運賃計算システム VDM++ 仕様

佐原　伸

法政大学
情報科学研究科

概要

レコードと列を使った要求仕様の例。
距離から運賃を求めるようにしている。
Makefile を使って、make コマンドで、モデルの検証とドキュメントの PDF ファイル生成を一挙に行うようにしている。

目次

1 **クラス図** 2

2 CRUD_SEQ 4

4 **運賃表辞書** 7
4.1 運賃表 (型) 7

4 **運賃表辞書** 7
4.1 運賃表 (型) 7

5 **路線網** 9
5.1 路線単位レコード 9
5.2 関数 9

6 **路線検索** 11

7 **ダイクストラ算法による路線検索** 13
7.1 最短経路 13

8 **ダイクストラ算法** 14

9 **回帰テストケース** 17
9.1 責任 17

10 参考文献、索引 20

1 クラス図

VDM++ 仕様のクラス図は以下の通りである。

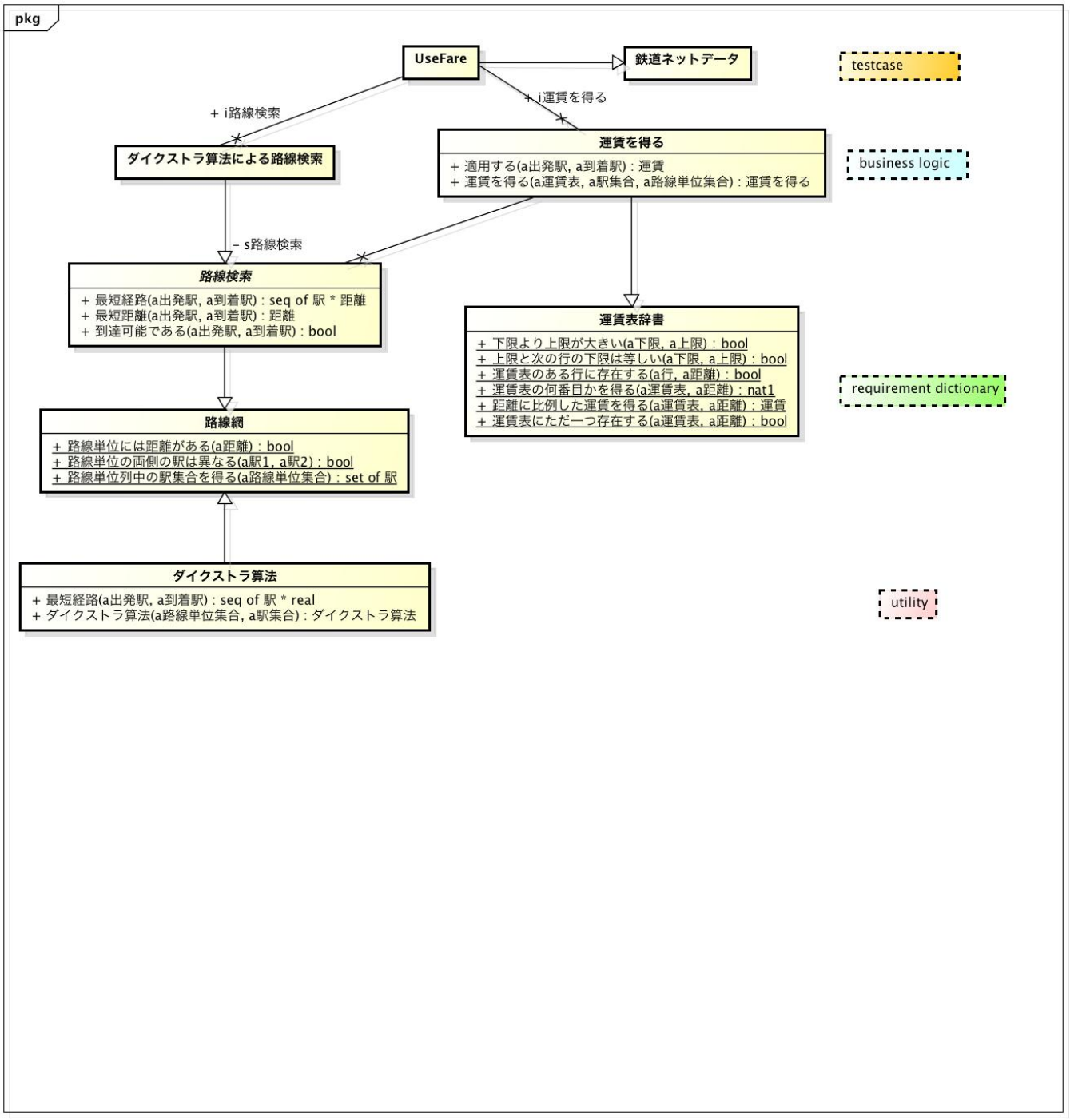

図 1　Class Diagram

2 CRUD_SEQ

集合の CRUD 関数ライブラリー。

```
class
CRUD_SEQ
functions
public
  CreateSeqH[@Elem] : @Elem * seq of @Elem -> seq of @Elem
  CreateSeqH(e,s) ==
    [e]^s
  post RESULT = [e]^s
end
CRUD_SEQ
```

3 運賃表辞書

運賃表の用語 (名詞と述語) を定義する要求辞書 (Requirement Dictionary) である。

```
class
運賃表辞書
types
  public 運賃 = nat;
public
  行 :: f 下限 : 路線網 ‘距離
       f 上限 : 路線網 ‘距離
       f 運賃 :- 運賃;
```

3.1 運賃表 (型)

運賃表の各行の上限と下限は重複しないこと。

```
  public 運賃表 = seq of 行
  inv w 運賃表 ==
       forall i, j in set inds w 運賃表 &
           下限より上限が大きい (w 運賃表 (i).f 下限, w 運賃表 (i).f 上限) and
           j = i + 1 =>
           上限と次の行の下限は等しい (w 運賃表 (i).f 上限, w 運賃表 (j).f 下限)
functions
public static
  距離に比例した運賃を得る : 運賃表 * 路線網 ‘距離 -> 運賃
  距離に比例した運賃を得る (a 運賃表, a 距離) ==
    let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
    a 運賃表 (n).f 運賃
  pre  運賃表にただ一つ存在する (a 運賃表, a 距離)
  post let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
       RESULT = a 運賃表 (n).f 運賃 ;
public static
  運賃表のある行に存在する : 行 * 路線網 ‘距離 +> bool
  運賃表のある行に存在する (a 行, a 距離) ==
    a 行.f 下限 <= a 距離 and a 距離 < a 行.f 上限;
public static
```

```
  運賃表にただ一つ存在する : 運賃表 * 路線網 `距離 -> bool
  運賃表にただ一つ存在する (a 運賃表, a 距離) ==
    exists1 i in set inds a 運賃表 & 運賃表のある行に存在する (a 運賃表 (i), a 距離);
public static
  運賃表の何番目かを得る : 運賃表 * 路線網 `距離 -> nat1
  運賃表の何番目かを得る (a 運賃表, a 距離) ==
    let  i in set inds a 運賃表 be st  運賃表のある行に存在する (a 運賃表 (i), a 距離) in
    i;
public static
  下限より上限が大きい : 運賃 * 運賃 -> bool
  下限より上限が大きい (a 下限, a 上限) ==
    a 下限 < a 上限;
public static
  上限と次の行の下限は等しい : 運賃 * 運賃 -> bool
  上限と次の行の下限は等しい (a 下限, a 上限) ==
    a 下限 = a 上限
end
```

運賃表辞書

Test Suite : vdm.tc

Class : 運賃表辞書

Name	#Calls	Coverage
運賃表辞書 `下限より上限が大きい	0	0%
運賃表辞書 `運賃表の何番目かを得る	0	0%
運賃表辞書 `距離に比例した運賃を得る	0	0%
運賃表辞書 `運賃表にただ一つ存在する	0	0%
運賃表辞書 `運賃表のある行に存在する	0	0%
運賃表辞書 `上限と次の行の下限は等しい	0	0%
Total Coverage		**0%**

4 運賃表辞書

運賃表の用語 (名詞と述語) を定義する要求辞書 (Requirement Dictionary) である。

```
class
運賃表辞書
types
  public 運賃 = nat;
public
  行 :: f 下限 : 路線網 ‘距離
       f 上限 : 路線網 ‘距離
       f 運賃 :- 運賃;
```

4.1 運賃表 (型)

運賃表の各行の上限と下限は重複しないこと。

```
  public 運賃表 = seq of 行
  inv w 運賃表 ==
       forall i, j in set inds w 運賃表 &
           下限より上限が大きい (w 運賃表 (i).f 下限, w 運賃表 (i).f 上限) and
           j = i + 1 =>
           上限と次の行の下限は等しい (w 運賃表 (i).f 上限, w 運賃表 (j).f 下限)
functions
public static
  距離に比例した運賃を得る : 運賃表 * 路線網 ‘距離 -> 運賃
  距離に比例した運賃を得る (a 運賃表, a 距離) ==
    let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
    a 運賃表 (n).f 運賃
  pre  運賃表にただ一つ存在する (a 運賃表, a 距離)
  post let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
       RESULT = a 運賃表 (n).f 運賃 ;
public static
  運賃表のある行に存在する : 行 * 路線網 ‘距離 +> bool
  運賃表のある行に存在する (a 行, a 距離) ==
    a 行.f 下限 <= a 距離 and a 距離 < a 行.f 上限;
public static
```

```
  運賃表にただ一つ存在する : 運賃表 * 路線網 `距離 -> bool
  運賃表にただ一つ存在する (a 運賃表,a 距離) ==
     exists1 i in set inds a 運賃表 & 運賃表のある行に存在する (a 運賃表 (i),a 距離);
public static
  運賃表の何番目かを得る : 運賃表 * 路線網 `距離 -> nat1
  運賃表の何番目かを得る (a 運賃表,a 距離) ==
    let  i in set inds a 運賃表 be st  運賃表のある行に存在する (a 運賃表 (i),a 距離) in
    i;
public static
  下限より上限が大きい : 運賃 * 運賃 -> bool
  下限より上限が大きい (a 下限,a 上限) ==
    a 下限 < a 上限;
public static
  上限と次の行の下限は等しい : 運賃 * 運賃 -> bool
  上限と次の行の下限は等しい (a 下限,a 上限) ==
    a 下限 = a 上限
end
```

運賃表辞書

Test Suite : vdm.tc

Class : 運賃表辞書

Name	#Calls	Coverage
運賃表辞書 `下限より上限が大きい	0	0%
運賃表辞書 `運賃表の何番目かを得る	0	0%
運賃表辞書 `距離に比例した運賃を得る	0	0%
運賃表辞書 `運賃表にただ一つ存在する	0	0%
運賃表辞書 `運賃表のある行に存在する	0	0%
運賃表辞書 `上限と次の行の下限は等しい	0	0%
Total Coverage		**0%**

5 路線網

運賃計算に限らず使用することのできる、路線網のドメイン・モデルである。

```
class
路線網
types
  public 駅 = token;
  public 距離 = real;
  public 駅集合 = set of 駅
  inv w駅集合 == card w駅集合 >= 2;
```

5.1 路線単位レコード

路線単位レコードは、両端の駅が同一にも関わらず複数の距離があることは考慮していない。

```
public
  路線単位 :: f駅1 : 駅
              f駅2 : 駅
              f距離 : 距離
  inv w路線単位 == 路線単位の両側の駅は異なる(w路線単位.f駅1, w路線単位.f駅2) and 路線単位には距離がある(w路線単位.f距離);
  public 路線単位集合 = set of 路線単位
  inv w路線単位集合 == w路線単位集合 <> {}
```

5.2 関数

```
functions
public static
  路線単位の両側の駅は異なる : 駅 * 駅 -> bool
  路線単位の両側の駅は異なる(a駅1, a駅2) ==
    a駅1 <> a駅2;
public static
  路線単位には距離がある : 距離 -> bool
  路線単位には距離がある(a距離) ==
    a距離 > 0;
public static
```

```
  路線単位列中の駅集合を得る : 路線単位集合 -> set of 駅
  路線単位列中の駅集合を得る(a 路線単位集合) ==
    dunion {{w 路線単位.f 駅 1, w 路線単位.f 駅 2} | w 路線単位 in set a 路線単位集合 }
end
```

路線網

Test Suite : vdm.tc

Class : 路線網

Name	#Calls	Coverage
路線網 ‘路線単位には距離がある	24	√
路線網 ‘路線単位の両側の駅は異なる	24	√
路線網 ‘路線単位列中の駅集合を得る	0	0%
Total Coverage		**42%**

6 路線検索

路線検索の抽象クラスである。

```
class
路線検索 is subclass of 路線網
types
public
  経路条件 :: 発駅 : 駅
              着駅 : 駅
  inv w 経路条件 ==
      w 経路条件.発駅 <> w 経路条件.着駅
instance variables
  protected s 駅集合 : 駅集合;
  protected s 路線単位集合 : 路線単位集合;
  inv let w 路線単位集合中の駅集合 = 路線単位列中の駅集合を得る (s 路線単位集合) in
      w 路線単位集合中の駅集合 subset s 駅集合

operations
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * 距離
  最短経路 (a 出発駅, a 到着駅) ==
    is subclass responsibility;
public
  最短距離 : 駅 * 駅 ==> 距離
  最短距離 (a 出発駅, a 到着駅) ==
    (   def mk_(-, w 最短距離) = 最短経路 (a 出発駅, a 到着駅) in
        if w 最短距離 > 0
        then return w 最短距離
        else exit <到達不可能である>
    )
end
路線検索
```

Test Suite : vdm.tc

Class : 路線検索

Name	#Calls	Coverage
路線検索 ‘最短経路	0	0%

Name	#Calls	Coverage
路線検索 ‘最短距離	0	0%
Total Coverage		**0%**

7 ダイクストラ算法による路線検索

ダイクストラ算法にによって最短経路を求める。

```
class
ダイクストラ算法による路線検索 is subclass of 路線検索
operations
public
  ダイクストラ算法による路線検索 : 駅集合 * 路線単位集合 ==> ダイクストラ算法による路線検索
  ダイクストラ算法による路線検索 (a 駅集合, a 路線単位集合) == atomic
    (  s 駅集合 := a 駅集合;
       s 路線単位集合 := a 路線単位集合
    );
```

7.1 最短経路

ダイクストラ算法にによって、最短経路の路線単位列を求め、経路に変換する。
経路が繋がっていない場合は空列を返す。

```
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * 距離
  最短経路 (a 出発駅, a 到着駅) ==
    (   dcl w 最短距離アルゴリズム : ダイクストラ算法 := new ダイクストラ算法 (s 路線単位集合
,s 駅集合);
        return w 最短距離アルゴリズム.最短経路 (a 出発駅, a 到着駅)
    )
end
ダイクストラ算法による路線検索
```

Test Suite : vdm.tc
Class : ダイクストラ算法による路線検索

Name	#Calls	Coverage
ダイクストラ算法による路線検索 ‘最短経路	0	0%
ダイクストラ算法による路線検索 ‘ダイクストラ算法による路線検索	0	0%
Total Coverage		**0%**

8 ダイクストラ算法

路線単位の長さが非負である鉄道グラフ（ネットワーク）の上で、ある駅から他の任意の駅への最短経路・最短距離を求める。

「アルゴリズム辞典」のアルゴリズム (p.455) をそのまま使用。ただし、配列は写像を使って実装した。

```
class
ダイクストラ算法 is subclass of 路線網
types
  public 確定 = <未確定> | <既確定>;
  public X 確定 = map 駅 to 確定;
  public V 最短距離 = map 駅 to real;
  public P 直前の駅 = map 駅 to 駅
instance variables
  public s 路線単位集合 : 路線単位集合;
  public s 駅集合 : 駅集合;
  public x : X 確定 := { |-> };
  public v : V 最短距離 := { |-> };
  public p : P 直前の駅 := { |-> };

values

  v 無限大 = 10000000000000000
operations
public
  ダイクストラ算法 : set of 路線単位 * set of 駅 ==> ダイクストラ算法
  ダイクストラ算法 (a 路線単位集合, a 駅集合) ==
    (   s 路線単位集合 := a 路線単位集合;
        s 駅集合 := a 駅集合
    );
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * real
  最短経路 (a 出発駅, a 到着駅) ==
    (   dcl i : 駅 := a 出発駅;
        for all w 駅 in set s 駅集合
        do if w 駅 = a 出発駅
           then (   x(a 出発駅) := <既確定>;
                    v(a 出発駅) := 0
                )
```

```
            else (   x(w 駅) := <未確定>;
                     v(w 駅) := v 無限大
                 ) ;
        for all - in set s 駅集合
        do (   def Ni = {u.f 駅 2 | u in set s 路線単位集合 & u.f 駅 1 = i};
                   Nu = {u | u in set s 路線単位集合 & u.f 駅 1 = i} in
               (   for all j in set Ni
                   do (   if x(j) = <未確定>
                          then def w = v(i) + d(Nu, i, j) in
                               if w < v(j)
                               then (   v(j) := w;
                                        p(j) := i
                                    )
                      ) ;
                   def Ni 未確定 = {e | e in set Ni & x(e) = <未確定>} in
                   if Ni 未確定 <> {}
                   then let   s in set Ni 未確定 be st   forall s1 in set Ni 未確定 & v(s) <= v(s1) in
                        (   i := s;
                            x(i) := <既確定>
                        )
               )
           ) ;
        def w 最短経路 = 経路を作る(p, a 出発駅, a 到着駅);
            w 最短距離 = v(a 到着駅) in
        return mk_(w 最短経路, w 最短距離)
    )
functions
  d : 路線単位集合 * 駅 * 駅 -> real
  d(a 路線単位集合, a 駅 1, a 駅 2) ==
    let   di in set a 路線単位集合 be st   di.f 駅 1 = a 駅 1 and di.f 駅 2 = a 駅 2 in
    di.f 距離
operations
  経路を作る : P 直前の駅 * 駅 * 駅 ==> seq of 駅
  経路を作る(aP 直前の駅, a 出発駅, a 到着駅) ==
    (   dcl w 最短経路 : seq of 駅 := [],
            w 駅 : 駅 := a 到着駅;
        if a 到着駅 not in set dom aP 直前の駅
        then exit <到着駅が直前の駅の中に存在しない>
        else while w 駅 <> a 出発駅
```

```
            do (   w 最短経路 := CRUD_SEQ‘CreateSeqH[駅](w 駅, w 最短経路);
                   w 駅 := aP 直前の駅(w 駅)
               );
        return CRUD_SEQ‘CreateSeqH[駅](a 出発駅, w 最短経路)
    )
end
```

ダイクストラ算法

Test Suite : vdm.tc

Class : ダイクストラ算法

Name	#Calls	Coverage
ダイクストラ算法‘d	0	0%
ダイクストラ算法‘最短経路	0	0%
ダイクストラ算法‘経路を作る	0	0%
ダイクストラ算法‘ダイクストラ算法	0	0%
Total Coverage		**0%**

9 回帰テストケース

9.1 責任

「運賃を得る」クラス、及び関連するクラスをテストする。

```
.....................................................................................
class
TestCaseT is subclass of TestCase, 鉄道ネットデータ
values
public
  v 最大値 = 100000000
instance variables
  public s 路線検索 : 路線検索 := new ダイクストラ算法による路線検索 (v 駅集合, v 路線単位集合);
  public s 運賃を得る : 運賃を得る := new 運賃を得る ([
                                      mk_運賃表辞書 ‘行 (0, 1, 150),
                                      mk_運賃表辞書 ‘行 (1, 3, 160),
                                      mk_運賃表辞書 ‘行 (3, 6, 190),
                                      mk_運賃表辞書 ‘行 (6, 10, 220),
                                      mk_運賃表辞書 ‘行 (10, 15, 250),
                                      mk_運賃表辞書 ‘行 (15, v 最大値, 300)],
                                     v 駅集合,
                                     v 路線単位集合);

operations
public
  print : seq of char ==> ()
  print (s) ==
    let - = new IO ().echo (s) in
    skip
end
TestCaseT
class
TestCaseT0001 is subclass of TestCaseT
operations
public
```

```
  test01 : () ==> ()
  test01() ==
    (   def wDistance =
                s 路線検索.最短距離 (v 東京, v 新宿) in
        assertTrue("\t test01 計算結果が間違っている \n",
             wDistance = 7.7 and
             s 運賃を得る.適用する (v 東京, v 新宿) = 220)
    );
public
  test02 : () ==> ()
  test02() ==
    (   def wDistance =
                s 路線検索.最短距離 (v 四ツ谷, v 品川) in
        assertTrue("\t test02 計算結果が間違っている。\n",
             wDistance = 9.5 and
             s 運賃を得る.適用する (v 四ツ谷, v 品川) = 220)
    );
public
  test03 : () ==> ()
  test03() ==
    (   assertTrue("\t test03 計算結果が間違っている。\n",
             let mk_ (w 最短経路, w 最短距離) = s 路線検索.最短経路 (v 池袋, v 品川) in
             w 最短経路 = [mk_token ("池袋"), mk_token ("新宿"), mk_token ("品川")] and
             w 最短距離 = 14.9)
    );
public
  testE01 : () ==> ()
  testE01() ==
    (   trap <到着駅が直前の駅の中に存在しない>
        with  print("\t testE01 期待した事前条件エラーを検出した。\n") in
        (   def - =
                    s 路線検索.最短距離 (v 東京, v 東京) in
            print("\t testE01 予想外のエラーに遭遇。\n")
        )
    );
public
  testE02 : () ==> ()
  testE02() ==
    (   trap <到着駅が直前の駅の中に存在しない>
        with  print("\t testE02 期待した事前条件エラーを検出した。\n") in
```

```
        (   def - =
                    s 路線検索.最短距離(v 東京, v コペンハーゲン) in
            print("\t testE02 予想外のエラーに遭遇。\n")
        )
    )
end
TestCaseT0001
.....................................................................................................
```

10 参考文献、索引

VDM++[2] は、1970 年代中頃に IBM ウィーン研究所で開発された VDM-SL[1] を拡張し、さらにオブジェクト指向拡張した形式仕様記述言語である。

参考文献

[1] Kyushu University. VDM-SL 言語マニュアル. Kyushu University, 第 2.0 版, 2016. Revised for VDMTools V9.0.2.

[2] Kyushu University. VDMTools VDM++ 言語マニュアル. Kyushu University, 第 2.0 版, 2016. Revised for VDMTools V9.0.2.