Vol.2015-SE-188 No.3
Vol.2015-EMB-37 No.3
2015/6/4

VDM++ 仕様から自動生成した Java コードの振る舞い検証手法の提案

大森 洋一[1]　林 信宏[1]　日下部 茂[1]　荒木 啓二郎[1]

概要：ソフトウェア開発の効率改善や品質向上の観点から，仕様からのプログラム自動生成の実用化が期待されている．フォーマルメソッドは仕様の無矛盾性を保証できることが知られており，ソフトウェア開発の早期段階から仕様を検証し，関係者の合意を得る有力な手法である．一方，実行するプログラムに対する検証も必要であり，フォーマルな仕様の検証とプログラムの検証の重複を排除することで，さらに開発効率を向上できると考えられる．本研究では，フォーマルな仕様記述言語 VDM++ を用いて記述した機能仕様から実行可能な Java コードを自動生成する場合の検証手法を検討する．具体的には VDM++ によりソフトウェアの機能を検証し，実行プログラムの振る舞いをモデル検査ツール JPF により検証する．すなわち，個々の機能は VDM++ の仕様を満たしているが，それらを組み合わせたときに望ましくない振る舞いをしていないことを確認する手法を提案し，教科書の例題を用いた振る舞い検証の事例を示す．

キーワード：フォーマルメソッド, 仕様記述, 自動コード生成, 振る舞い検証

A parctical approach of behavior verification to Java code that was automatically generated from VDM++ specification

Abstract: Practical use of automatic program generation from a specification is expected in terms of improving efficiency and quality of the software development. Formal methods are known to be powerful ways to ensure the consistency of the specification and reach the agreement of stakeholders from the early stage of the software development. Verification of the program to be executed is also necessary, hence elimination of the duplicated verification of the program and the formal specification is important for further improvement of the development efficiency. We consider a verification method over automatically generate Java code from a functional specification described by a formal specification language VDM++. The idea is that functionalities are verified in VDM++ and behaviors of executable program are verified by JPF, a model checking tool. We propose a practical method to ensure that undesirable behaviors which are combinations of functions are not appear, though individual function meets the specifications of the VDM++. A case study of the verification is shown using a small example in the VDM++ textbook.

Keywords: Formal method, Specification Description, Code Generation, Behaviour Verification

1. はじめに

ソフトウェア開発の効率改善や品質向上の観点から，仕様からのプログラム自動生成の実用化について，さまざまな研究が行われてきた．その歴史は古く 1940 年代に遡るが，ほとんどはより抽象的な記述を仕様として受け取り，より効率のよいプログラムを生成するものであり，アルゴリズムを自動生成するような方式はほとんど実用化されていない [6]．Parnas の論文は 1985 年のものであるが，制約論理を用いた研究や特定分野向けの応用では成果があるものの，一般的なプログラムに関しては現在でもあまり状況が変わっていない．本研究で利用するフォーマルメソッドの一種である VDM のプログラム自動生成ツールも，アルゴリズムレベルの自動生成ではなく，より抽象的な仕様記述言語 VDM++ から Java 言語への文法的な対応づけによる変換を行うものである．

[1] 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu Univ ersity

1.1 研究方針

本稿では，このようなツールによる仕様記述からのプログラム自動生成の現状を評価し，プログラム自動生成を前提としたソフトウェア開発の各工程における検証過程の位置づけを検討する．ソフトウェア検証は機能に関する検証と振る舞いに関する検証に大別できるので，今回はそれぞれを仕様レベル，プログラムレベルで検証するように役割分担を行う場合について検討する．検証はソフトウェア開発の重要な作業ではあり，仕様とプログラムの両方で機能と振る舞いの両方を検証するのが理想であるが，常用上の効率は無視できない．検証項目の重複をできるだけ避けるために，仕様に対して機能検証を，プログラムに対して振る舞いを検証する組み合わせを行うこととする．

このような方針をとるのは， VDM が機能仕様の記述に適しているが振る舞いの検証はやや苦手としているからである．プログラムの自動生成を前提とすると， VDM を用いて仕様記述および機能検証を行った結果に基づいて自動生成した Java プログラムは， VDM 上の機能に関する検証成果を反映したものになっていると期待できる．一方，振る舞いに関する検証は十分でない可能性があるので，これを Java プログラムに対するモデル検査ツール JPF を用いて検証する．JPF の検証条件は機能の組み合わせにより実現される振る舞いに関するものに限定してよい．本稿では特に問題となることの多い並行処理に関する振る舞いについて検討する．

このような方針では，プログラムの自動生成の効果は VDM 仕様から Java プログラムへの自明な書き換えをツール任せにする省力化と，生成される Java プログラムの VDM 仕様への適合性の向上が主となる．このため，プログラム自動生成による作業量の削減についての評価などは行わない．

1.2 フォーマルメソッドの応用

先述のように，汎用的なプログラムの自動生成を行うためには，対象となるソフトウェアの仕様をアルゴリズムも含めて厳密に記述しなくてはならない．フォーマルメソッドは，計算機システムの仕様を数理的に表記することおよび，その仕様を設計の検証基盤として利用する手法である [3]．フォーマルメソッドは仕様の無矛盾性を保証できることが知られており，ソフトウェア開発の早期段階から仕様を検証し，関係者の合意を得る有力な手法である．特に上流工程からのソフトウェア品質向上に有用であることが知られており，提案初期の頃から利用されてきたプラント制御や交通管制といったクリティカルシステムだけでなく，さまざまな応用分野，適用範囲において多くの適用例がある [2]．

フォーマルメソッドを適用し，ソフトウェアの仕様を数理的なモデルとして記述することにより，その数理体系における矛盾や曖昧さを取り除くことが可能となるので，フォーマルメソッドを用いた仕様記述からのプログラムの自動生成は，自然言語や UML など他の手法によりも解釈の一意性や仕様の無矛盾性の観点から優位である．

本研究では，フォーマルなモデルの記述には VDM++ を採用した．VDM++ は，一階述語論理と集合論に基づく意味論をもつフォーマルメソッド VDM の仕様記述言語である [4]．VDM は対象の仕様を有限状態機械によるモデルとして記述していくことに重点をおいており，数学的な仕様から実装に近い仕様までさまざまな抽象度で記述可能である．VDM++ は宣言型の言語であり，明示的に指定しない限り，ファイル内の記述順には依存しない．クラスによるカプセル化とアクセス制御，継承など OOA/OOD 記述や非同期並行処理オブジェクト指向設計の記法を一部サポートしている．

VDM++ のファイル構成は図のようになっている．

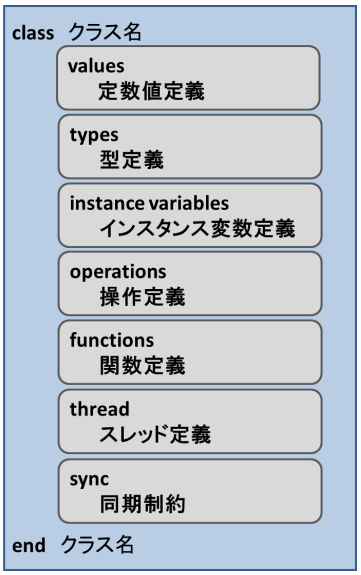

図 1 VDM クラスの構成
Fig. 1 Structure of a VDM class

内部の楕円は Section と呼ばれ，それぞれの対応する定義を記述する．同じ Section を複数書くことも許される．

また，関数や操作を記述する際，陽記述と陰記述の 2 つの異なる方法で記述することが可能である．陽記述は，関数や操作の仕様をアルゴリズムを陽に記述することで規定し，入力値を用いて結果の値を計算可能とする．陰記述は，関数や操作の仕様を事前条件や事後条件により規定し，計算アルゴリズムは記述しない．陰記述では，動作アルゴリズムを仕様として規定せずに実装任せとし，結果が満たすべき性質をキーワード post で記述される事後条件で表現している．それぞれの表現の違いについて，2 次方程式の解法における例を，陽記述については図 2 に，陰記述については図 3 に示す．

以下，本稿の構成は，第 2 章で， VDM ツールによる

```
explicit_solver : real * real * real -> real
explicit_solver(a,b,c) ==
  -b+sqrt(b*b-4*a*c)/(2*a*c)
pre b*b-4*a*c >= 0
post a*RESULT*RESULT + b*RESULT + c = 0;
```

図 2 陽記述 VDM++
Fig. 2 Explicit VDM++

```
implicit_solver(a:real,b:real,c:real) x:real
post a*x*x + b*x + c = 0;
```

図 3 陰記述 VDM++
Fig. 3 Implicit VDM++

Java プログラム生成の現状と記述すべき仕様について述べる．第 3 章では，事例によるプログラムの自動生成を行い VDM 仕様と Java プログラムを対照する．さらに，第 4 章では VDM 仕様および Java プログラムの検証項目について評価し，第 5 章でまとめを行う．

2. ツールによるプログラム生成

2.1 VDMTools

デンマークの IFAD 社において開発された VDMTools は VDM++ の検証支援ツールである．現在は日本の SCSK が権利を取得し，無料で利用可能となっている [7]．このツールはさまざまな検証機能を提供しているが，本研究では以下の機能を利用する．

- 仕様の構文検査・型検査
  構文検査・型検査機能は，構文の正しさや型の整合性などを検査する機能である．構文エラーがなければ，VDMTools のマネージャウィンドに S マークが表示される．型エラーがなければ，S マークの隣に T マークが表示される．型検査は構文検査が成功していなければ行うことができない．
- 仕様アニメーション
  陽記述は VDMTools の実行ウィンドウにコマンドを入力することで，計算結果の出力や値の表示を対話的に行うことができる．これを仕様のアニメーションと呼び，必要なデータを与えることで具体的な値の計算を行い，アルゴリズムのデバッグをしたり，仕様の妥当性を確認したりできる．具体的な使用例を図 4 に示す．
- Java コード自動生成
  VDMTools で Java コード自動生成機能を実行すれば，Java プログラムが自動生成される．自動生成の例として，挿入ソートの VDM++ 陽記述仕様を図 5 に，それを元に自動生成された Java プログラムを図 6 に示す．

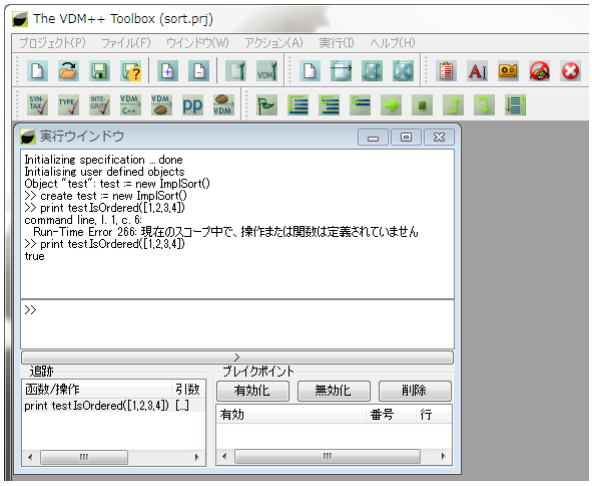

図 4 VDMTools による対話的実行
Fig. 4 Interactive execution on VDMTools

```
class InsertSort

operations

  public Sort: seq of int ==> seq of int
  Sort(l) ==
    return InsertSorted(hd l, l)

functions

  InsertSorted: int * seq of int -> seq of int
  InsertSorted(i,l) ==
    cases true :
      (l = [])    -> [i],
      (i <= hd l) -> [i] ^ l,
      others      -> [hd l] ^ InsertSorted(i,tl l)
    end
  measure Len;

  Len: int * seq of int -> nat
  Len(-,list) ==
    len list
end InsertSort
```

図 5 VDM++ による挿入ソート仕様
Fig. 5 Insertion Sort on VDM++

```
//
// THIS FILE IS AUTOMATICALLY GENERATED!!
//
// Generated at 2015-05-13 by the VDM++ to JAVA Code Generator
// (v9.0.4 - Thu 01-May-2014 08:46:17 +0900)
//
// ***** VDMTOOLS START Name=HeaderComment KEEP=NO
// ***** VDMTOOLS END Name=HeaderComment

// This file was genereted from "InsertSort.vdmpp".

// ***** VDMTOOLS START Name=package KEEP=NO
// ***** VDMTOOLS END Name=package
// ***** VDMTOOLS START Name=imports KEEP=NO
import java.util.List;
import java.util.ArrayList;
import jp.vdmtools.VDM.UTIL;
import jp.vdmtools.VDM.Sentinel;
import jp.vdmtools.VDM.EvaluatePP;
import jp.vdmtools.VDM.CGException;
// ***** VDMTOOLS END Name=imports

public class InsertSort implements EvaluatePP {

// ***** VDMTOOLS START Name=sentinel KEEP=NO
  volatile Sentinel sentinel;
// ***** VDMTOOLS END Name=sentinel
```

```
// ***** VDMTOOLS START Name=InsertSortSentinel KEEP=NO
  class InsertSortSentinel extends Sentinel {

    public final int Sort = 0;
    public final int nr_functions = 1;

    public InsertSortSentinel () throws CGException {}

    public InsertSortSentinel (EvaluatePP instance) throws CGException {
      init(nr_functions, instance);
    }
  }
// ***** VDMTOOLS END Name=InsertSortSentinel
;

// ***** VDMTOOLS START Name=evaluatePP#1|int KEEP=NO
  public Boolean evaluatePP (int fnr) throws CGException {
    return Boolean.TRUE;
  }
// ***** VDMTOOLS END Name=evaluatePP#1|int

// ***** VDMTOOLS START Name=setSentinel KEEP=NO
  public void setSentinel () {
    try {
      sentinel = new InsertSortSentinel(this);
    }
    catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
// ***** VDMTOOLS END Name=setSentinel

// ***** VDMTOOLS START Name=vdm_init_InsertSort KEEP=NO
  private void vdm_init_InsertSort () {
    try {
      setSentinel();
    }
    catch (Exception e) {
      e.printStackTrace(System.out);
      System.out.println(e.getMessage());
    }
  }
// ***** VDMTOOLS END Name=vdm_init_InsertSort

// ***** VDMTOOLS START Name=InsertSort KEEP=NO
  public InsertSort () throws CGException {
    vdm_init_InsertSort();
  }
// ***** VDMTOOLS END Name=InsertSort

// ***** VDMTOOLS START Name=Sort#1|List KEEP=NO
  public List Sort (final List l) throws CGException {
    sentinel.entering(((InsertSortSentinel)sentinel).Sort);
    try {
      return InsertSorted(UTIL.NumberToInt(l.get(0)), l);
    }
    finally {
      sentinel.leaving(((InsertSortSentinel)sentinel).Sort);
    }
  }
// ***** VDMTOOLS END Name=Sort#1|List

// ***** VDMTOOLS START Name=InsertSorted#2|Number|List KEEP=NO
  private List InsertSorted (final Number i, final List l) throws CGException {
    List varRes_3 = null;
    boolean succ_4 = false;
    {
      /* (l = []) -> [i] */
      /* (l = []) */
      succ_4 = (UTIL.equals(Boolean.TRUE, \\
  Boolean.valueOf(UTIL.equals(l, new ArrayList()))));
      if (succ_4) {
        /* [i] */
        List tmpSeq_10 = new ArrayList();
        tmpSeq_10.add(i);
        varRes_3 = tmpSeq_10;
      }
    }
    if (!succ_4) {
      /* (i <= hd l) -> [i] ^ l */
      /* (i <= hd l) */
      succ_4 = (UTIL.equals(Boolean.TRUE, \\
  Boolean.valueOf(i.intValue() <= UTIL.NumberToInt(l.get(0)).intValue())));
      if (succ_4) {
        /* [i] ^ l */
        List tmpSeq_17 = new ArrayList();
        tmpSeq_17.add(i);
        varRes_3 = new ArrayList(tmpSeq_17);
        varRes_3.addAll(l);
      }
    }
    /* others */
    if (!succ_4) {
      List tmpSeq_21 = new ArrayList();
      tmpSeq_21.add(UTIL.NumberToInt(l.get(0)));
      varRes_3 = new ArrayList(tmpSeq_21);
      varRes_3.addAll(InsertSorted(i, new ArrayList(l.subList(1, l.size()))));
    }
    return varRes_3;
  }
// ***** VDMTOOLS END Name=InsertSorted#2|Number|List

// ***** VDMTOOLS START Name=Len#2|Number|List KEEP=NO
  private Number Len (final Number var_1_1, final List var_2_2) throws CGException {
    List list = null;
    /* list */
    list = var_2_2;
    return new Integer(list.size());
  }
// ***** VDMTOOLS END Name=Len#2|Number|List

}
;
```

図 6 自動生成された Java プログラム
Fig. 6 Translated Java Program

生成されたコードの部分は，最初と最後の行に書かれているようなコメントで囲まれており， VDM++ 仕様のどこと対応しているのか分かりやすいような工夫がされている．実際に生成されたプログラムを実行する場合は，VDMTools に付随している VDM.jar を追加する必要がある．現行のツールにおける制限として，数値型については，全て Nuber インターフェースを用いているので，生成後に Integer など Java の適切な型へ変更する必要がある．

2.2 モデル検査ツール JPF

モデル検査とは，検証の対象となる振る舞いの仕様が望ましい性質をみたすかどうかを，あらゆる振る舞いの可能性を自動的に網羅することによって調べる技術である [8]．モデル検査は形式検証技術の１つであり，振る舞いの仕様と性質をそれぞれモデル，および命題で表現し，それらの間の充足関係を調べる．その際に，モデルで起こり得るあらゆる動作パターンを網羅的に列挙し，それぞれのパターンについて命題を充足するかどうかを調べる．後者の動作パターンに対する充足関係は簡単に調べられるため，システム全体の正しさを調べることができる．

振る舞いの仕様を表現するモデルとして，有限オートマトンのような数学的枠組みによる状態遷移モデルを使用する．また性質は、時相論理と呼ばれる論理体系における命題として記述する．時相論理は，「将来いつかある性質が成り立つ」や「次の時刻である性質が成り立つ」といった，時間に関係する性質を記述するための時相演算子を備えた論理体系である．時相論理の特徴として，時系列的な変化に関する性質も記述できるため，並行・分散システムに対して検証したい性質の記述に適している，という点がある．具体的な検証可能な振る舞いの仕様としては，以下のものが挙げられる．

- ステートマシン図などで状態遷移モデルとして規定された仕様記述・モデル
- C 言語や Java など，手続き的に記述されたプログラム

また，時相論理で記述可能な性質としては，以下のものが挙げられる．

- 安全性
  システムが，ある条件のもとで，ある政党でない状況に陥ることが決して起こらないことを表す．例えば，デッドロックやライブロックがないといった性質，あるパラメータが指定された範囲から外れることがない，常に排他制御が正しく行われているといった性質の検証である．
- 活性
  システムが，ある条件のもとで，将来いつか必ず，ある特定の状況が起こることを表す．
- 公平性
  システムが，ある条件のもとで，ある特定の状況が無

限回起こることを表す．特に通信の並行システムの場合は，動作可能なプロセスは必ずそのうち実際に動作実行できるようにスケジューリングする必要がある．

Java PathFinder (JPF) とは，実行可能な Java プログラムのモデル検査ツールである [5]．JPF は NASA の Ames Research Center で開発され，2005 年にオープンソース・ソフトウェアとして公開された．このツールを利用することで，Java コードにおけるデッドロック，競合状態，アプリケーションのアサーションなどといったものを検出することが可能である．

JPF のシステム構造を図 7 に示す．

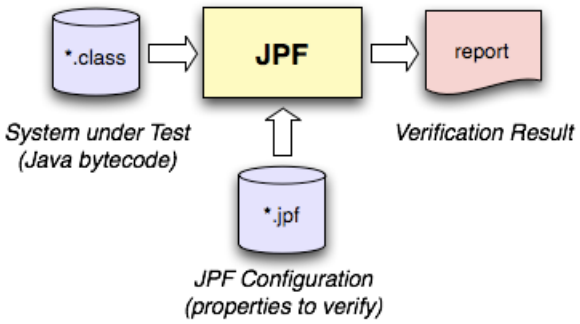

図 7 JPF のシステム構造
Fig. 7 Structure of JPF

JPF を実行するためには，検査対象である Java プログラムと検証したい性質が記述された JPF ファイルの 2 つが必要である．

JPF ファイルは，JPF の動作を制御する設定が記述されたファイルである．JPF を実行する際によく使われる設定は，以下の通りである．

target 検査したいクラス，メソッドを指定する

listener 検査したい性質 (デッドロック，競合状態など) を指定する

search 全パターンを探索するにあたって，どのような優先度 (深さ優先，幅優先など) で探索するのかを指定する．

report report ファイルの形式 (出力する情報や出力先など) を指定する

実行が終了した後は，検証結果が記された report ファイルが出力される．report ファイルには，検証時間，探索したパスの数，探索したパスの最大の深さなどの情報が記載されている．また，設定を追加することで，エラー発生時の各インスタンスの状態，初期状態からエラー発生時までの処理の流れといった情報も出力可能である．

3. ATM の事例

検証事例として，銀行の ATM システムの例題を取り上げる [1]．ATM モデルは VDM++ の陽記述で記述されており，VDMTools のアニメーション機能により実行可能である．本事例は，国際会議におけるフォーマルメソッド比較の一環として記述されており，今回の検証のために恣意的に作成したものではない．また，要求とフォーマルな仕様の齟齬といった問題についても今回は検討しない．

3.1 ATM の構成

ATM モデルは 6 つのクラスで構成されており，クラス図を図 8 に示す．

本事例に対して，以下の手順を適用した．

(1) 陽記述の VDM++ モデルを作成する．

(2) VDMTools の構文検査・型検査機能で静的な検証を行う．

(3) VDMTools のアニメーション機能で，VDM++ モデルをテストする．

(4) VDMTools の Java コード自動生成機能で，VDM++ モデルから Java プログラムを生成する．

(5) 検証したい性質を記述した JPF ファイルを作成する

(6) 生成した Java プログラムと作成した JPF ファイルをもとに JPF を実行する

それぞれのクラスは，次のようになっている．

- AccountDB
  番号に紐付けられた CashAccount のデータベースを管理するクラス．指定した番号の預金アカウントに対して，引き出し，預金といった操作を行う．
- BankAccount
  銀行口座を表現したクラス．口座の名義や口座番号，支店コードの情報を保持する．
- CashAccount
  口座の預金を管理するクラス．この預金アカウントが保持している預金残高に対して，引き出し，預金といった操作を行う．BankAccount クラスのサブクラスである．
- ATMCard
  ATM 用のカードを表現したクラス．カード番号や有効期限の情報を保持している．BankAccount クラスのサブクラスである．
- Keypad
  数値の入出力を管理するクラス．データの読み取り，書き込みの操作を行う．
- ThreadBank
  AccountDB クラスにスレッド機構を追加したクラス．スレッド機構を用いて，預金アカウントに対して引き出し，預金といった操作を行う．AccountDB クラスのサブクラスである．

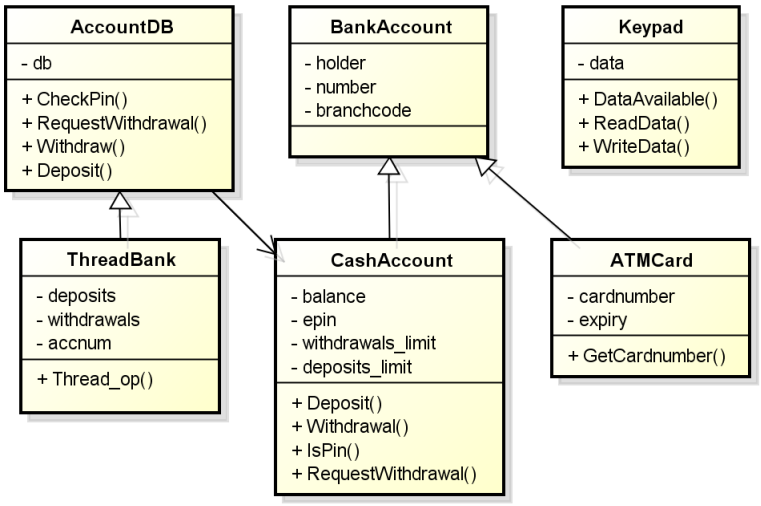

図 8 ATM のクラス構成
Fig. 8 Relation among ATM subsystem

3.2 Java プログラムへの変換

VDMTools の機能を利用して，VDM++で記述された ATM モデルから Java プログラムを自動生成するためには VDMTools の構文検査・型検査の検証を通れば生成可能である．今回の事例では，予め Java プログラムは生成可能であった．しかし，実際に Java プログラムを実行可能とするには，各インスタンスを適切に初期化しなければならない．このため，VDMTools の仕様アニメーション機能を用いて，対応するテストクラスにより仕様を実行し，VDM++ で記述された仕様の妥当性を確認し，適切な初期値を定めることにした．

具体的な銀行 ATM モデルに対するテストクラスを表 1 に示す．

それぞれのファイルの変換前，変換後の大きさを表 2 に示す．

3.3 JPF による検証

VDM++ を用いたしように対して，静的およびアニメーションによる機能仕様の記述および検証を行い，Java プログラムを生成した．

自動生成された Java プログラムを JPF で検証する手順を説明する．検証する性質は，以下の 3 つである．

- デッドロック
- 競合状態
- しらみつぶし探索

これらを検証するためには，検証する性質の設定が記述された JPF ファイルが必要である．それぞれの検証における JPF ファイルの記述について，デッドロックの設定は図 9，競合状態の設定は図 10，しらみつぶし探索の設定は図 11 にそれぞれ示す．

これらの検証設定と VDM++ のテストの時に用いたシ

表 1 TestBank クラス テストケース一覧
Table 1 Test cases for TestBank class

ケース	内容	想定される結果
1	DB に存在する口座へ，預金して引き出す	正常終了
2	DB に存在しない口座に，預金する	AccountDB クラスの Deposit の事前条件違反
3	DB に存在する口座に，不正な金額を預金する	CashAccount クラス Deposit の事前条件違反
4	DB に存在する口座から，負の残高になる程の金額を引き出す	正常終了
5	DB に存在する口座に，預金限度額を越える金額を預金する	CashAccount クラス Deposit の事後条件違反
6	DB に存在する口座から，引出限度額を越える金額を引き出す	CashAccount クラス Withdraw の事後条件違反
7	DB に存在する口座に対して，2 つのスレッドが独立に預金して引き出す	正常終了
8	DB に存在する口座に対して，2 つのスレッドが独立に不正な金額を預金して引き出す	CashAccount クラス Deposit の事前条件違反

表 2 ファイルサイズ比較
Table 2 Size of files

クラス名	VDM++ 行数	VDM++ ファイルサイズ	Java 行数	Java ファイルサイズ
AccoutDB	43	956	201	6173
BankAccout	43	956	136	6173
CashAccout	43	847	171	5083
ATMCard	17	362	122	3274
Keypad	24	424	141	3512
ThreadBank	30	436	129	3303
TestBank	126	2670	399	11603
計	298	5972	1299	36822

```
target = TestBank
listener=.listener.DeadlockAnalyzer
report.console.property_violation=error,trace,snapshot
```

図 9 デッドロック検証設定
Fig. 9 JPF for deadlock check

```
target = TestBank
listener=gov.nasa.jpf.listener.PreciseRaceDetector
report.console.property_violation=error,trace,snapshot
```

図 10 競合検証設定
Fig. 10 JPF for race condition check

```
target = TestBank
cg.enumerate_random = true
report.console.property_violation=error,trace,snapshot
```

図 11 しらみつぶし検証設定
Fig. 11 JPF for exhaustive check

ナリオを組み合わせて JPF による振る舞い検証を行った．デッドロックと競合状態の検証は，スレッドを対象にしているので，表 1 におけるケース 7，ケース 8 で JPF を実行する．しらみつぶし探索は JPF の制約により Java の整数型しか検証できないため，表 1 におけるケース 1 で JPF を実行する．ケース 1 は預金する金額，引き出す金額の 2 つの入力がある．Java の整数型がとりうる値すべてを検査することは現実的ではないので，さらに -1000000〜+1000000 の値の範囲で-1000000 から 10000 ずつ増加させて入力の値とし，それぞれ 200 個ずつ，組み合わせて 40000 通りを検証する．このように，実際の検証にあたっては，検証環境や目的により，検証条件を検討する必要があり， JPF についてもプログラム生成を自動化したとしても，全くの手放しとはいかない．これらの組み合わせを，表 3 に示す．

4. 考察

4.1 実行結果

本稿における検証環境を以下に示す．

- The VDM++ Toolbox v 9.0.5 (VDMTools)
- Java PathFinder version 1.3

表 3 JPF による検証の概要
Table 3 Summary of JPF verification

検証項目	JPF ファイル	テストケース
デッドロック	ソースコード 9	ケース 7，ケース 8
競合状態	ソースコード 10	ケース 7，ケース 8
しらみつぶし探索	ソースコード 11	ケース 1

- Eclipse4.4 Luna (Java 8，JDK 1.8，VDM.jar)
- CPU i5-460M プロセッサー 2.53GHz (2 コア 4 スレッド)
- メモリ 4GB

表 4 VDM++ インタープリタ実行結果一覧
Table 4 Results of VDM++ animations

ケース	想定	実行結果
1	正常終了	正常終了
2	AccountDB クラスの Deposit の事前条件違反	AccountDB クラスの Deposit の事前条件違反
3	CashAccount クラスの Deposit の事前条件違反	CashAccount クラスの Deposit の事前条件違反
4	正常終了	正常終了
5	CashAccount クラス Deposit の事後条件違反	CashAccount クラス Deposit の事後条件違反
6	CashAccount クラス Withdraw の事後条件違反	CashAccount クラス Withdraw の事後条件違反
7	正常終了	正常終了だが，初期状態から変化なし
8	CashAccount クラス Deposit の事前条件違反	正常終了だが，初期状態から変化なし

この環境で，時間的なオーバヘッドなく VDM++ からの Java プログラム生成を行うことができた．しかしながら，ソフトウェア開発におけるコーディングの重みは小さくなっており，今回の事例においてもコーディング作業の多くを占めるアルゴリズムやデータ構造の設計は VDM++ の陽記述として決定済みである．このことから，従来のフォーマルメソッド適応でしばしばみられた，仕様とプログラムのそれぞれで設計を行う 2 度手間を避けるという効果はあったが，コーディング作業を決定的に小さくしたとは言えない．ただし，繰り返しの修正や確実な変更というプログラムの品質保証の観点からは，プログラムの自動生成の効果は非常に大きい．

VDMTools のインタープリタ機能によるテストケース実行結果を表 4 に示す．スレッドを用いていないテストケースは，実行結果が想定された結果と同じであった．一方，スレッドを用いているケース 7，ケース 8 においては実行後も初期状態の値から変化がみられず，想定とは異なる結果になった．このようなマルチスレッドによる制御モデルのように，複雑な VDM++ 仕様からの生成に関してはプログラム生成部分も含めてあまりあてにならないという残

念な結果になった．

自動生成された Java プログラムにおけるテストケースの実行結果を表 6 に示す．スレッドを用いているケース 7，ケース 8 において，実行結果が想定した結果と同じであった．ただ，想定では事後条件を違反しているケース 5，ケース 6 において，実行結果は正常終了となっている．これは，VDMTools の Java コード自動生成機能では， VDM++ の操作定義の事後条件は Java コードの生成対象となっておらず，実用上の問題は小さいものの本来の検証条件が抜けているためである．

表 5 Java テスト実行結果一覧
表 6 Results of Java tests

ケース	想定	実行結果
1	正常終了	正常終了
2	AccountDB クラスの Deposit の事前条件違反	AccountDB クラスの Deposit の事前条件違反
3	CashAccount クラスの Deposit の事前条件違反	CashAccount クラスの Deposit の事前条件違反
4	正常終了	正常終了
5	CashAccount クラス Deposit の事後条件違反	正常終了
6	CashAccount クラス Withdraw の事後条件違反	正常終了
7	正常終了	正常終了
8	CashAccount クラス Deposit の事前条件違反	CashAccount クラス Deposit の事前条件違反

JPF による自動生成された Java コードの振る舞いの検証結果をまとめたものを，表 7 に示す．

性質とケース	エラーメッセージ	検証時間 [sec]	探索したパス数	最大の深さ
DL 7	”no error detected”	92	1069	442
DL 8	”no error detected”	24	1069	292
RC 7	”race for field”	2	168	169
RC 8	”race for array element”	5	291	294
ES 1	”Precondition failure in Deposit”	0	0	3

※ DL=デッドロック，RC=競合状態，ES=しらみつぶし探索
表 7 JPF 振る舞い検証結果まとめ

5. おわりに

本稿では，ソフトウェア開発におけるプログラム自動生成を前提とした，検証手法の適用を検討した．フォーマルメソッドを利用した開発早期からの仕様検証と，プログラムレベルでのモデル検査による検証を組み合わせ，事例を用いた簡単な評価を行った．本事例では， VDM++ による銀行 ATM の事例に対して， VDM++ による仕様記述を行い， VDMTools による Java プログラムの自動生成を行った．生成された Java プログラムに対して，モデル検査ツール JPF を利用した振る舞い検証を行い，概ね予想通りの結果を確認することができた．

この事例適用の結果，次のような知見が得られた．

- VDM++ からの Java プログラム生成は，かなりの部分で可能である．ただし，自動生成された Java コードの可読性は人手によるコードより劣るので，期待通りに動かなかった場合にどこを修正すればよいかを特定するのは難しい場合がある．生成されたプログラムを信頼して利用するためには事前の評価を十分に行い，問題点を洗い出しておく必要がある．
- 検証にあたっては， VDM++ の妥当性確認に用いるシナリオと， JPF の振る舞い検証に用いるシナリオは，当然ながら優先順位なども含め，共通性が高い．VDM++ のテストから JPF の検証条件の導出を行うなど，より効率的な検証手法の確立が望ましい．
- 今回の検証の切り分け方針は，それぞれの得意な検証を組み合わせられたので，妥当であったといえる．逆に，仕様で振る舞いを検証し，プログラムレベルで機能を検証する組み合わせについても評価したい．

謝辞 本研究は，青山慎二氏 (当時九州大学，現東芝) の研究を発展させたものであり，主に JPF ツール調査に関する貢献に感謝する．本研究の一部は， JSPS 科研費 24220001 ，基盤研究 (S) 「アーキテクチャ指向形式手法に基づく高品質ソフトウェア開発法の提案と実用化」の成果による．

参考文献

[1] Denvir, B. T., Oliveira, J. N. and Plat, N.: The Cash-Point (ATM) 'Problem', *Formal Aspect of Computing*, Vol. 12, No. 4, pp. 211–215 (2000).
[2] IPA/SEC: 厳密な仕様記述における形式手法成功事例調査報告書，技術報告，独立行政法人情報処理推進機構，http://www.ipa.go.jp/sec/reports/20130125.html (2013).
[3] Jones, C. B.: Software Development based on Formal Methods, *Proceedings of the CRAI Workshop on Software Factories and Ada*, LNCS, Vol. 275, Springer-Verlag, pp. 153–172 (1987).
[4] Larsen, P. G., Mukherjee, P., Plat, N., Verhoef, M., Fitzgerald, J., 酒匂寛 (訳)：VDM++によるオブジェクト指向システムの高品質設計と検証，翔泳社 (2010).
[5] NASA, J.-W. J. P. F.: http://babelfish.arc.nasa.gov/trac/jpf/wiki.
[6] Parnas, D. L.: Software Aspects of Strategic Defense Systems, *Communication of ACM*, Vol. 28, No. 12, pp. 1326–1335 (1985).
[7] SCSK システムズ：http://vdmtools.jp/.
[8] 吉岡信成，青木利晃，田原康之：SPIN による設計モデル検証，近代科学社 (2008).