

# VDM++関数型回帰テスト支援ライブラリ

佐原伸日本フィッツ株式会社

情報技術研究所

TEL : 03-3623-4683

shin.sahara@jfits.co.jp

2004 年 7 月 22 日 ~ 24 日

## 概要

VDM++基本ライブラリ [5] の一つとして、回帰テスト支援ライブラリをオブジェクト指向によって作成した。しかし、上記開発を通して関数型指向の仕様記述が優れていることが判明したため、回帰テスト支援ライブラリを、関数型プログラミングの技術を適用して再開発した。

本稿では、このライブラリの実装を示し、オブジェクト指向版と比較して利点が大きいことを示す。

## 1 はじめに

証券会社向けパッケージソフトウェア (TradeOne システム) の開発に際し、品質の向上と生産性向上を狙って、形式仕様記述言語 VDM++[2] を使用し、要求仕様を定義した。しかし、VDM++には、簡単な数学ライブラリと基本的なファイル入出力ライブラリがあるだけで、回帰テスト支援ライブラリなどの基本的なライブラリは皆無であり、いくつかのライブラリ群を構築した。

この経験を通して、関数型指向の仕様記述が優れていることが判明したため、回帰テスト支援ライブラリを、関数型プログラミング技術を用いて再開発し、オブジェクト指向版と比較した。

結果として、回帰テストのためのテストケースの記述量が 35.2%減少し、ライブラリ自体の記述量も減少し単純化できた。

以下、第 2 節に VDM++の概要を述べ、3 節でライブラリの実装を示す。第 4 節でオブジェクト指向版との比較を示し、5 節でまとめる。本稿で使用する数学記号の説明は 6 節に示した。

## 2 VDM++

VDM[1] は IBM のウィーン研究所で 1970 年代中頃に開発されたモデルベースの形式手法である。その形式仕様記述言語 VDM-SL[6] は 1966 年に ISO 標準 (ISO/IEC 13817-1) になっている。

VDM++は欧州連合の ESPRIT 計画 AFRODITE プロジェクトで開発された、オブジェクト指向機能を VDM-SL に追加した形式仕様記述言語である。不変条件・事前条件・事後条件・陰仕様記述などの形式手法を支援する機能だけでなく、クラス・多重継承といったオブジェクト指向プログラミング、およびパターン・内包表記・型変数・高階関数などの関数型プログラミング機能<sup>1</sup>も持っている。

今回使用した VDM++は、日本語の識別子と注釈の使用が可能であるが、本稿のように  $\text{\LaTeX}$  で VDM++ソースを書く場合は、清書する際に日本語がうまく表示できないので、VDM++ソースは英語の数学記号表現となっている。

## 3 ライブラリの実装

以下に、回帰テスト支援ライブラリ本体の FTest-Driver クラス<sup>2</sup>と、ログ表示のための FTestLogger クラスを示し、次に、テスト対象となる Set クラスと、そのテストケースを記述した SetT を示す。

<sup>1</sup>残念ながら、遅延評価方式でなく、先行評価方式である。

<sup>2</sup>関数型指向なので、クラスは単にモジュールと考えてよい。

### 3.1 FTestDriver

回帰テストを実行するモジュール。

TestCase 型は、テストケース 1 件を表す。

class FTestDriver

types

public

```
1.0  TestCase :: testCaseName : char*
      .1      testResult :  $\mathbb{B}$ 
```

run は、与えられたテストケース列から結果列を得る。結果がすべて true ならば全体成功メッセージを表示し、1 つでも失敗があれば全体失敗メッセージを表示する。

functions

public static

```
2.0  run : TestCase*  $\rightarrow \mathbb{B}$ 
      .1  run (t)  $\triangleq$ 
      .2    let m = "Result-of-testcases.",
      .3      r = [isOk (t(i)) | i  $\in$  inds t] in
      .4    if  $\forall i \in$  inds r  $\cdot$  r(i)
      .5    then FTestLogger'SuccessAll(m)
      .6    else FTestLogger'FailureAll(m);
```

isOk は、与えられたテストケースのテスト結果を確認し、true ならば成功メッセージを表示し、false ならば失敗メッセージを表示する。

public static

```
3.0  isOK : TestCase  $\rightarrow \mathbb{B}$ 
      .1  isOK (t)  $\triangleq$ 
      .2    if GetTestResult (t)
      .3    then FTestLogger'Success (t)
      .4    else FTestLogger'Failure (t);
```

GetTestResult は、テスト結果を得る。

public static

```
4.0  GetTestResult : TestCase  $\rightarrow \mathbb{B}$ 
      .1  GetTestResult (t)  $\triangleq$ 
      .2    t.testResult;
```

GetTestName は、テスト名を得る。

public static

```
5.0  GetTestName : TestCase  $\rightarrow$  char*
      .1  GetTestName (t)  $\triangleq$ 
      .2    t.testCaseName
```

end FTestDriver

Test Suite : vdm.tc

Class : FTestDriver

Name	#Calls	Coverage
FTestDriver'run	3	86%
FTestDriver'isOk	34	70%
FTestDriver'GetTestName	34	✓
FTestDriver'GetTestResult	34	✓
<b>Total Coverage</b>		<b>83%</b>

### 3.2 FTestLogger

テストのログを管理する関数を提供する。

class *FTestLogger*  
values

6.0 *historyFileName* = "VDMTESTLOG.TXT";

7.0 *io* = new *IO*()

*Success* は、成功メッセージをファイルに追加し、標準出力に表示し、true を返す。

functions

public static

8.0 *Success* : *FTestDriver*‘*TestCase* →  $\mathbb{B}$

.1 *Success* (*t*)  $\triangle$

.2 let *message* =

.3 *FTestDriver*‘*GetTestName* (*t*) $\curvearrowright$   
"tOK.\n",

.4 - = *io.fecho* (*historyFileName*, *message*, APPEND),

.5 - = *io.echo* (*message*) in

.6 true;

*Failure* は、失敗メッセージをファイルに追加し、標準出力に表示し、false を返す。

public static

9.0 *Failure* : *FTestDriver*‘*TestCase* →  $\mathbb{B}$

.1 *Failure* (*t*)  $\triangle$

.2 let *message* = *FTestDriver*‘*GetTestName* (*t*) $\curvearrowright$   
"tNG.\n",

.3 - = *io.fecho* (*historyFileName*, *message*, APPEND),

.4 - = *io.echo* (*message*) in

.5 false;

*SuccessAll* は、全体成功メッセージをファイルに追加し、標準出力に表示し、true を返す。

public static

10.0 *SuccessAll* :  $\text{char}^* \rightarrow \mathbb{B}$

.1 *SuccessAll* (*m*)  $\triangle$

.2 let *message* = *m*  $\curvearrowright$  "tOK!!\n",

.3 - = *io.fecho* (*historyFileName*, *message*, APPEND),

.4 - = *io.echo* (*message*) in

.5 true;

*FailureAll* は、全体失敗メッセージをファイルに追加し、標準出力に表示し、false を返す。

public static

11.0 *FailureAll* :  $\text{char}^* \rightarrow \mathbb{B}$

.1 *FailureAll* (*m*)  $\triangle$

.2 let *message* = *m*  $\curvearrowright$  "tNG!!\n",

.3 - = *io.fecho* (*historyFileName*, *message*, APPEND),

.4 - = *io.echo* (*message*) in

.5 false

end *FTestLogger*

**Test Suite :** vdm.tc

**Class :** FTestLogger

Name	#Calls	Coverage
FTestLogger‘Failure	0	0%
FTestLogger‘Success	34	✓
FTestLogger‘FailureAll	0	0%
FTestLogger‘SuccessAll	3	✓
<b>Total Coverage</b>		<b>50%</b>

### 3.3 FSet

集合に関わる関数を提供する。集合演算であらかじめ定義された機能以外の機能を定義する。

class *FSet*

AsSequence は、集合からシーケンスに変換する。

functions

public static

```
12.0  AsSequence[@T] : @T-set → @T*
.1    AsSequence(aSet)  $\triangleq$ 
.2      if aSet = {}
.3      then []
.4      else let s  $\in$  aSet in
.5        [s]  $\curvearrowright$  AsSequence[@T](aSet \ {s})
post HasSameElems[@T](RESULT, aSet);
```

HasSameElems は、列が集合の要素を過不足無く含む事を表す。

public static

```
13.0  HasSameElems[@T] : (@T*)  $\times$  (@T-set)
→  $\mathbb{B}$ 
.1    HasSameElems(s, aSet)  $\triangleq$ 
.2      (elems s = aSet)  $\wedge$  (len s = card aSet);
```

Combinations は、集合から要素数 n 個の組み合わせを得る。

public static

```
14.0  Combinations[@T] :  $\mathbb{N}_1 \rightarrow @T\text{-set} \rightarrow @T\text{-set-set}$ 
.1    Combinations(n)(aSet)  $\triangleq$ 
.2      {e | e  $\in \mathcal{F}$  aSet  $\cdot$  card e = n};
```

Fmap は、関数 f を集合に適用した結果の集合を返す。本関数は、通常の間数型言語の map 関数と同等であるが、map は VDM++ では予約語のため、Fmap と命名した。

public static

```
15.0  Fmap[@T1, @T2] : (@T1 → @T2) → @T1-set → @T2-set
.1    Fmap(f)(aSet)  $\triangleq$ 
.2      {f(s) | s  $\in$  aSet}
```

end *FSet*

Test Suite : vdm.tc

Class : FSet

Name	#Calls	Coverage
FSet.Fmap	3	✓
FSet.AsSequence	5	72%
FSet.Combinations	6	✓
FSet.HasSameElems	1	✓
<b>Total Coverage</b>		<b>87%</b>

### 3.4 FSetT

FSet のテストを行う。testcase2 は、回帰テスト支援ライブラリのテストのため、意図的に false になるようにしている。

class *FSetT* is subclass of *FSet*

functions

public static

```
16.0  run : () → ℤ
      .1  run () ≜
      .2    let testcases = [t1 (), t2 (), t3 (), t4 ()] in
      .3    FTestDriver.run (testcases);
```

#### 3.4.1 HasSameElems を検査する

```
17.0  t1 : () → FTestDriver.TestCase
      .1  t1 () ≜
      .2    mk-FTestDriver.TestCase
      .3    (
      .4      "FSetT01 : \t2HasSameElems3092691C67FB3059308B",
      .5      HasSameElems[ℤ] (FSet.AsSequence[ℤ] ({1, 2, 3, 4}), {1, 2, 3, 4}));
```

#### 3.4.2 Combinations を検査する

```
18.0  t2 : () → FTestDriver.TestCase
      .1  t2 () ≜
      .2    mk-FTestDriver.TestCase
      .3    (
      .4      "FSetT02 : \t2Combinations3092691C67FB3059308B",
      .5      Combinations[ℤ] (2) ({1, 2, 3}) = {{1, 2}, {1, 3}, {2, 3}} ∧
      .6      Combinations[ℤ] (2) ({1, 2, 3, 4}) =
      .7      {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}} ∧
      .8      Fmap[ℤ-set, ℤ-set-set] (Combinations[ℤ] (2)) ({1, 2, 3}, {1, 2, 3, 4}) =
      .9      {{1, 2}, {1, 3}, {2, 3}}, {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}} ∧
      .10     Combinations[ℤ] (3) ({1, 2, 3, 4}) = {{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}} ∧
      .11     Combinations[char*] (2) ({"Sahara", "Sato", "Sako", "Yatsu", "Nishikawa"}) =
      .12     {{"Sahara", "Sato"}, {"Sahara", "Nishikawa"}, {"Sahara", "Yatsu"},
      .13     {"Sahara", "Sako"}, {"Sato", "Nishikawa"}, {"Sato", "Yatsu"},
      .14     {"Sato", "Sako"}, {"Nishikawa", "Yatsu"}, {"Nishikawa", "Sako"},
      .15     {"Yatsu", "Sako"}});
```

### 3.4.3 Fmap を検査する

```
19.0  t3 : () → FTestDriver `TestCase
.1    t3 ()  $\triangle$ 
.2      mk-FTestDriver `TestCase
.3      (
.4        "FSetT03 : \t2Fmap3092691C67FB3059308B",
.5        Fmap[ℤ, ℤ] (λ x : ℤ · x mod 3) ({1, 2, 3, 4, 5}) = {0, 1, 2});
```

### 3.4.4 Fmap のエラーケースを検査する

```
20.0  t4 : () → FTestDriver `TestCase
.1    t4 ()  $\triangle$ 
.2      mk-FTestDriver `TestCase
.3      (
.4        "FSetT04 : \t2Fmap306E30A830E930FC30B130FC30B93092691C67FB3059308B",
.5        (FSet `Fmap[ℤ, ℤ] (λ x : ℤ · x mod 3) ({1, 2, 3, 4, 5}) = {0, 1}) = false)
```

end FSetT

以下にオブジェクト指向版の SetT クラスの一部を示す。

テストグループ

--Set のテスト

```
class SetT is subclass of TestDriver
```

```
functions
```

```
tests : () -> seq of TestCase
```

```
tests () ==
```

```
[
```

```
    new SetT01(),
```

```
    new SetT02(),
```

```
    new SetT03()
```

```
];
```

```
end SetT
```

-----  
--列との比較と、列への変換。

```
class SetT01 is subclass of TestCase
```

```
operations
```

```
protected test: () ==> bool
```

```
test() ==
```

```
    return
```

```
        Set'hasSameElems[int](Set'asSequence[int]({1,2,3,4}),{1,2,3,4}) ;
```

```
end SetT01
```

-----  
--組み合わせを得る。

```
class SetT02 is subclass of TestCase
```

```
operations
```

```
protected test: () ==> bool
```

```
test() ==
```

```
    return
```

```
        Set'Combinations[int](2)({1,2,3}) = {{1,2},{1,3},{2,3}} and
```

```
        Set'Combinations[int](2)({1,2,3,4}) = {{1,2},{1,3},{1,4},{2,3},{2,4},{3,4}} and
```

```
        Set'fmap[set of int, set of set of int](Set'Combinations[int](2))({{1,2,3},{1,2,3,4}}) =
```

```
        {{{1,2},{1,3},{2,3}},{1,2},{1,3},{1,4},{2,3},{2,4},{3,4}} and
```

```
        Set'Combinations[int](3)({1,2,3,4}) = {{1,2,3},{1,2,4},{1,3,4},{2,3,4}} and
```

```
        Set'Combinations[seq of char](2)({"佐原","佐藤","酒匂","谷津","西川"})=
```

```
        {"佐原","佐藤"}, {"佐原","西川"}, {"佐原","谷津"}, {"佐原","酒匂"}, {"佐藤","西川"},
```

```
        {"佐藤","谷津"}, {"佐藤","酒匂"}, {"西川","谷津"}, {"西川","酒匂"}, {"谷津","酒匂"} ;
```

```
end SetT02
```

## 4 関数型ライブラリとオブジェクト指向ライブラリの比較

回帰テスト支援ライブラリのオブジェクト指向版と、関数型指向の本ライブラリを比較した。下表は、両者の行数を比較したものである。

表 4. 関数型ライブラリとオブジェクト指向 (OO) ライブラリの行数 (DSI)<sup>3</sup>比較

モジュール	関数型	OO	比率
回帰テスト支援ライブラリ	61	71	85.9%
テストケース (FSetT, FSequenceT)	289	546	53.9%

回帰テスト支援ライブラリについての上記の差は、オブジェクト指向版が `TestCase` をクラスとしているのに対し、関数型指向版はレコード型で定義していることにより、クラス定義の記述量分、オブジェクト指向版の方が行数が多い。

テストケースの場合は、オブジェクト指向版がテストケース毎に 1 クラス定義しなければならないのに対し、関数型指向版は 1 レコード定義すればよいので 2 行で済む、という差が出ている。

TradeOne システムの開発に際しては、約 3 万行のテストケースを記述したが、上記比率を適用すると約 1 万 6 千行で済み、1 万 4 千行削減できることになる。

また、関数型指向版の方が、各関数の依存関係が少なく、理解しやすく、テストケースも作成しやすい。

## 5 まとめ

VDM++ の回帰テスト支援ライブラリを、関数型指向で簡単に実装できることを示し、オブジェクト指向版と比較して、特にテストケースの記述量がかなり削減できることを示した。

なお、本稿自体が回帰テスト支援ライブラリおよびテストケースのソースコードとなっていて、VDM++ インタープリタによって実行し、コードカバレッジを計測した結果を清書したものになっていることを付記しておく。

<sup>3</sup>DSI は Distributed Software Instruction の略。注釈を除くソースコード中の命令数。本ライブラリの場合、ほぼ 1 命令 = 1 行なので、行数に比例すると考えてよい。

## 6 数学記号とその意味

表 5. 数学記号とその意味

数学記号	ASCII 表現	説明
$\cdot$	<code>&amp;</code>	束縛と式の区切り記号
$\times$	<code>*</code>	積
$\leq$	<code>&lt;=</code>	以下
$\geq$	<code>&gt;=</code>	以上
$\neq$	<code>&lt;&gt;</code>	不等
$\rightarrow$	<code>-&gt;</code>	関数の対応を示す記号
$\Rightarrow$	<code>=&gt;</code>	含意
$\triangleq$	<code>==</code>	定義
$\dagger$	<code>++</code>	列の更新
$\subseteq$	<code>subset</code>	部分集合
$\sim$	<code>~</code>	列の連接
$\mathcal{F}$	<code>power</code>	べき
$\dots\text{-set}$	<code>set of ...</code>	集合
$\dots*$	<code>seq of ...</code>	列
$\dots^+$	<code>seq1 of ...</code>	要素数 1 以上の列
$\mathbb{B}$	<code>bool</code>	ブール型
$\mathbb{N}$	<code>nat</code>	自然数
$\mathbb{Z}$	<code>int</code>	整数
$\neg$	<code>not</code>	否定
$\cap$	<code>inter</code>	共通部分
$\cup$	<code>union</code>	合併
$\in$	<code>in set</code>	帰属関係
$\notin$	<code>not in set</code>	非帰属関係
$\wedge$	<code>and</code>	連言
$\vee$	<code>or</code>	選言
$\forall$	<code>forall</code>	全称
$\lambda$	<code>lambda</code>	ラムダ

## 7 謝辞

オブジェクト指向版の回帰テスト支援ライブラリは、最初に酒匂寛氏が作成し、筆者が一部修正した。また、本稿の記述に用いた  $\text{\LaTeX}$  は、小川弘和氏の `pTeX(sjis)+JMacros for MacOSX Installer Package` を使用した。ここに謝意を表する。



## 参考文献

- [1] Cliff Jones : Systematic Software Development using VDM, Prentice Hall International(1990)
- [2] IFAD : The IFAD VDM++ Language V6.8, IFAD(2001)
- [3] 佐原伸 : 事務システムにおける形式仕様適用例, ソフトウェア技術者協会, ソフトウェア・シンポジウム (2001)
- [4] 佐原伸 : 大規模事務処理システムにおける形式手法の適用経験, ソフトウェア技術者協会, ソフトウェア・シンポジウム (2003)
- [5] 佐原伸 : VDM++ 基本ライブラリの作成, ソフトウェア技術者協会, ソフトウェア・シンポジウム (2004 予定)
- [6] ジョン・フィッツジェラルド, ピーター・ゴーム・ラーセン著, 荒木啓二郎, 張漢明, 荻野隆彦, 佐原伸, 染谷誠 訳 : ソフトウェア開発のモデル化技法, 岩波書店 (2003)