値の組み合わせを満たす入力データを生成する．

乱数生成を行い，条件式と真偽値の組み合わせを満たせば，その数値を入力データとして，戻り値と併せてテストケースとする．これを戻り値の数だけ繰り返し，全ての戻り値のテストケースを生成する．

ここで，条件式が充足不能である場合，入力データの生成は不可能である．そのため，入力データ生成に時間制限を設け，一定時間が経過した場合は，入力データ生成処理を終了する．

引数の個数:1			
引数型:	第1引数:nat		
テストケースNo.	入力データ	-->	期待出力データ
No.1	natMin-1	-->	"Undefined Action"
No.2	natMin	-->	"a mod 4 = 0 and !(a > 92)"
No.3	natMax	-->	"!(a mod 4 = 0)"
No.4	natMax+1	-->	"Undefined Action"
No.5	4	-->	"a mod 4 = 0 and !(a > 92)"
No.6	3	-->	"!(a mod 4 = 0)"
No.7	5	-->	"!(a mod 4 = 0)"
No.8	93	-->	"!(a mod 4 = 0)"
No.9	92	-->	"a mod 4 = 0 and !(a > 92)"
No.10	3424	-->	"a mod 4 = 0 and a > 92"
No.11	52	-->	"a mod 4 = 0 and !(a > 92)"
No.12	1217	-->	"!(a mod 4 = 0)"

図 9 図 5 から改良した BWDM で生成したテストケース
Fig. 9 Testcases generated by BWDM with the proposed method from Fig. 5

3.3 結果

図 9 に，2.3 節で問題に挙げた図 5 に対して，提案手法を実装した BWDM を適用することで生成したテストケースを示す．2.3 節に示した，提案手法の実装を行う前の出力である図 6 と比較すると，図 5 中に存在する全ての戻り値が，期待出力データとして存在していることを確認でき，C1 カバレッジを 100%にできた．

4. 考察

本稿では，VDM++仕様を基にしたテストケース自動生成ツール BWDM において，if-then-else 式同士の構造の中にある戻り値に対するテストケース生成を行えない問題を解決するために，if-then-else 式の構造認識に基づいたテストケース生成手法の提案を行った．

提案手法は，まず，VDM++ファイルから if-then-else 式を抽出した．次に，抽出した if-then-else 式を元に，木構造を生成した．そして，生成した木構造の各末端の戻り値のノードからルートに向かって辿ることによって，入力データを生成するために必要な条件式と真偽値の組み合わせを得た．最後に，乱数を生成し，条件式と真偽値の組み合わせを満たすものを入力データとし，戻り値とセットでテストケースとして出力した．新たに生成したテストケースで，C1 カバレッジ 100%を達成できた．

今回提案した手法により，if-then-else 式同士の構造の中にある戻り値に対するテストケースを生成できることを確認できた．よって，既存の BWDM のテストケース生成手法に，今回提案したテストケース生成手法を追加で実装することによって，if-then-else 式同士の構造の中にある戻り値に対するテストケース生成を行えない既存の問題を解決できる．

4.1 人手によるテストケース作成との比較検証

被験者と BWDM とで，テストケース作成に要した時間

表 2　テストケース作成に要した時間の比較
Table 2　The required time for testcases generation

被験者 or BWDM	時間
被験者 A	2m 38s
被験者 B	2m 40s
被験者 C	2m 50s
被験者 D	2m 06s
BWDM	38s

の比較検証を行った．その結果を，**表 2** に示す．

対象とした VDM++仕様は，2.3 節で用いた図 5 である．記述してある戻り値 “a mod 4 = 0 and a > 92”，“a mod 4 = 0 and !(a > 92)”，“!(a mod 4 = 0)” に対するテストケース 3 件を作成する時間を計測した．生成するテストケースとしては，以下を基準とした．

(1) if-then-else 式の制御フローから条件式を導き，それを満たす入力データである
(2) 入力データ生成には，乱数生成を用いる

検証に参加したメンバーは本研究室の大学院生 4 人であり，普段からソースコードの読み書きを行い，基本的なプログラミングの知識を有している．VDM++の文法の知識を持たない者も含まれるが，今回の検証に必要な文法は，事前に他の VDM++の例を用いてレクチャーした．

人手による検証では，パソコン上に図 5 を表示させ，エディタ上に，各自が考えた入力データ及びその際の戻り値の組み合わせを記述するのに要した時間を計測した．入力データと戻り値の組み合わせが不正確な場合，間違いを指摘し，被験者が正しい組み合わせを記述した時点で時間計測終了とした．また，BWDM が乱数生成を用いているため，被験者が作成するテストケースも，境界値ではないランダムなものでもよいものとした．

BWDM による検証では，デシジョンテーブル生成支援ツールによるデシジョンテーブルファイルの生成と，BWDM によるテストケースファイル生成をマウス操作で行うのに要した時間を計測した．

なお，Java の System.nanoTime[9] メソッドを用いて，マウス操作を省いた純粋なテストケース生成処理に BWDM が要した時間を計測した結果，0.41 秒であった．

人手と比較した結果，1 分半から 2 分程の時間短縮を確認できた．対象にした VDM++仕様には，VDM++独特の文法等は含まれないため，VDM++に対する慣れなどの影響は無視できるものと思われる．また，人手によるテストケース生成の場合，ヒューマンエラーも見られた．具体的には，「a mod 4 = 0 かつ a > 92」を満たす入力データを考える際に，> と >= を読み違えてしまい，92 を入力データとして挙げてしまった．仕様の規模が拡大すると，人手とコンピュータとの処理効率の差に加えて，ヒューマンエラーの有無などにより，テストケース生成に要する時間の差は更に拡大していくと思われる．

一方で，被験者の中には，今回の if-then-else 式に対する境界値と言える 92, 96 を入力データとして作成するものも含まれた．木構造から上記のような境界値を生成することは，BWDM の今後の課題である．

4.2　関連研究

記号実行 (Symbolic Execution)[10] を用いることで，今回の提案手法同様に，if 式が入れ子になった場合も全ての戻り値に対してテストケース生成を行った事例として，高松らの研究 [11] がある．記号実行は，各戻り値に対する制約条件 (条件式) を導出する．また，SMT(Satisfiable Modulo Theories) ソルバを用いて，制約条件を満たす入力値の導出も行われている．

オブジェクト指向プログラミングにおけるソフトウェアテストを行うには，テスト対象のインスタンス生成・状態の変更などの前準備のメソッド実行を，テストケース内で行う必要がある．そのようなテストケース自動生成ツールの 1 つに Seeker[12] があり，高松らは Seeker の機能拡張を行った．Seeker では，記号実行と具体的な値を組み合わせた動的記号実行 (Dynamic Symbolic Execution) を用いて，ソースコード中の条件式を基に，全ての戻り値に対する引数の組を生成している．

VDM とテストに関する研究において，記号実行的アプローチを取り，木構造化した制御フローから各戻り値に至る要素に対する条件式を導出し，入力データを生成したのは，本研究が初である．

また，VDM 仕様を基にしたテストケースの自動生成法に関する研究の 1 つに，馬場らの研究がある [13]．馬場らは，VDM 仕様記述からコーナーケースに対する単体テストを目的に，テストドライバの雛形の作成を行うツールを開発した．馬場らのツールは，VDM-SL 仕様内の操作定義をテストケース作成の対象としており，そのため，VDM++を対象に，関数定義からテストケースを作成する BWDM とは異なる．また，馬場らのツールは，入力データとして記述内の条件式を出力するため，テストの際にはテスト担当者が入力データは指定する必要がある．これに対して，BWDM は具体的な数値の入力データを出力するため，そのままテストを実行できることが BWDM の利点であるといえる．

また，VDM を用いたソフトウェア開発をサポートするツールに，Overture IDE がある [14]．Overture は機能の一つとして，組み合わせテストをサポートしている [15]．Overture は，VDM 仕様中に traces 定義としてに入力されたテストパターンに従い，テストケースを自動生成し，実行する．テストパターンとは，テスト設計者が正規表現を用いて，出力するテストケースを定義したものであり，テストデータは，テスト設計者が指定する．BWDM が生成

したテストデータを使用することで，境界値分析と組み合わせテストを併用したテストを実行可能である．

5. おわりに

本稿では，VDM++を基にしたテストケース自動生成ツール BWDM における，if-then-else 式同士の構造の中にある戻り値に対するテストケース生成を行えない問題を解決するために，if-then-else 式の構造認識に基づくテストケース生成手法を提案した．

提案手法は，まず，VDM++ファイルから if-then-else 式を抽出し，それを元に木構造を生成した．そして，生成した木構造の各末端の戻り値のノードからルートに向かって辿ることで，入力データを生成するために必要な条件式と真偽値の組み合わせを得た．最後に，乱数を生成し，条件式と真偽値の組み合わせを満たすものを入力データとし，戻り値とセットでテストケースとして出力した．生成したテストケースで，C1 カバレッジを 100%にできた．

提案手法により，既存の BWDM では生成できなかった，if-then-else 式による構造の中にある戻り値に対するテストケースの生成が可能なことを確認した．故に，既存の BWDM のテストケース生成処理に，今回提案したテストケース生成処理を追加で実装することによって，if-then-else 式同士の構造の中にある戻り値に対するテストケース生成を行えない問題を解決できる．

ネストしている 2 つの if-then-else 式を含んだ仕様を対象として，テストケース生成に要した時間を人手と比較検証した結果，1 分半から 2 分程の時間短縮を確認できた．

以上により，今回の提案手法に基づいた BWDM へ入力データ生成処理の実装によって，既存の BWDM の問題点を解決し，BWDM の有用性が向上したと言える．

以下に，BWDM の今後の課題を示す．

- 乱数以外による入力データの生成
  本研究では，木構造の生成に主眼を置いたため，入力データ生成に乱数を用いた．現状，条件式の内容や条件式の数次第では，生成完了までに実用性に欠ける時間が掛かってしまう．乱数以外の手法としては，SMT ソルバを用いたものが考えられる．また，戻り値に対する条件式を満たす整数が存在しない場合，戻り値はデッドコードであり，入力データ生成は行えない．本研究では入力データ生成処理が一定時間終わらない場合，その条件式の生成処理を終了し，次の条件式へと移行するが，SMT ソルバにより充足不能な条件式を検知することができる．また，境界値を導出することもテストを行う上で大きな利点がある．
- 未対応の定義部や構文，演算子，型への対応
  操作定義，型定義などの定義，事前条件や事後条件などの構文に既存の BWDM は未対応である．このため，これらの記述からテストケースを生成する手法を考案し，BWDM に実装することで，BWDM の有用性が更に向上すると考えている．
- 両辺が変数である if 条件式への対応
  if 条件式中の両辺が変数である場合，既存の BWDM は境界値分析を行うことができないため，入力データ生成を行えない．このため，そのような if 条件式から境界値を抽出する処理を BWDM に実装することで，BWDM の適用範囲の更なる拡大を見込める．

参考文献

[1] IPA 独立行政法人 情報処理推進機構: なぜ形式手法か, 入手先 ⟨http://sec.ipa.go.jp/users/seminar/seminar_tokyo_20150912-02.pdf⟩ (参照 2017.8.4)
[2] 荒木啓二郎, 張漢明: プログラム仕様記述論, オーム社 (2002)
[3] SQuBOK 策定部会: ソフトウェア品質知識体系ガイド 第 2 版, オーム社 (2014)
[4] 立山博基, 片山徹郎: VDM++仕様を基にした境界値テストケース自動生成ツール BWDM の試作について, 平成 28 年度 電気・情報関係学会 九州支部連合大会, p.90 (2016)
[5] 立山博基, 片山徹郎: VDM++仕様に対する境界値分析を用いたテストケース自動生成, JaSST'16 九州, p.32 (2016)
[6] Hiroki Tachiyama, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, and Naonobu Okazaki: *Prototype of Test Cases Automatic Generation Tool BWDM Based on Boundary Value Analysis with VDM++*, The 2017 International Conference on Artificial Life and Robotics(ICAROB 2017), pp.275-278 (2017)
[7] 西川拳太, 片山徹郎: VDM++を用いたデシジョンテーブル生成支援ツールについて, 平成 25 年度 電気関係学会 九州支部連合大会, p.381 (2013)
[8] Nick Battle: GitHub - nickbattle/vdmj, 入手先 ⟨https://github.com/nickbattle/vdmj⟩ (参照 2017.8.4)
[9] Sun Microsystems: *System(Java Platform SE6)*, 入手先 ⟨https://docs.oracle.com/javase/jp/6/api/java/lang/System.html⟩ (参照 2017.8.4)
[10] James C. King: *Symbolic Execution and Program Testing*, Communications of The ACM Vol.19, No.7, pp. 385-394, DOI: 10.1145/360248.360252 (1976)
[11] 高松宏樹, 佐藤晴彦, 小山聡, 栗原正仁: 動的記号実行によるメソッドの複雑度を考慮したテストケース自動生成, 情報処理学会研究報告 Vol.2014-SE-185, No.27 pp. 1-7, NAID: 110009803966 (2014)
[12] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Zhendong Su: *Synthesizing Method Sequences for High-Coverage Testing*, SIGPLAN Notices Vol.46, No.10, pp.189-206, DOI: 10.1145/2076021.2048083 (2011)
[13] 馬場勇輔, 荒木啓二郎, 日下部茂, 大森洋一: 形式仕様記述のプロパティベーステストへの活用, 情報処理学会火の国シンポジウム予稿集, 入手先 ⟨https://www.ipsj-kyushu.jp/page/ronbun/hinokuni/1005/5C/5C-1.pdf⟩ (参照 2017.8.4)
[14] The Overture Project: Overture Tool Formal Modelling in VDM, 入手先 ⟨http://overturetool.org/⟩ (参照 2017.8.4)
[15] バルテス株式会社 R&D 部 石原一宏: テスト項目数を大幅に削減する「組み合わせテストの効果的活用方法」, 入手先 ⟨http://jasst.jp/symposium/jasst13tokyo/pdf/E3-2.pdf⟩ (参照 2017.8.4)

©2017 Information Processing Society of Japan