VDM++ 仕様を対象とした テストケース自動生成ツールBWDMの 適用範囲拡大による実用性の向上

2023.3.9

宮崎大学大学院 工学研究科 工学専攻

武藤崇史　片山徹郎

目次

- 背景と目的
- 拡張(i)
  背景と目的，手段，適用例
- 拡張(ii)
  背景と目的，手段，適用例
- 拡張(iii)
  背景と目的，手段，適用例
- 考察
- まとめ
- 今後の課題

背景と目的

ソフトウェアにバグが混入する原因の一つとして，上流工程の設計段階で自然言語を一般に用いていることが挙げられる

自然言語の持つ曖昧さにより，仕様書が本来意図していない意味で捉えられてしまう

厳密な仕様を作成するために，形式手法が用いられる

背景と目的

VDM++

- 形式手法の一つにVDM(Vienna Development Method)がある
- 1996年に，VDM-SLが仕様記述言語として世界初のISO標準化
- VDM++はVDM-SLをオブジェクト指向拡張を施した形式仕様記述言語

本研究ではVDM++仕様記述を対象とする

背景と目的

自然言語，あるいは形式手法を用いた設計のいずれにおいても，
実装後にソフトウェアテストを行う必要がある

しかし，
人手によるテストケースの設計は手間と時間がかかる

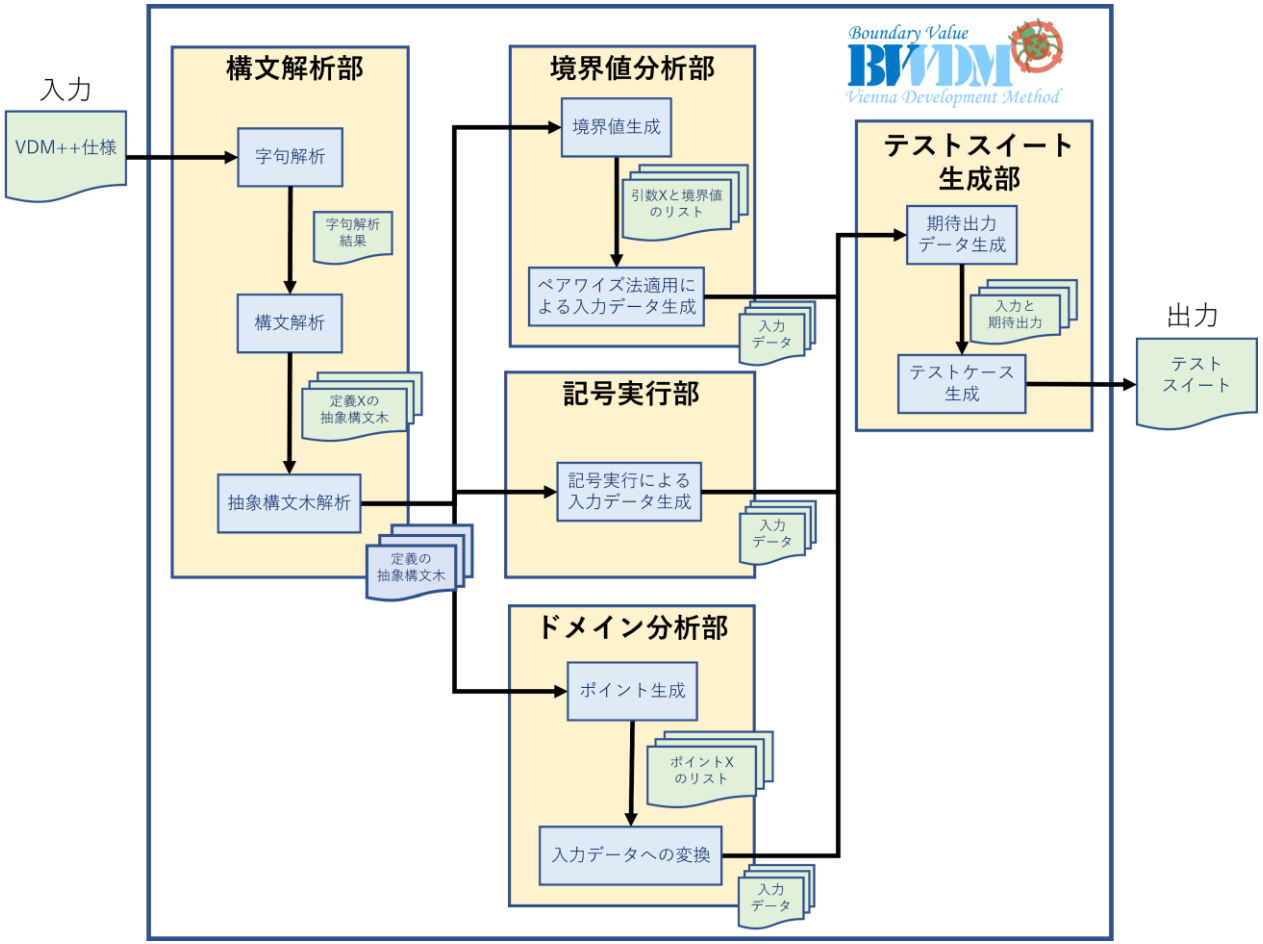
背景と目的

テストケース自動生成ツールBWDM

- 立山氏が開発したBWDM(Boundary Value/Vienna Development Method)を平木場氏が拡張
- VDM++仕様記述を対象に，テストケースを自動で生成するCLIツール
- Java，Kotlinで記述

背景と目的

テストケース自動生成ツールBWDM



- VDM支援ツールVDMJの字句解析機能と構文解析機能を利用する
- 境界値分析，記号実行，ドメイン分析によるテストケースを生成

背景と目的

BWDMの入出力例

```
1: class 暦
2:
3: functions
4:   うるう年判定: nat -> seq of char
5:   うるう年判定(西暦) ==
6:     if(西暦 mod 4 = 0) then
7:       if(西暦 mod 100 = 0) then
8:         if(西暦 mod 400 = 0) then
9:           "うるう年"
10:        else
11:          "平年"
12:      else
13:        "うるう年"
14:    else
15:      "平年";
16:
17: end 暦
```

```
関数名 : うるう年判定
引数の型 : 西暦:nat
戻り値の型 : seq of (char)
生成テストケース数 : 17件(境界値分析:13/記号実行:4)

各引数の境界値
西暦 : 4294967295 4294967294 0 -1 3 4 5 99 100 101 399 400 401

境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> "平年"
No.3 : 0 -> "うるう年"
No.4 : -1 -> Undefined Action
No.5 : 3 -> "平年"
No.6 : 4 -> "うるう年"
No.7 : 5 -> "平年"
No.8 : 99 -> "平年"
No.9 : 100 -> "平年"
No.10 : 101 -> "平年"
No.11 : 399 -> "平年"
No.12 : 400 -> "うるう年"
No.13 : 401 -> "平年"

記号実行によるテストケース
No.1 : 400 -> "うるう年"
No.2 : 100 -> "平年"
No.3 : 4 -> "うるう年"
No.4 : 1 -> "平年"
```

入力エラーの場合は”Undefined Action”と出力する

背景と目的

BWDMの問題点

- 型定義ブロックに対応していない
- 不変条件と事前条件と事後条件に対応していない
- オブジェクトの状態を変更する操作定義のテストケースを 生成できない
- 入れ子構造にある関数定義及び操作定義に対応していない

背景と目的

BWDMの実用性の向上を目的に,

既存のBWDMが持つ4つの問題点を解決して,
BWDMの適用範囲を拡大する

拡張

4つの問題点に対応する4つの拡張

(ⅰ) 不変条件と事前条件と事後条件の解析及び評価

→不変条件と事前条件と事後条件に対応していない問題点を解決

(ⅱ) 型定義ブロックへの対応

→型定義ブロックに対応していない問題点を解決

(ⅲ)オブジェクトの状態に対するテストケース生成機能の追加

→オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点を解決

(ⅳ)入れ子構造にある関数定義及び操作定義への対応

→入れ子構造にある関数定義及び操作定義に対応していない問題点を解決

拡張

4つの問題点に対応する**3つ**の拡張

（ｉ）不変条件と事前条件と事後条件の解析及び評価

→不変条件と事前条件と事後条件に対応していない問題点を解決

→オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点を解決

（ii）型定義ブロックへの対応

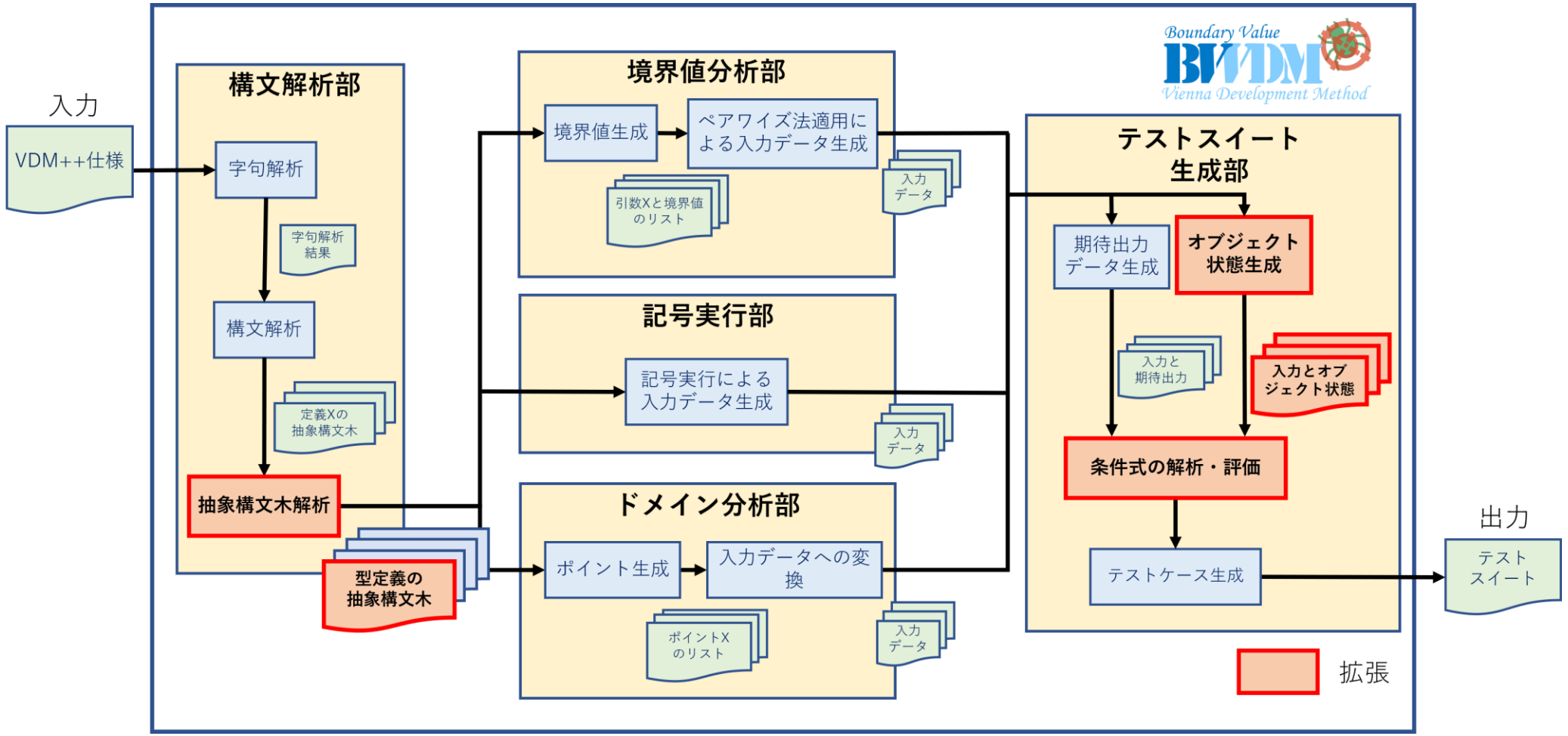
→型定義ブロックに対応していない問題点を解決

（iii）入れ子構造にある関数定義及び操作定義への対応

→入れ子構造にある関数定義及び操作定義に対応していない問題点を解決

拡張

拡張後のBWDMの構造



拡張(ｉ)

4つの問題点に対応する**3つ**の拡張

(ｉ)不変条件と事前条件と事後条件の解析及び評価

→不変条件と事前条件と事後条件に対応していない問題点を解決

→オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点を解決

(ii)型定義ブロックへの対応

→型定義ブロックに対応していない問題点を解決

(iii)入れ子構造にある関数定義及び操作定義への対応

→入れ子構造にある関数定義及び操作定義に対応していない問題点を解決

拡張(ｉ):背景と目的

不変条件と事前条件と事後条件に対応していない問題点

- 不変条件
  - 型定義ブロックとインスタンス変数定義ブロックに記述される
  - オブジェクトが存在している間，常に真となる条件式
  - inv (条件式) で記述される
- 事前条件
  - 操作定義ブロックと定数定義ブロックに記述される
  - 操作または関数が実行される前に真となる条件式
  - pre (条件式) で記述される
- 事後条件
  - 操作定義ブロックと関数定義ブロックに記述される
  - 操作または関数を実行した後に真となる条件式
  - post (条件式) で記述される

拡張(ｉ):背景と目的

不変条件と事前条件と事後条件に対応していない問題点

```
1: class 成績評価
2:
3: functions -- 関数定義ブロック
4:  成績評価 : nat -> seq of char
5:      成績評価(テスト点) ==
6:          if(テスト点 >= 60) then
7:              if(テスト点 >= 70) then
8:                  if(テスト点 >= 80) then
9:                      if(テスト点 >= 90) then
10:                         "秀"
11:                     else
12:                         "優"
13:                 else
14:                     "良"
15:             else
16:                 "可"
17:         else
18:             "不可"
19: pre テスト点 <= 100;
20:
21: end 成績評価
```

```
関数名 : 成績評価
引数の型 : テスト点:nat
戻り値の型 : seq of (char)
生成テストケース数 : 17件(境界値分析:12/記号実行:5)

各引数の境界値
テスト点 : 4294967295 4294967294 0 -1 60 59 70 69 80 79 90 89

境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> "秀"
No.3 : 0 -> "不可"
No.4 : -1 -> Undefined Action
No.5 : 60 -> "可"
No.6 : 59 -> "不可"
No.7 : 70 -> "良"
No.8 : 69 -> "可"
No.9 : 80 -> "優"
No.10 : 79 -> "良"
No.11 : 90 -> "秀"
No.12 : 89 -> "優"
```

事前条件を認識していないため,
間違った期待出力を生成している

拡張(ｉ):背景と目的

オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11: カード利用額:「円」:= 0;
12: inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

- if-then-else式がない定義に対しては入力データを取得できず，テストケースを生成できない

→分岐がなくてもオブジェクトの状態が変更する場合，テストすべき項目がある

- 操作実行後のオブジェクトの状態を考慮せずにテストケースを生成している

→操作の処理内容によっては，実行後のオブジェクトが不正な状態になる場合がある

拡張(ｉ):背景と目的

オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点

```
1: class 支払い
2:
3: types
4:  public「円」= nat;
5:
6: values
7:  カード利用限度額:「円」= 100000;
8:
9: instance variables
10:  保有ポイント: nat := 700;
11:  カード利用額:「円」:= 0;
12:  inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

操作後に検証すべき条件式がある

- if-then-else式がない定義に対しては入力データを取得できず，テストケースを生成できない

→分岐がなくてもテストすべき場合がある

- 操作実行後のオブジェクトの状態を考慮せずにテストケースを生成している

→操作の処理内容によっては，実行後のオブジェクトが不正な状態になる場合がある

拡張(ｉ):手段

不変条件と事前条件と事後条件の解析及び評価

- VDM++仕様では，不変条件と事前条件と事後条件の3種類の条件式を設定できる
- 操作実行後のオブジェクトの状態を算出し，3種類の条件式に適用した際の真偽値を求めて，表示する期待出力を決定する

表示する期待出力	該当する条件
return文の結果	3種類の条件式を全て充足し，正常に出力できる
Input Error	事前条件もしくは引数の制約条件を充足しない
Failure	事後条件か，インスタンス変数の制約条件か，戻り値の制約条件のいずれかを充足しない
ー	3種類の条件式を全て充足するが，出力がない

拡張(ｉ):手段

不変条件と事前条件と事後条件の解析及び評価

```
1: class 支払い
2:
3: types
4:  public「円」= nat;
5:
6: values
7:  カード利用限度額:「円」= 100000;
8:
9: instance variables
10:  保有ポイント: nat := 700;
11:  カード利用額:「円」:= 0;
12:  inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:     保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

引数「金額」が1000の時

- カード利用額 = 1000 (0 + 1000)
- 保有ポイント = 710 (700 + 1000 div 100)

0 <= 710 <= 4294967294

0 <= 1000 <= 4294967294

1000 <= 100000

1000 >= 1000

3種類の条件式を全て充足するが，出力がないため，
期待出力は「ー」となる

拡張(ｉ):手段

不変条件と事前条件と事後条件の解析及び評価

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11:  カード利用額:「円」:= 0;
12:  inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

操作後に検証すべき条件式の境界値を取るための入力値を算出する

- 操作後に検証すべき条件式中のインスタンス変数を，操作の式に置換して境界値を求める

0 + 金額 <= 100000

操作前のカード利用額　カード利用限度額

→金額の入力値(100000と100001)を取得

※入力値は記号実行を用いて取得する

拡張(ｉ):適用例

VDM++仕様

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11: カード利用額:「円」:= 0;
12: inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

拡張後のBWDMに適用した際の出力

```
関数名 : カードで支払う
引数の型 : 金額:nat
戻り値の型 : ()

各引数の境界値
金額 : 4294967295 4294967294 0 -1 1000 999 100000 100001

>> 境界値分析によるテストケース
No.1 : 4294967295 -> Input Error (FAILED: 金額 <= 4294967294)
No.2 : 4294967294 -> Failure (FAILED: カード利用額 <= 100000)
No.3 : 0 -> Input Error (FAILED: 金額 >= 1000)
No.4 : -1 -> Input Error (FAILED: 金額 >= 1000, 金額 >= 0)
No.5 : 1000 -> -
No.6 : 999 -> Input Error (FAILED: 金額 >= 1000)
No.7 : 100000 -> -
No.8 : 100001 -> Failure (FAILED: カード利用額 <= 100000)
```

拡張(ｉ):適用例

VDM++仕様

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11: カード利用額:「円」:= 0;
12: inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

拡張後のBWDMに適用した際の出力

```
関数名 : カードで支払う
引数の型 : 金額:nat
戻り値の型 : ()

各引数の境界値
金額 : 4294967295 4294967294 0 -1 1000 999 100000 100001

>> 境界値分析によるテストケース
No.1 : 4294967295 -> Input Error (FAILED: 金額 <= 4294967294)
No.2 : 4294967294 -> Failure (FAILED: カード利用額 <= 100000)
No.3 : 0 -> Input Error (FAILED: 金額 >= 1000)
No.4 : -1 -> Input Error (FAILED: 金額 >= 1000, 金額 >= 0)
No.5 : 1000 -> -
No.6 : 999 -> Input Error (FAILED: 金額 >= 1000)
No.7 : 100000 -> -
No.8 : 100001 -> Failure (FAILED: カード利用額 <= 100000)
```

操作前・操作後ともに正常であるため，期待出力を「ー」としている
（return文がある場合は，その結果を期待出力とする）

拡張(ｉ):適用例

VDM++仕様

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11: カード利用額:「円」:= 0;
12: inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

拡張後のBWDMに適用した際の出力

```
関数名 : カードで支払う
引数の型 : 金額:nat
戻り値の型 : ()

各引数の境界値
金額 : 4294967295 4294967294 0 -1 1000 999 100000 100001

>> 境界値分析によるテストケース
No.1 : 4294967295 -> Input Error (FAILED: 金額 <= 4294967294)
No.2 : 4294967294 -> Failure (FAILED: カード利用額 <= 100000)
No.3 : 0 -> Input Error (FAILED: 金額 >= 1000)
No.4 : -1 -> Input Error (FAILED: 金額 >= 1000, 金額 >= 0)
No.5 : 1000 -> -
No.6 : 999 -> Input Error (FAILED: 金額 >= 1000)
No.7 : 100000 -> -
No.8 : 100001 -> Failure (FAILED: カード利用額 <= 100000)
```

事前条件か引数の制約条件を充足しないため，期待出力を「Input Error」としている

拡張(ｉ):適用例

VDM++仕様

```
 1: class 支払い
 2:
 3: types
 4:  public「円」= nat;
 5:
 6: values
 7:  カード利用限度額:「円」= 100000;
 8:
 9: instance variables
10:  保有ポイント: nat := 700;
11: カード利用額:「円」:= 0;
12: inv カード利用額 <= カード利用限度額;
13:
14: operations
15: カードで支払う:「円」==> ()
16: カードで支払う(金額) ==
17:     (カード利用額 := カード利用額 + 金額;
18:      保有ポイント := 保有ポイント + 金額 div 100)
19: pre 金額 >= 1000;
20:
21: end 支払い
```

拡張後のBWDMに適用した際の出力

```
関数名 : カードで支払う
引数の型 : 金額:nat
戻り値の型 : ()

各引数の境界値
金額 : 4294967295 4294967294 0 -1 1000 999 100000 100001

>> 境界値分析によるテストケース
No.1 : 4294967295 -> Input Error (FAILED: 金額 <= 4294967294)
No.2 : 4294967294 -> Failure (FAILED: カード利用額 <= 100000)
No.3 : 0 -> Input Error (FAILED: 金額 >= 1000)
No.4 : -1 -> Input Error (FAILED: 金額 >= 1000, 金額 >= 0)
No.5 : 1000 -> -
No.6 : 999 -> Input Error (FAILED: 金額 >= 1000)
No.7 : 100000 -> -
No.8 : 100001 -> Failure (FAILED: カード利用額 <= 100000)
```

不変条件を充足しないため，期待出力を「Failure」としている

拡張(ii)

4つの問題点に対応する**3つ**の拡張

(i) 不変条件と事前条件と事後条件の解析及び評価

→不変条件と事前条件と事後条件に対応していない問題点を解決

→オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点を解決

(ii) 型定義ブロックへの対応

→型定義ブロックに対応していない問題点を解決

(iii)入れ子構造にある関数定義及び操作定義への対応

→入れ子構造にある関数定義及び操作定義に対応していない問題点を解決

拡張(ii):背景と目的

型定義ブロックに対応していない問題点

```
1: class 平成生まれと令和生まれ
2:
3:  types -- 型定義ブロック
4:     public「年」= nat;
5:
6:     public「月」= nat1
7:     inv m == m <= 12;
8:
9:  functions -- 関数定義ブロック
10:    生まれ判定 :「年」*「月」-> seq of char
11:       生まれ判定(年, 月) ==
12:          if(年 <= 2019) then
13:             if(月 < 4) then
14:                "平成の早生まれ"
15:             else
16:                if(月 > 4) then
17:                   "令和の遅生まれ"
18:                else
19:                   "平成の遅生まれ"
20:          else
21:             if(月 < 4) then
22:                "令和の早生まれ"
23:             else
24:                "令和の遅生まれ"
25:    pre 年 > 1990;
26:
27: end 平成生まれと令和生まれ
```

「月」型は
1~12の自然数

- 型定義ブロック(types)で定義されている型が，どのような型であるか特定できない
- 型の境界値を入力とするテストケースを生成できない

「月」型の境界値

- nat1型における境界値：
  0, 1, 4294967295, 4294967296
- 不変条件における境界値：
  12, 13

拡張(ii):背景と目的

型定義ブロックに対応していない問題点

```
1: class 平成生まれと令和生まれ
2:
3: types -- 型定義ブロック
4:     public「年」= nat;
5:
6:     public「月」= nat1
7:     inv m == m <= 12;
8:
9: functions -- 関数定義ブロック
10:    生まれ判定 :「年」*「月」-> seq of char
11:        生まれ判定(年, 月) ==
12:            if(年 <= 2019) then
13:                if(月 < 4) then
14:                    "平成の早生まれ"
15:                else
16:                    if(月 > 4) then
17:                        "令和の遅生まれ"
18:                    else
19:                        "平成の遅生まれ"
20:            else
21:                if(月 < 4) then
22:                    "令和の早生まれ"
23:                else
24:                    "令和の遅生まれ"
25:    pre 年 > 1990;
26:
27: end 平成生まれと令和生まれ
```

型を特定できていない

既存のBWDMに適用した際の出力

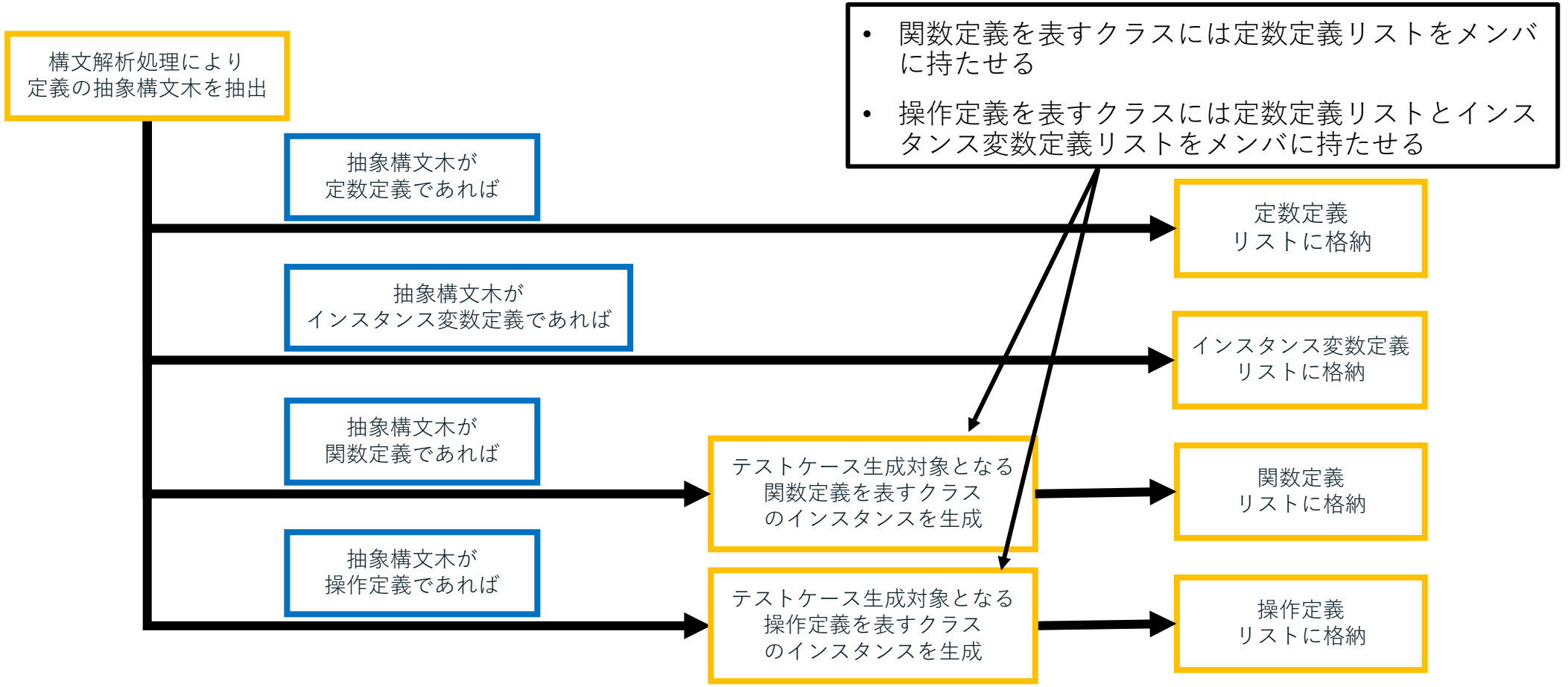
```
関数名 : 生まれ判定
引数の型 : 年:(unresolved 「年」) 月:(unresolved 「月」)
戻り値の型 : seq of (char)
生成テストケース数 : 11件(境界値分析:6/記号実行:5)

各引数の境界値
年 : 2019 2020
月 : 3 4 5

境界値分析によるテストケース
No.1 : 2019 3 -> "平成の早生まれ"
No.2 : 2020 3 -> "令和の早生まれ"
No.3 : 2019 4 -> "平成の遅生まれ"
No.4 : 2020 4 -> "令和の遅生まれ"
No.5 : 2019 5 -> "令和の遅生まれ"
No.6 : 2020 5 -> "令和の遅生まれ"
~
```

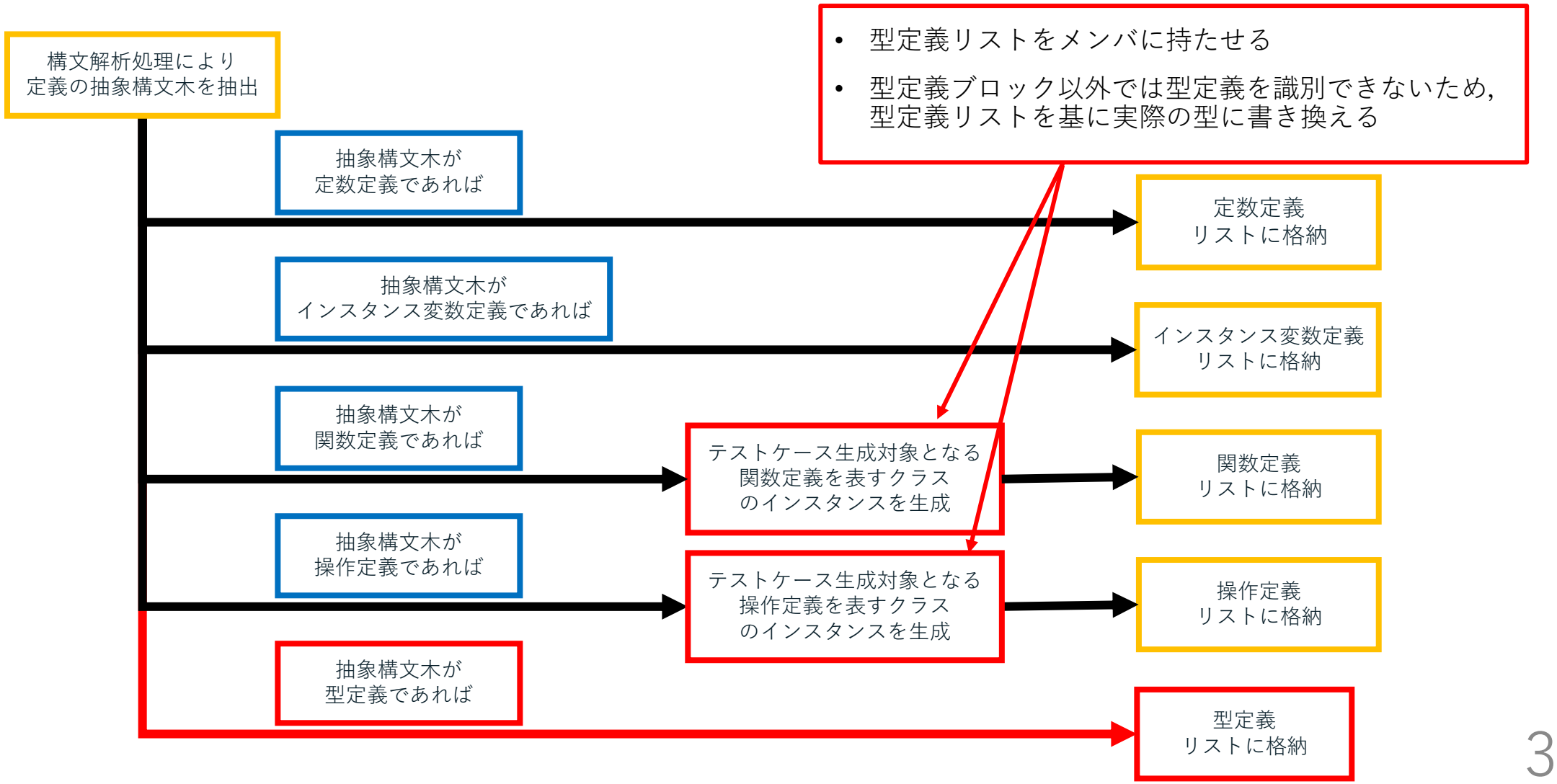
拡張(ii):手段

既存の構文解析部の抽象構文木解析



拡張(ii):手段

既存の構文解析部の抽象構文木解析



拡張(ii):手段

型定義の不変条件に対する境界値生成

```
1: class 平成生まれと令和生まれ
2:
3: types -- 型定義ブロック
4:     public「年」= nat;
5:
6:     public「月」= nat1
7:     inv m == m <= 12;
8:
9: functions -- 関数定義ブロック
10:    生まれ判定 :「年」*「月」-> seq of char
11:        生まれ判定(年, 月) ==
12:            if(年 <= 2019) then
13:                if(月 < 4) then
14:                    "平成の早生まれ"
15:                else
16:                    if(月 > 4) then
17:                        "令和の遅生まれ"
18:                    else
19:                        "平成の遅生まれ"
20:            else
21:                if(月 < 4) then
22:                    "令和の早生まれ"
23:                else
24:                    "令和の遅生まれ"
25:    pre 年 > 1990;
26:
27: end 平成生まれと令和生まれ
```

引数型に「月」型を使用している場合，この式の境界値も生成する

- BWDMはif-then-else式の条件式を境界値分析部に渡して，入力データとなる境界値を生成する
- 引数型に不変条件がある場合，if-then-else式の条件式と同様に不変条件の条件式も境界値分析部に渡す

拡張(ii):適用例

VDM++仕様

```
class 平成生まれと令和生まれ

types -- 型定義ブロック
    public「年」= nat;

    public「月」= nat1
    inv m == m <= 12;

functions -- 関数定義ブロック
    生まれ判定 :「年」*「月」-> seq of char
        生まれ判定(年, 月) ==
            if(年 <= 2019) then
                if(月 < 4) then
                    "平成の早生まれ"
                else
                    if(月 > 4) then
                        "令和の遅生まれ"
                    else
                        "平成の遅生まれ"
            else
                if(月 < 4) then
                    "令和の早生まれ"
                else
                    "令和の遅生まれ"
    pre 年 > 1990;

end 平成生まれと令和生まれ
```

拡張後のBWDMに適用した際の出力の一部

```
関数名 : 生まれ判定
引数の型 : 年:nat 月:nat1
戻り値の型 : seq of (char)
生成テストケース数 : 77件(境界値分析:72/記号実行:5)

各引数の境界値
年 : 4294967295 4294967294 0 -1 1991 1990 2019 2020
月 : 4294967296 4294967295 1 0 12 13 3 4 5

境界値分析によるテストケース
No.1 : 4294967295 4294967296 -> Undefined Action
No.2 : 4294967294 4294967296 -> Undefined Action
=== 中略 ===
No.33 : 4294967295 12 -> Undefined Action
No.34 : 4294967294 12 -> "令和の遅生まれ"
No.35 : 0 12 -> Undefined Action
No.36 : -1 12 -> Undefined Action
No.37 : 1991 12 -> "令和の遅生まれ"
No.38 : 1990 12 -> Undefined Action
No.39 : 2019 12 -> "令和の遅生まれ"
No.40 : 2020 12 -> "令和の遅生まれ"
No.41 : 4294967295 13 -> Undefined Action
No.42 : 4294967294 13 -> Undefined Action
No.43 : 0 13 -> Undefined Action
No.44 : -1 13 -> Undefined Action
No.45 : 1991 13 -> Undefined Action
No.46 : 1990 13 -> Undefined Action
No.47 : 2019 13 -> Undefined Action
No.48 : 2020 13 -> Undefined Action
No.49 : 4294967295 3 -> Undefined Action
No.50 : 4294967294 3 -> "令和の早生まれ"
=== 以下省略 ===
```

拡張(ii):適用例

VDM++仕様

```
1: class 平成生まれと令和生まれ
2:
3: types -- 型定義ブロック
4:     public「年」= nat;
5:
6:     public「月」= nat1
7:     inv m == m <= 12;
8:
9: functions -- 関数定義ブロック
10:    生まれ判定 :「年」*「月」-> seq of char
11:        生まれ判定(年, 月) ==
12:            if(年 <= 2019) then
13:                if(月 < 4) then
14:                    "平成の早生まれ"
15:                else
16:                    if(月 > 4) then
17:                        "令和の遅生まれ"
18:                    else
19:                        "平成の遅生まれ"
20:            else
21:                if(月 < 4) then
22:                    "令和の早生まれ"
23:                else
24:                    "令和の遅生まれ"
25:    pre 年 > 1990;
26:
27: end 平成生まれと令和生まれ
```

「年」型と「月」型がそれぞれ
nat型，nat1型だと特定できている

拡張後のBWDMに適用した際の出力の一部

```
関数名 : 生まれ判定
引数の型 : 年:nat 月:nat1
戻り値の型 : seq of (char)
生成テストケース数 : 77件(境界値分析:72/記号実行:5)

各引数の境界値
年 : 4294967295 4294967294 0 -1 1991 1990 2019 2020
月 : 4294967296 4294967295 1 0 12 13 3 4 5

境界値分析によるテストケース
No.1 : 4294967295 4294967296 -> Undefined Action
No.2 : 4294967294 4294967296 -> Undefined Action
=== 中略 ===
No.33 : 4294967295 12 -> Undefined Action
No.34 : 4294967294 12 -> "令和の遅生まれ"
No.35 : 0 12 -> Undefined Action
No.36 : -1 12 -> Undefined Action
No.37 : 1991 12 -> "令和の遅生まれ"
No.38 : 1990 12 -> Undefined Action
No.39 : 2019 12 -> "令和の遅生まれ"
No.40 : 2020 12 -> "令和の遅生まれ"
No.41 : 4294967295 13 -> Undefined Action
No.42 : 4294967294 13 -> Undefined Action
No.43 : 0 13 -> Undefined Action
No.44 : -1 13 -> Undefined Action
No.45 : 1991 13 -> Undefined Action
No.46 : 1990 13 -> Undefined Action
No.47 : 2019 13 -> Undefined Action
No.48 : 2020 13 -> Undefined Action
No.49 : 4294967295 3 -> Undefined Action
No.50 : 4294967294 3 -> "令和の早生まれ"
=== 以下省略 ===
```

拡張(ii):適用例

VDM++仕様

```
1: class 平成生まれと令和生まれ
2:
3: types -- 型定義ブロック
4:     public「年」= nat;
5:
6:     public「月」= nat1
7:     inv m == m <= 12;
8:
9: functions -- 関数定義ブロック
10:    生まれ判定 :「年」*「月」-> seq of char
11:        生まれ判定(年, 月) ==
12:            if(年 <= 2019) then
13:                if(月 < 4) then
14:                    "平成の早生まれ"
15:                else
16:                    if(月 > 4) then
17:                        "令和の遅生まれ"
18:                    else
19:                        "平成の遅生まれ"
20:            else
21:                if(月 < 4) then
22:                    "令和の早生まれ"
23:                else
24:                    "令和の遅生まれ"
25:    pre 年 > 1990;
26:
27: end 平成生まれと令和生まれ
```

に対応したテストケースを生成している

拡張後のBWDMに適用した際の出力の一部

```
関数名 : 生まれ判定
引数の型 : 年:nat 月:nat1
戻り値の型 : seq of (char)
生成テストケース数 : 77件(境界値分析:72/記号実行:5)

各引数の境界値
年 : 4294967295 4294967294 0 -1 1991 1990 2019 2020
月 : 4294967296 4294967295 1 0 12 13 3 4 5

境界値分析によるテストケース
No.1 : 4294967295 4294967296 -> Undefined Action
No.2 : 4294967294 4294967296 -> Undefined Action
=== 中略 ===
No.33 : 4294967295 12 -> Undefined Action
No.34 : 4294967294 12 -> "令和の遅生まれ"
No.35 : 0 12 -> Undefined Action
No.36 : -1 12 -> Undefined Action
No.37 : 1991 12 -> "令和の遅生まれ"
No.38 : 1990 12 -> Undefined Action
No.39 : 2019 12 -> "令和の遅生まれ"
No.40 : 2020 12 -> "令和の遅生まれ"
No.41 : 4294967295 13 -> Undefined Action
No.42 : 4294967294 13 -> Undefined Action
No.43 : 0 13 -> Undefined Action
No.44 : -1 13 -> Undefined Action
No.45 : 1991 13 -> Undefined Action
No.46 : 1990 13 -> Undefined Action
No.47 : 2019 13 -> Undefined Action
No.48 : 2020 13 -> Undefined Action
No.49 : 4294967295 3 -> Undefined Action
No.50 : 4294967294 3 -> "令和の早生まれ"
=== 以下省略 ===
```

拡張(iii)

4つの問題点に対応する**3つ**の拡張

(i) 不変条件と事前条件と事後条件の解析及び評価

→不変条件と事前条件と事後条件に対応していない問題点を解決

→オブジェクトの状態を変更する操作定義のテストケースを生成できない問題点を解決

(ii) 型定義ブロックへの対応

→型定義ブロックに対応していない問題点を解決

(iii)入れ子構造にある関数定義及び操作定義への対応

→入れ子構造にある関数定義及び操作定義に対応していない問題点を解決

拡張(iii):背景と目的

入れ子構造にある関数定義及び操作定義に対応していない問題点

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
 ボーダーラインを取得: () ==> nat
 ボーダーラインを取得 () ==
    if(平均点 < 60) then
       return 平均点
    else
       return 60;

 平均点と比較: nat ==> seq of char
 平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
       return "平均点以上"
    else
       return "平均点未満";

functions
 成績を確認: nat -> seq of char
 成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
       "再テスト"
    else
       平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

- 定義の中で別の関数定義や操作定義を呼び出す場合がある
- if-then-else式の中で関数定義や操作定義を呼び出していると，入力値の取得や期待出力の導出ができない

```
各引数の境界値
テスト点 : 4294967295 4294967294 0 -1

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> Undefined Action
No.3 : 0 -> Undefined Action
No.4 : -1 -> Undefined Action
```

拡張(iii):背景と目的

入れ子構造にある関数定義及び操作定義に対応していない問題点

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
 ボーダーラインを取得: () ==> nat
 ボーダーラインを取得 () ==
    if(平均点 < 60) then
       return 平均点
    else
       return 60;

 平均点と比較: nat ==> seq of char
 平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
       return "平均点以上"
    else
       return "平均点未満";

functions
 成績を確認: nat -> seq of char
 成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
       "再テスト"
    else
       平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

- 定義の中で別の関数定義や操作定義を呼び出す場合がある
- if-then-else式の中で関数定義や操作定義を呼び出していると，入力値の取得や期待出力の導出ができない

```
各引数の境界値
テスト点 : 4294967295 4294967294 0 -1

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> Undefined Action
No.3 : 0 -> Undefined Action
No.4 : -1 -> Undefined Action
```

拡張(iii):背景と目的

入れ子構造にある関数定義及び操作定義に対応していない問題点

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
 ボーダーラインを取得: () ==> nat
 ボーダーラインを取得 () ==
    if(平均点 < 60) then
       return 平均点
    else
       return 60;

 平均点と比較: nat ==> seq of char
 平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
       return "平均点以上"
    else
       return "平均点未満";

functions
 成績を確認: nat -> seq of char
 成績を確認 (テスト点) ==
    if(テスト点 < 60) then
       "再テスト"
    else
       平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

- return文で関数定義や操作定義を呼び出していると，return文の処理をそのまま期待出力として表示してしまう

```
各引数の境界値
テスト点 : 4294967295 4294967294 0 -1 59 60

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> 平均点と比較(テスト点)
No.3 : 0 -> "再テスト"
No.4 : -1 -> Undefined Action
No.5 : 59 -> "再テスト"
No.6 : 60 -> 平均点と比較(テスト点)
```

拡張(iii):手段

入れ子構造にある関数定義及び操作定義への対応

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return "平均点以上"
    else
        return "平均点未満";

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        "再テスト"
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

入れ子構造にある条件式から境界値（入力値）を取得する

1. 対象とする条件式に到達するための条件式を全て抽出する
2. 呼び出している定義の結果を求める条件式を生成する
3. 1.と2.の条件式を前提条件として，対象とする条件式の境界値を求める

拡張(iii):手段

入れ子構造にある関数定義及び操作定義への対応

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return "平均点以上"
    else
        return "平均点未満";

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        "再テスト"
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

入れ子構造にある条件式から境界値（入力値）を取得する

1. 対象とする条件式に到達するための条件式を全て抽出する

   引数の制約条件と事前条件を抽出する

   - テスト点 >= 0
   - テスト点 <= 4294967294
   - テスト点 <= 100

拡張(iii):手段

入れ子構造にある関数定義及び操作定義への対応

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return "平均点以上"
    else
        return "平均点未満";

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        "再テスト"
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

入れ子構造にある条件式から境界値（入力値）を取得する

2. 呼び出している定義の結果を求める条件式を生成する

呼び出している定義の結果と，そこに到達するための条件式を結合する

- ボーダーラインを取得() = 70 and 70 < 60
- ボーダーラインを取得() = 60 and 70 >= 60

平均点

複数ある場合は，orで結合する

(ボーダーラインを取得() = 70 and 70 < 60) or
(ボーダーラインを取得() = 60 and 70 >= 60)

拡張(iii):手段

入れ子構造にある関数定義及び操作定義への対応

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return "平均点以上"
    else
        return "平均点未満";

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        "再テスト"
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

入れ子構造にある条件式から境界値（入力値）を取得する

3. 1.と2.の条件式を前提条件として，対象とする条件式の境界値を求める

1.と2.で取得した前提条件

- テスト点 <= 100
- テスト点 >= 0
- テスト点 <= 4294967294
- (ボーダーラインを取得() = 70 and 70 < 60) or (ボーダーラインを取得() = 60 and 70 >= 60)

前提条件よりボーダーラインを取得()の結果が60となるため，テスト点の境界値59と60が求められる

拡張(iii):手段

入れ子構造にある関数定義及び操作定義への対応

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return "平均点以上"
    else
        return "平均点未満";

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        "再テスト"
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

入れ子構造にある定義の期待出力を決定する

if-then-else式中に関数定義や操作定義が呼び出されている場合，その定義の木構造に入力値を適用して，実行フローを特定する

特定した実行フローのreturn文で関数定義もしくは操作定義が呼び出されている場合，その定義の木構造を基に，期待出力を決定する

拡張(iii):適用例

VDM++仕様

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
 ボーダーラインを取得: () ==> nat
 ボーダーラインを取得 () ==
    if(平均点 < 60) then
       return 平均点
    else
       return 60;

 平均点と比較: nat ==> seq of char
 平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
       return “平均点以上”
    else
       return “平均点未満”;

functions
 成績を確認: nat -> seq of char
 成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
       “再テスト”
    else
       平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

拡張後のBWDMに適用した際の出力

```
関数名 : 成績を確認
引数の型 : テスト点:nat
戻り値の型 : seq of (char)

各引数の境界値
テスト点 : 4294967295 4294967294 0 -1 100 101 59 60 70 69

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> Undefined Action
No.3 : 0 -> "再テスト"
No.4 : -1 -> Undefined Action
No.5 : 100 -> "平均点以上"
No.6 : 101 -> Undefined Action
No.7 : 59 -> "再テスト"
No.8 : 60 -> "平均点未満"
No.9 : 70 -> "平均点以上"
No.10 : 69 -> "平均点未満"
```

拡張(iii):適用例

VDM++仕様

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return “平均点以上”
    else
        return “平均点未満”;

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        “再テスト”
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

拡張後のBWDMに適用した際の出力

```
関数名 : 成績を確認
引数の型 : テスト点:nat
戻り値の型 : seq of (char)

各引数の境界値
テスト点 : 4294967295 4294967294 0 -1 100 101 59 60 70 69

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> Undefined Action
No.3 : 0 -> "再テスト"
No.4 : -1 -> Undefined Action
No.5 : 100 -> "平均点以上"
No.6 : 101 -> Undefined Action
No.7 : 59 -> "再テスト"
No.8 : 60 -> "平均点未満"
No.9 : 70 -> "平均点以上"
No.10 : 69 -> "平均点未満"
```

入れ子構造にある条件式から境界値（入力値）を取得している

拡張(iii):適用例

VDM++仕様

```
class 成績確認

instance variables
 private 平均点 : nat := 70;

operations
ボーダーラインを取得: () ==> nat
ボーダーラインを取得 () ==
    if(平均点 < 60) then
        return 平均点
    else
        return 60;

平均点と比較: nat ==> seq of char
平均点と比較 (テスト点) ==
    if(テスト点 >= 平均点) then
        return “平均点以上”
    else
        return “平均点未満”;

functions
成績を確認: nat -> seq of char
成績を確認 (テスト点) ==
    if(テスト点 < ボーダーラインを取得 ()) then
        “再テスト”
    else
        平均点と比較 (テスト点)
    pre テスト点 <= 100;

end 成績確認
```

拡張後のBWDMに適用した際の出力

```
関数名 : 成績を確認
引数の型 : テスト点:nat
戻り値の型 : seq of (char)

各引数の境界値
テスト点 : 4294967295 4294967294 0 -1 100 101 59 60 70 69

>>境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> Undefined Action
No.3 : 0 -> "再テスト"
No.4 : -1 -> Undefined Action
No.5 : 100 -> "平均点以上"
No.6 : 101 -> Undefined Action
No.7 : 59 -> "再テスト"
No.8 : 60 -> "平均点未満"
No.9 : 70 -> "平均点以上"
No.10 : 69 -> "平均点未満"
```

呼び出し先の操作定義の出力を期待出力としている

考察:人手によるテストケース作成との比較

入れ子構造にある定義を含む
VDM++仕様書

```
1: class 成績確認
2:
3: instance variables
4:  private 平均点 : nat := 70;
5:
6: operations
7:  ボーダーラインを取得: () ==> nat
8:  ボーダーラインを取得 () ==
9:      if(平均点 < 60) then
10:         return 平均点
11:     else
12:         return 60;
13:
14: 平均点と比較: nat ==> seq of char
15: 平均点と比較 (テスト点) ==
```

人手による作成

テストケース

Boundary Value BWDM Vienna Development Method

拡張後のBWDMによる生成

テストケース

拡張後のBWDMに新たに適用可能となったVDM++仕様から
人手によってテストケースを作成する時間と
拡張後のBWDMに適用してテストケースを生成する時間を比較する

考察:人手によるテストケース作成との比較

実験結果

人手によるテストケース作成時間

被験者	作成時間
被験者A	12m 38s
被験者B	10m 54s
被験者C	8m 33s
被験者D	7m 04s
被験者E	8m 50s
被験者F	8m 02s
被験者G	7m 43s
被験者H	9m 59s

人手とBWDMの比較

	時間
被験者8人の平均	**9m 12s**
BWDM	**1.3s**

人手によるテストケース作成の平均時間に比べて，拡張後のBWDMを使用した場合は，9分程度（約99.76%）の時間短縮が確認できた

また，人手による作成ではヒューマンエラーも見られた

考察:関連研究との比較

Aamer Nadeem 氏らの研究*

- VDM++仕様を対象にしたテストケース自動生成手法の提案
- 不変条件と事前条件を同値分割して，有効な入力空間からランダムに決定した代表値を入力データとする
- メソッド呼び出しの有効な手順を記述した仕様を用意して，テストシーケンスを生成する

BWDMはVDM++仕様のみを用意すればテストケースを生成できる．また，BWDMで生成するテストケースは，境界値テスト，ドメイン分析テストで使用でき，記号実行により境界値分析ではカバーできない実行フローに対するテストも期待できる．さらにオブジェクトに影響を及ぼすメソッドに対して，オブジェクトの状態のテストを実施できる．

*Aamer Nadeem, Muhammad Jaffar-Ur-Rehman. Automated Test Case Generation from IFAD VDM++ Specifications. SEPADS 05: 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems, No.28, pp.1-7, 2005.

まとめ

■ 目的

BWDMの実用性の向上

■ 手段

- 不変条件と事前条件と事後条件の解析及び評価
- 型定義ブロックへの対応
- 入れ子構造にある関数定義及び操作定義への対応

■ 結論

BWDMに3つの拡張を施したことにより，既存の4つの問題点を解決し，適用範囲が拡大した

また，BWDMに新たに適用可能となったVDM++仕様においても，テストケース生成に要する時間を9分程度（約99.76%）短縮でき，ヒューマンエラーを除去できることを確認できた

したがって，

拡張後のBWDMの実用性が向上し，

BWDMを使用した場合のテスト工程の作業効率が向上した

今後の課題

- **整数型以外の型への対応**

  拡張後のBWDMでは，int型とnat型とnat1型のみに対応可能であり，それ以外のreal型やrat型などの型に対応していないため，対応が必要である

- **再帰構造にある関数定義及び操作定義への対応**

  拡張後のBWDMでは，操作及び関数呼び出しが含まれる定義のテストケースを生成する際に，呼び出し先の定義の抽象構文木解析処理を終えていない場合，テストケースを生成できず，再帰構造にある関数及び操作定義のテストケースを生成できため，対応が必要である

- ~~**操作後の値を参照する条件式への対応**~~

  ~~拡張後のBWDMでは，操作後の値を参照する条件式に対する境界値を取得することができないため，対応する必要がある~~

  ☝ **現在は拡張(ｉ)のオブジェクトの状態に対応したテストケース生成により対応済み**