形式仕様記述のプロパティベーステストへの活用

馬場 勇輔[1,a)]　荒木 啓二郎[2,b)]　日下部 茂[2,c)]　大森 洋一[2,d)]

概要：形式的な仕様記述は，仕様における曖昧さに起因する欠陥を予防し，手戻りによるコストを減らすことが期待されている．我々は，形式的な仕様記述に含まれるプロパティを，自動テストの一種であるプロパティベーステストで用いる手法を研究している．形式的な仕様記述を用いたソフトウェア開発において，形式的な仕様記述を自動テストに用いることで，手戻りによるコストを減らすだけでなく，テストのコストを減らすことができ，ソフトウェアの規模や複雑さの増大に伴うテストのコスト増大の問題が解決できる．本発表では，既存の形式的な仕様記述を例にとって提案手法の予備評価を行い，今後の課題とその解決策を論じる．

Utilizing formal specification for property-based testing

Baba Yusuke[1,a)]　Araki Keijiro[2,b)]　Kusakabe Shigeru[2,c)]　Omori Yoichi[2,d)]

Abstract: Formal specification is expected to prevent defects due to ambiguity of the specification. In addition, properties in the formal specification is useful for property-based testing which is one of automatic testing methods. Utilizing formal specification for property-based testing solves the problem the software testing cost which is increaseing along with the increase of the size and complexity of software in recent year. We explain our proposed method and evaluate with preliminary experiment using existing formal specification.

1. はじめに

ソフトウェア開発は，対象の要求分析や設計・仕様策定などの上流工程と，上流工程の成果を基にコーディング・テストなどを行う下流工程に大別される．開発の上流工程における一つの工程である仕様策定で作成した仕様が曖昧なままソフトウェア開発を進めると，しばしばその曖昧さに起因する欠陥を作りこむことも多い．開発が下流工程まで進んでからそのような仕様の曖昧さによる欠陥が発見された場合，仕様の修正まで戻る必要があり，手戻りのコストがかかることになる．そのため，仕様を厳密に記述して曖昧さを減らすことで，仕様の曖昧さに起因する欠陥を予防することが期待される．

また，近年，ソフトウェアの大規模化，複雑化が進むにつれ，テストに対するコストがより大きくなってきている．そのため，テストのコストの削減を目的とした，テストの自動化のための手法・ツールが多く研究・開発されてきた．

本研究では，形式仕様記述をテスト自動化のための情報源にすることで，形式仕様記述を用いたソフトウェア開発において，テストのコストを減らすことを目的とする．

本稿では，第 2 章で本研究の位置づけについて，関連研究と本研究との違いを中心に述べる．第 3 章で形式手法について，本研究で用いる手法を中心に紹介する．第 4 章で，テスト自動化について，本研究で用いるツールを含めて述べる．第 5 章で，提案する手法について説明し，第 6 章で提案手法に関する予備評価を行う．最後に，第 7 章でまとめを述べる．

[1] 九州大学大学院 システム情報科学府 情報知能工学専攻
Faculty of Information Science and Electrical Engineering, Kyushu University
[2] 九州大学大学院 システム情報科学研究院 情報知能工学部門
Faculty of Information Science and Electrical Engineering, Kyushu University
[a)] y.baba@nanotsu.ait.kyushu-u.ac.jp
[b)] araki.@nanotsu.ait.kyushu-u.ac.jp
[c)] kusakabe@nanotsu.ait.kyushu-u.ac.jp
[d)] yomori@ait.kyushu-u.ac.jp

2. 関連研究

形式手法をテストに役立てようとする考えは，新しいものではない．形式手法での仕様記述言語は，モデルベースの言語，有限状態ベースの言語，代数的な言語など，様々な種類があり，それぞれに対して，テストに役立てるための研究がなされてきた [1]．本研究で用いる形式手法 VDM の仕様記述言語は，モデルベースの言語に分類される．モデルベースの形式的な仕様は，集合や列，関係，関数を用いてシステムの状態が記述され，状態の操作は，事前・事後条件が与えられた述語によって記述される．モデルベースの言語として，VDM のほかに Z 記法や B メソッドが挙げられる．代表的なテスト技法として，同値分割がある．これは，入力領域を，複数の領域に分割し，それぞれの領域についてテストケースを選択するものである．モデルベースの仕様からのテスト自動生成技法のほとんどは，同値分割の考えに基づいて研究されている．本研究では，Dick らにより示された VDM 仕様の同値分割手法 [7] を用いている．ただし，Dick らは，同値分割によって得た情報を，有限状態機械を得るのに用いているのに対し，本研究では，同値分割によって得た情報をテストコードの生成に用いている．

次に，関連研究の一つである文献 [2] と，本研究の違いについて述べる．プログラミング言語 FoCaLiZe のコードに対し，ランダムなテストデータの自動生成を行うツールである FocalTest の研究がなされていた [2]．プログラミング言語 FoCaLiZe は，仕様記述から実装まで段階的に記述できる言語であり，形式手法での仕様記述言語の分類における，代数的な言語であると考えられる．また，実行可能なプログラムと，プログラムが満たすべきプロパティを記述でき，形式的な仕様と見ることもできる．プロパティは，事前条件と事後条件に分けることができる．

ランダムなテストデータの自動生成では，MC/DC(Modified Condition/Decision Coverage) に代表される強力なカバレッジ要求を満たすようなテストスイートを得ることは難しい．そこで，文献 [8] では，事前条件の MC/DC を満たすテストスイートを生成できるように，FocalTest を改善した．具体的には，ランダムにテストデータを生成するのではなく，制約推論を用いた手法でテストデータの生成を行うように FocalTest を改善した．

本研究と，文献 [2] との違いを述べる．まず，対象とする言語について，文献 [2] では，プログラムもプロパティも同じ実装言語で記述することが前提となっている．このため，テスト工程の最中にプロパティを記述してテストすることも考えられ，このプロパティの正しさの検証が行われずにテストが行われる可能性がある．これに対し，本研究では，プロパティは，上流工程における形式的な仕様の中に含まれるものであり，上流工程で曖昧さを可能な限り除去されることが前提となっている．このため，プロパティベーステストのツール単体でテストを行うよりも，より正しさの保証されたテストができる可能性がある．

しかしながら，文献 [2] では，MC/DC を満たすテストスイートを得るために，ランダムなテストデータ生成ではなく，制約を解くことによるテストデータ生成を行っている．これに対し，本研究では，ランダムなテストデータ生成を用いている．そのため，強力なカバレッジ要求などを満たすようなテストデータは生成できない可能性がある．

3. 形式手法

形式手法とは，形式仕様記述言語と呼ばれる，数学的に厳密に意味づけられた言語を用いて，開発対象のシステムを記述し，システムがユーザの要求を満たしているかなどを論理的に推論する仕組みを提供する手法である [3].

形式手法では，形式仕様記述と呼ばれる仕様を記述する．形式仕様記述とは，開発対象システムの状態のデータ構造やそれを操作する機能などを抽象化し，形式的な仕様として記述をすることで，仕様の形式的な検証を行うことを可能にする手法である．本研究では，形式手法の一つである VDM を用いる．

3.1 VDM

システムが持つ型や状態，入力に対する出力への写像を示す関数，状態を変更するための操作などを記述できる．

図 1 に，VDM の形式仕様記述言語である VDM++を用いた仕様例を示す．図 1 では，String という文字の列を示す型を定義し，文字列の反転を示す操作 reverseString，文字列の追加を行う関数 appendReverse を定義している．reverseString のように事前条件 (pre)，事後条件 (post) によって定義した関数・操作を陰関数・陰操作，appendString のようにアルゴリズムによって定義した仕様を陽関数・陽操作という．また，陽関数・陽操作においても事前条件，事後条件を定義することが可能である．事前条件，事後条件が定義された陽仕様を拡張陽関数・拡張陽操作という．

3.2 VDM の開発支援ツール

VDM には，開発支援のためのツールが複数存在する．VDM の仕様記述言語には VDM-SL，VDM++，VDM-RT があり，処理できる言語やツールのもつ機能に違いがある部分を以下に説明する．共通する機能としては，構文・型検査機能，インタープリタ，デバッグ機能がある．また，OS について，VDMPad のサーバプログラム以外は，Windows，Linux，Mac Os X で動作する．

VDMJ

コマンドラインツールであり，VDM-SL，VDM++，VDM-RT に対応している．カバレッジ計測機能や証

```
class StringExample
types
 String = seq of char

operations
 reverseString(s:String)out:String
 pre s <> ""
 post forall i in set inds s & out(i)=s((len s)+1-i)

functions
 appendString:String * String -> String
 appendString(s1, s2) == s1^s2

end StringExample
```

図 1　VDM 仕様例

明課題生成機能を持っている．

Overture Tool

VDMJ を用いた Eclipse ベースの統合開発環境であり，VDM-SL，VDM++，VDM-RT に対応している．カバレッジ計測機能や証明課題生成機能を持っている．

VDMTools

外部エディタを用いて，開発を行うツールであり，VDM-SL，VDM++，VDM-RT に対応している．証明課題生成機能，コードカバレッジ計測機能，実行可能仕様から Java や C++への生成機能がある．

VDMPad

Max OS X・Linux で動作するサーバプログラムや，Web 上での開発ができる統合開発環境であり，VDM-SL のみに対応している．状態変数の視覚的な表現機能など，上記の 3 つのツールよりもインタラクティブな機能がある．

また，VDM++に対応したテストフレームワークである，VDMUnit がある．

4. テスト自動化

4.1 ソフトウェア開発プロセス

図 2 にソフトウェア開発におけるプロセスの一つである V 字モデルを示す．図 2 は，コーディングより左側が上流工程，コーディングを含む右側が下流工程となり，青の矢印に沿ってソフトウェア開発が進むことを示している．上流工程において，要求分析では要求仕様書，機能設計では機能仕様書のように，成果物が出力される．上流工程における成果物は，同じレベルの下流工程（例：機能設計と結合テスト）のためのテストベースとなる．一般的な開発では，成果物は自然言語で記述されており，仕様の曖昧さに起因した欠陥がテストベースに混入することがありうる．これにより，仕様の曖昧さに起因した欠陥を取り除くための手戻りによるコストがかかり，余計なテストのコストがかかる可能性がある．

前章で述べた形式手法を用いたソフトウェア開発では，上流工程における成果物は，形式仕様記述言語にて記述されることになる．よって，テストベースでの曖昧さが減り，

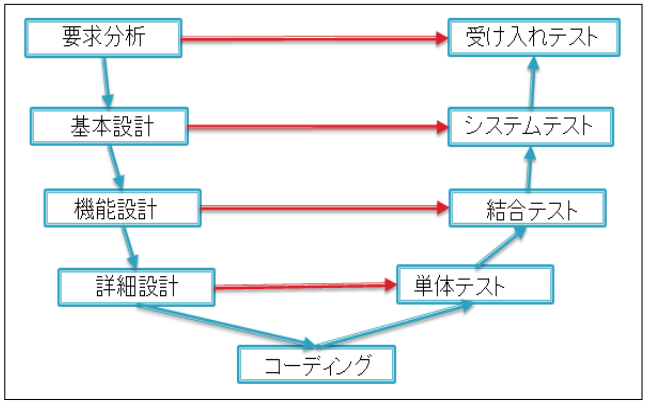

図 2　ソフトウェア開発プロセスの V 字モデル

自然言語による仕様のときよりも，テストのコストを減らせることが期待される．

4.2 テスト自動化

図 2 における各テスト工程（単体テスト，結合テスト，システムテスト，受け入れテスト）は，テスト分析，テスト設計，テスト実装，コード解析，テスト実行，テスト結果管理（モニタリングとコントロール，報告書作成），テストウェア管理，インシデント管理の結果の成否の確認の 8 つの作業に分けることができる [4]．テストの自動化とは，この作業のいくつかについて自動的に行うことをいう．テストの自動化を進めるに当たり，テスト工程のどの作業を自動化するかを明確にする必要がある．テスト設計は，テストベースに従って行う．テストベースとは，テストケースを作成する際に参考にする情報の集合である．前述の形式的な仕様もテストベースになりうる．形式的な仕様は状態やそれに対する操作の入出力の情報が記述されているため，単体テストや結合テスト，システムテストに対するテストベースとなる．本研究では，VDM 仕様をテストベースとする前提で，テスト実装・実行の自動化の支援を目的としている．この目的のため，テスト自動化の手法の一つであるプロパティベーステストを用いる．

4.3 プロパティベーステスト

プロパティベーステストとは，テスト自動実行の手法の一つであり，テスト対象の性質（プロパティ，property）が満たされるか否かの観点から，データを自動生成して確かめる手法である．ツール例としては，QuickCheck や ScalaCheck などが挙げられる．QuickCheck や ScalaCheck は，テストデータをランダムに多数生成し，プロパティが満たされるか否かを自動的にテストを実行するツールである．また，これらはテストデータを自動生成する機能（ジェネレータ）と，それを用いてプロパティを満たすか否かを自動テストする機能に分けることができ，目的に応じて使用する．プロパティベーステストが自動化するテストの作業は，テスト分析やテスト設計，テスト実行，テスト結果

管理であり，目的によって異なってくる．ジェネレータのみを用いるのであれば，テスト分析やテスト設計の自動化をすることになり，プロパティを満たすか否かまでテストするのであれば，テスト実行やテスト結果管理まで自動化することになる．

4.4 テスト自動化ツール

自動化ツールは，それぞれ自動化の対象とするテスト工程中の作業をもっている．本研究で用いる，テスト実行の作業を自動化するツールについて，比較のために既存のツールを挙げる．

Selenium

Web アプリケーションに対するテスト自動実行支援のツールである．ブラウザなどのユーザーインターフェィスからの操作を記録し，テストコードを保存する．このテストコードを実行することで，記録時の操作の再現を行うことができる．

JUnit

プログラミング言語 Java で記述されたコードに対する，ユニットテストの実行支援ツールである．Junit の API を利用したテストコードを作成し，テストコードを実行することで，テスト実行および，結果の検証ができる．

例に挙げたテスト実行の作業を自動化する 2 つのツールは，テストコード作成者が具体的な入力値を指定してテストを実行する．これに対し，前節で述べたプロパティベーステストは，テストデータを自動的に多数生成してテストを実行する．これは，ユーザが想定していないような入力値によるテストを行えることが期待される．また，近年のテスト対象の大規模化，複雑化から，具体的な入力値を指定して行うコストは増大化していると考えられる．

また，テスト実装の作業を自動化するツールについても既存のツールが存在する．形式的な仕様はテストベースとなりうるが，記述することが難しいことから，テスト項目表・テストケース表から JUnit のテストコードを自動的に生成する TERASOLUNA RACTES for UT[5] がある．

後述する提案手法では，形式的な仕様からのテスト実装の作業の自動化を行うツールを作成する．形式的な仕様について，VDM の仕様記述言語にて記述することを前提としている．VDM 仕様は一部の機能のみの記述も可能であり，一部のみであればテスト項目表・テストケースを自然言語仕様から抽出する難しさとさほど変わらないと考える．一部の機能へのツールの適用ができ，ツールが出力したテストコードでのテストが有用であれば，他の機能の記述を行うモチベーションにもなることが期待される．

5. 提案手法

形式仕様記述では，開発対象のプロパティが記述される

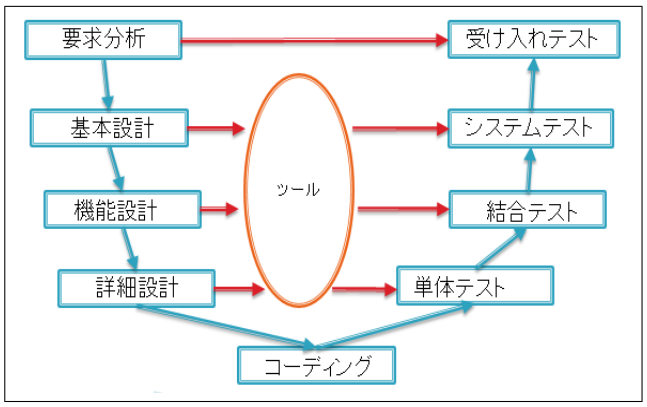

図 3 ツール適用時の V 字モデル

ことになる．本研究では，VDM 仕様から開発対象のプロパティを抽出し，抽出したプロパティをプロパティベーステストに活用することを提案する．提案手法を実現するためのツールを作成することで，VDM を用いたソフトウェア開発におけるテスト自動化の支援を図る．図 2 にツールの適用を加えたときの V 字モデルを**図 3** に示す．図 3 において，基本設計，機能設計，詳細設計での成果物をツールの入力とし，プロパティベーステストのためのテストコードをツールの出力とし，各テスト工程で，テストコードを用いてテスト実行を行う図を示している．図 3 において，要求分析の成果物を入力としていないのは，要求分析の成果物には安全性などの非機能要求に関するものがあり，非機能要求に関する仕様は VDM 仕様での表現が難しく，ツールへの適用が難しいと考えたからである．

提案手法を用いたツールにおける，自動化を支援するテスト工程の作業は，テスト実装と実行である．

ここで，文献 [6] において，VDM の形式仕様記述言語 VDM++とプログラミング言語 Scala はよく似ていることが述べられている．このことから，テスト実装の自動化の観点において，適用が容易であると考えられる．VDM を用いたソフトウェア開発において，Scala で実装を行う場合について考え，プロパティベーステストのツールは，Scala のコードに対して用いることのできる ScalaCheck を用いる．

図 4 に ScalaCheck を用いたテストコードの例を示す．図 4 において，def はメソッドの定義を示し，genString メソッドは任意の文字列を生成するジェネレータである．property(”reverse_prop”) は文字列の反転に関するプロパティである．これは，genString メソッドによって生成された文字列 s が，s を 2 度反転した文字列と等しいことを示すプロパティである．また，property(”append_prop”) は文字列の追加に関するプロパティである．これは genString を 2 回呼び出し，任意の文字列 s1，s2 を生成し，実装した appendString メソッドに入力した結果が，s1 と s2 を連結した文字列と等しいことを示す．図 1 に示すような VDM 仕様から，図 4 に示すようなテストコードを自動生成する

```
import org.scalacheck.Arbitrary._
import org.scalacheck.Prop._
import org.scalacheck.Properties

object StringTest extends Properties("StringTest"){
val genString =arbitrary[String]

property("reverse_prop") = forAll(genString){s=> s.equals(s.reverse.reverse)}
property("append_prop") = forAll(genString, genString){x=> appendString(s1,s2) == s1+s2}
}
```

図 4　ScalaCheck テストコード例

ツールを作成することが，本研究の目標となる．

5.1　プロパティの変換

この節では，VDM 仕様に含まれるプロパティを変換する方針を示す．

ジェネレータに関しては，以下のようなジェネレータを作成する．

ジェネレータ群 1
　型定義の値を生成するジェネレータ

ジェネレータ群 2
　関数・操作定義の事前条件を満たす値を生成するジェネレータ

ジェネレータ群 3
　関数・操作定義の事前条件を満たさない値を生成するジェネレータ

ジェネレータ群 4
　関数・操作定義の事後条件を満たす値を生成するジェネレータ

また，上記のジェネレータを用いて，以下の観点から，関数・操作が満たすべきプロパティ群を作成する．

プロパティ群 1
　事前条件のない関数・操作に，入力型のジェネレータから生成したデータを，関数・操作に入力したときの出力結果が，出力型に属する値である．

プロパティ群 2
　事前条件をもつ関数・操作は，事前条件を満たす入力がされたとき，出力型に属する値を出力する．

プロパティ群 3
　事前条件をもつ関数・操作は，事前条件を満たさない入力がされたとき，出力型に属さない値が出力される（例外が発生する）．

プロパティ群 4
　事後条件をもつ関数・操作は，その出力が事後条件を満たす．

このプロパティ群の番号で用いるジェネレータは，同じ番号のジェネレータ群と対応する．ここで，ジェネレータ群 4・プロパティ群 4 では，論文 [7] に示されている，操作の事前・事後条件中の変数に関する同値分割の手法を用いて，ジェネレータ・プロパティを作成する．論文 [7] では，VDM 仕様における陰操作を対象とし，各操作の事前条件・事後条件などを集めて論理式とする．論理式を簡約化し，DNF(Disjunctive Normal Form）にすることで，各操作で用いられている変数に関するテストケースを得る．このテストケースは，DNF の各節であり，操作に用いられている変数に関する条件群を示す．よって，テストケース中の操作の入力変数に関する条件からジェネレータを作成し，このジェネレータから生成した値を入力した出力結果が，テストケース中の出力条件を満たしているというプロパティを作成する．

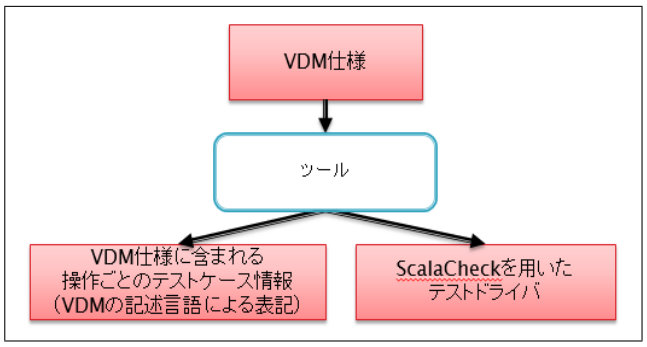

図 5　作成するツールの入出力

表 1　作成するツールの内部モジュール

モジュール名	入力	出力
Parser	VDM 仕様	AST
AST2DNF	AST	DNF
DNF2SMT	DNF	SMT
DNF2TestCase	DNF	テストケース
DNF2ScalaCheck	DNF	テストコード

5.2　作成するツール

作成するツールのツールの入出力と，内部処理について説明する．

ツールの入出力を**図 5** に示す．図 5 に示す出力の「操作ごとのテストケース情報」は，「ScalaCheck を用いたテストドライバ」を用いたテストがどのようなテストを行うことになるのかを示すために出力される．ここで，テストドライバとは，実装した操作を呼び出してテストを行うテストコードのことを指す．

また，ツールの内部処理を行うモジュールを**表 1** に示す．図 1 に示すモジュールについてそれぞれ説明する．

Parser
　ツールの入力である VDM 仕様を入力とし，VDM 仕様の構文解析を行い，抽象構文木 (AST) を xml 形式で出力する．

AST2DNF
　AST を入力とし，AST から前節で示した操作の用いられている変数に関する同値分割を行い，DNF を xml 形式で出力する．

DNF2SMT

DNF を入力とし，DNF の各節を SMT(Satisfiability Modulo Theories) に変換し，出力する．

SMTSolver

SMT を入力とし，DNF の節で充足不能な節を取り除き，DNF を出力する．

DNF2TestCase

DNF を入力とし，正しい入力型から出力型が生成されることを示すテストケースや操作に用いられている変数の同値分割を行ったテストケースを生成する．

DNF2ScalaCheck

DNF を入力とし，DNF 中に用いられている VDM の仕様記述言語の構文を Scala 構文に変換し，ScalaCheck を用いたテストドライバを生成する．

6. 予備評価

提案手法を用いたテストを行った際に，ジェネレータによるデータの生成に偏りが起きて，実装コードの一部しかテストできていないなどがないかの確認のため，予備評価を行う．

今回，VDM 仕様の例として，文献 [8] に示されている ATM での預金の引き出しに関する仕様を用いる．この ATM の仕様は，預金の引き出しの振る舞いと，振る舞いが正しいかを評価する部分とが明確に分かれており，プロパティの抽出が簡単であるため，予備評価にこの ATM の仕様を用いた．

図 6 に ATM での預金引き出しの振る舞い例を示す．図 6 では，キャッシュカードの情報が口座情報 DB に存在し，正しい PIN と預金額以内の引き出し額が入力されたときの振る舞いである．このとき，挿入されたキャッシュカードと，引き出し額の現金を出力し，口座情報 DB 中の預金額を変更する．

文献 [8] に示されている ATM の仕様中で定義されている関数・操作のうち，インターフェイスとなる操作を**図 7** に示す．図 7 は，ATM に入力が行われたとき，図 7 に示す操作が呼ばれ，カードの差し押さえや，要求額の金額を払いだすなどの処理が行われる仕様となる．図 7 の impl_ATM_引き出し仕様_実行が処理の実体となる操作のであり，事後条件中の ATM_引き出し仕様が，入力に対する出力が正しいか否かを示す関数となる．事後条件中の **RESULT** は，impl_ATM_引き出し仕様_実行の出力結果を示す．

また，impl_ATM_引き出し仕様_実行や ATM_引き出し仕様の補助関数として，複数定義されている．以降，ATM_引き出し仕様_実行とその補助関数（impl_ATM_引き出し仕様_実行含む）を操作的部分，ATM_引き出し仕様とその補助関数を宣言的部分として説明する．ただし，重複するものは，どちらにも属するものとする．

次に，型定義について説明する．INPUT 型では，挿入するキャッシュカード情報や暗証番号，引き出し額で構成される，ATM のユーザが入力するデータについて定義している．OUTPUT 型では，返還するキャッシュカードや払い出し額で構成される，ATM の出力するデータについて定義している．ACCOUNT_DB 型では，口座情報を管理するデータベースについて定義している．お金や番号に関する型は，簡単のため，全て正の整数として扱う．

この仕様では，型名や関数名，操作名は日本語で定義されているものがある．テスト実装の自動化の観点から，この仕様の日本語部分を英語に直したものを使用する．また，実装の制限として，以下のように定める．

制限 1

入力値が事前条件に反している場合の挙動など，仕様に記述されていないものは実装しないようにする

制限 2

操作的部分のみを実装する．

制限 1 の理由は，仕様に記述されていないことを勝手に実装することは，バグの温床となりうると考えるからである．制限 2 の理由は，ある入力に対する出力が正しいかどうかを判断するのはテストで行うので，宣言的部分に関しての実装は必要ないからである．

最後に，仕様での型と実装での型の対応は分かっているものとして，ジェネレータやプロパティを作成する．ジェネレータは前章で説明したとおりものを作成した．また，プロパティ群 1 は操作的部分の操作・関数のそれぞれについて作成した．プロパティ群 2，プロパティ群 3 は操作的部分の事前条件をもつ関数についてのみ作成した．プロパティ群 4 は事後条件を唯一持っている ATM_引き出し仕様_実行について作成した．

6.1 テスト結果

前節で，今回用いる仕様例と，それに対する実装，プロパティベーステストで用いる要素について説明した．提案手法を用いたテストの評価を行うために，Scala のビルドツールである sbt によりビルドを行い，テストを行う．ここでのテストは，ジェネレータからテストデータをランダムに生成し，プロパティが満たされているか否かについて

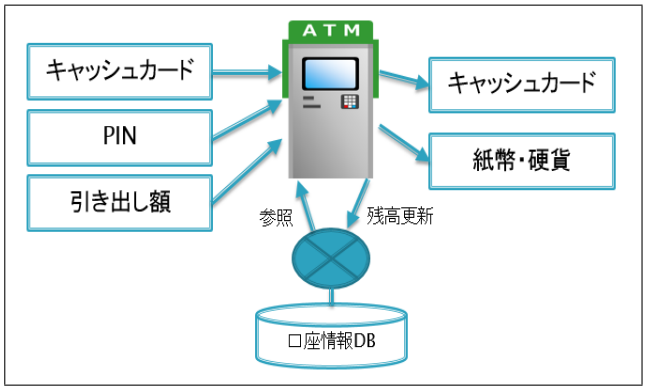

図 6 ATM での預金引き出しの振る舞い

```
public ATM_引き出し仕様_実行 : INPUT * ACCOUNT_DB * STATE
                          ==> OUTPUT * ACCOUNT_DB * STATE
ATM_引き出し仕様_実行(input, db_in, state_in) ==
  impl_ATM_引き出し仕様_実行(input, db_in, state_in)
pre true
post ATM_引き出し仕様(input, db_in, state_in, RESULT.#1,
                                RESULT.#2, RESULT.#3);
```

図 7　ATM_引き出し仕様_実行の定義

表 2　カバレッジ計測結果

プロパティ群番号	C0	C1
1	26.32%	16.67%
1+2	38.60%	16.67%
4	96.49%	100.00%

テストすることを指す．プロパティを満たされていることを 100 回確認できた場合，テストが成功したことを指す．プロパティを満たさない反例が示されたとき，テストが失敗したことを指す．それ以外でテストが終了したとき，テストが成功したとも失敗したともいえない状態となるため，テストコードや実装コードの改善が必要となる．テスト実行後の実装コードのステートメントカバレッジ (C0) とブランチカバレッジ (C1) を sbt scoverage によって計測した．計測結果を**表 2** に示す．

プロパティ群 1 でテストしたところ，C0 が 26.32%，C1 が 16.67%となった．プロパティ群 1 では，事前条件を持つ関数についてのプロパティにおいて，事前条件を満たさない値が入力されたときに例外が発生し，テストが終了した．プロパティ群 1 の，事前条件をもつ関数についてのプロパティをプロパティ群 2 のプロパティに変更してテストしたところ，C0 が 38.60%，C1 が 16.67%となった．プロパティ群 3 では，例外が発生してテストが終了するだけとなったため，カバレッジは省略する．プロパティ群 4 は，ATM_引き出し仕様_実行をテストするときのプロパティとなるため，他のプロパティ群とは組み合わせずにテストを実行した．テスト実行の結果，C0 が 96.49%，C1 が 100.00%となった．

6.2　考察

プロパティ群 1 やプロパティ群 2 で用いているようなジェネレータでは，条件分岐において，特定の条件に合致するものばかりが生成されてしまうため，カバレッジが低くなってしまっている．例えば，アカウント情報データベースにアカウント情報が存在しないキャッシュカードが挿入されたときが挙げられる．アカウント情報データベースのデータもキャッシュカードのデータも独立にランダムに生成されるため，例のような場合が多く起きる．

プロパティ群 3 では，呼び出されたメソッドが例外を発生するというプロパティを作成し，テストを行ったが，メソッドの実行中に例外が発生するのであって，メソッドの終了後に出力として例外を発するわけではないため，プロパティを満たすか否かのテストができていなかった．

プロパティ群 4 では，入力変数に関する同値分割を行い，各同値クラスを用いているため，全ての分岐をテストできている．C0 が 100%になっていない理由は，仕様に記述されていない内容が実装に含まれていたためであった．ただし，同値分割を行った結果を用いる場合，各同値クラスから一つずつデータを生成しテストを行った場合とテストの結果が変わらない．今回の仕様では，同値クラスの明確な境界がなかったため，境界値分析を行えなかった．明確な境界を持つ同値クラス群において，複数の変数に関する境界値を生成するジェネレータを作成できると考えられる．

7.　まとめ

本稿では，形式仕様記述をプロパティベーステストに用いることで，テスト実装・実行の自動化支援を行う手法を提案した．提案手法の予備評価のために，ATM の引き出しの例を用い，自動的に生成できると考えられるジェネレータ・プロパティを用いてプロパティベーステストを行った．今回用いた例は，簡単化したものであったため，詳細化した場合に，より有用な結果を得られないかを考察する必要がある．また，今回の事例研究では，仕様の型と実装の型の対応は分かっているという前提で行った．この前提に関する今後の課題として，仕様の型から実装の型との対応を渡すことで，仕様の型のジェネレータから生成された値を，実装の型の値に変換してプロパティベーステストに用いることができないかを考察する．

参考文献

[1] Hierons, R.M., Bogdanov, K., Bowen, J.P. *et al.* :Using Formal Specifications to Support Testing, ACM Computing Surveys, ACM (2009)
[2] Carlier, M., Dubois, C., and Gotlieb, A.: FocalTest: A Constraint Programming Approach for Property-Based Testing, Software and Data Technologies, Springer Berlin Heidelberg, pp. 140-155(2013)
[3] ディペンダブル・システムのための形式手法の実践ポータル, http://formal.mri.co.jp/ (2016.01.21 アクセス)
[4] ASTER テストツール WG: テストツールまるわかりガイド (入門編)Version 1.0.0, ソフトウェアテスト技術振興協会 (2012)
[5] TERASOLUNA RACTES ｜プロダクト｜ TERASOLUNA, http://www.terasoluna.jp/product/tool/ractes.html (2016.02.09 アクセス)
[6] Havelund, K.: Closing the Gap Between Specification and Programming: VDM++ and Scala, In Higher-Order Workshop on Automated Runtime Verification and Debugging (2011).
[7] Dick, J. and Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-based Specifications, FME ‘93: Industrial-Strength Formal Methods Lecture Notes in Computer Science Volume 670, pp 268-284(1993)

[8] 中津川泰正: VDM「仕様」の書き方,
http://aofa.csce.kyushu-u.ac.jp/documents/Nakatsugawa-2013-12.pdf (2016.02.01 アクセス)