宮崎大学大学院

修士学位論文

VDM++ 仕様を対象としたテストケース自動生成ツール BWDM におけるペアワイズ法とドメイン分析テストの適用のための機能拡張

2020 年 **1** 月

宮崎大学大学院 工学研究科 修士課程

工学専攻 機械・情報系コース 情報システム工学分野

平木場 風太

目次

概要

社会におけるソフトウェアの重要性は高まっており、ソフトウェアのバグが社会にもたらす影響も近年では甚大なものとなっている。ソフトウェアにバグが混入する原因の 1 つとして、上流工程のソフトウェア設計段階において、自然言語を一般的に用いていることが挙げられる。自然言語は元来、曖昧さを含んでいる。そのため、プログラマが、仕様書上の表記を、仕様書の作成者が本来意図していない意味で捉えてしまうことが起こる。プログラマが、仕様書の本来の意図から外れた認識に基づいて実装を行った結果、ソフトウェアにバグが混入されてしまう。

この問題を解決するための 1 つの方法として、形式手法を用いた上流工程でのソフトウェア設計が挙げられる。また、形式仕様記述言語の 1 つに VDM++(Vienna Development Method) がある。近年、形式仕様記述言語を用いた仕様記述が重要視されてきている。

一方で、作成したソフトウェアにはテストが必要であるが、人手によるテストケースの設計には手間と時間がかかる。そのため、VDM++ 仕様を対象としたテストケース自動生成を目的としたツール BWDM(Boundary Value/Vienna Develop Method) を立山らが開発した。

既存の BWDM は、VDM++ 仕様を入力として、境界値分析と記号実行を行い、テストケースを自動生成する。自動生成したテストケースによって、境界値テストと、if-then-else 式の構造認識に基づいたテストを実施できる。

しかし、既存の BWDM には、以下の 5 つの問題点がある。

(a) 境界値分析時にすべての組合せを用いてテストケースを生成するため、組合せ爆発を起こす可能性がある

(b) 複数の変数を含む条件式を持つ関数のテストケースを生成できない

(c) 1 つの関数しかテストケースを生成できない

(d) 定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できない

(e) 操作のテストケースを生成できない

そこで、本研究では、BWDM の有用性の向上を目的として、BWDM の拡張を行う。具体的には、上記 5 つの問題点を解決する。また、拡張後の BWDM の名称を、BWDM(Verification tool for Vienna Development Method) に変更する。

まず、本研究では、既存の BWDM における、境界値分析結果から生成するテストケース数が組合せ爆発を起こす可能性の排除を目的として、BWDM の拡張を行う。拡張した BWDM は、VDM++ 仕様に対して境界値分析を行い、ペアワイズ法を適用し、テストケースを自動生成する。これにより、境界値分析結果から生成するテストケース数が組合せ爆発を起こす可能性を排除することができたため、テスト工程の作業効率化を見込めると考えられる。

次に、本研究では、既存の BWDM における、複数の変数を含む条件式を持つ仕様に対してテストケース生成ができないという問題の解決を目的として、ドメイン分析テストのためのテストケース生成を、BWDM に適用する拡張を行う。まず、ドメイン分析部を追加し、on ポイント、off ポイント、in ポイント、out ポイントを生成できるようにする。また、SMT ソルバを用いて、充足可能性問題を解けるようにする。次に、テストケースに、正常系判定値、着目条件式、そして着目変数の情報を含めるために、テストケース生成部を拡張する。拡張した BWDM は、複数の変数を含む条件式を持つ VDM++ 仕様の構文解析と、ドメイン分析テストのためのテストケース生成ができるようになったことを確認した。また、ドメイン分析テストのためのテストケース生成に要した時間を人手と比較検証した結果、18 分程の時間短縮を確認できた。

加えて、本研究では、既存の BWDM における、1 つの関数しかテストケースを生成できないという問題、定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できないという問題、操作のテストケースを生成できないという問題の解決を目的として、BWDM における構文解析部の抽象構文木解析処理を修正し、BWDM を拡張する。これにより、BWDM の対応範囲が広がったため、より多くの VDM++ 仕様のテストケース生成が可能となった。

以上により、本研究で行った拡張によって、既存の BWDM の持つ 5 つの問題点を解決し、BWDM の有用性が向上したと言える。また、BWDM がテストケースを自動生成できる VDM++ 仕様の構文が増えたことにより、テストケース作成作業の自動化率が向上するため、テスト工程の作業効率が向上したと言える。

第 1 章

はじめに

社会におけるソフトウェアの重要性は高まっており、ソフトウェアのバグが社会にもたらす影響も近年では甚大なものとなっている [1]。ソフトウェアにバグが混入する原因の 1 つとして、上流工程のソフトウェア設計段階において、自然言語を一般的に用いていることが挙げられる [2]。自然言語は元来、曖昧さを含んでいる。そのため、プログラマが、仕様書上の表記を、仕様書の作成者が本来意図していない意味で捉えてしまうことが起こる。プログラマが、仕様書の本来の意図から外れた認識に基づいて実装を行った結果、ソフトウェアにバグが混入されてしまう。

この問題を解決するための 1 つの方法として、形式手法 (Formal Methods)[3] を用いた上流工程でのソフトウェア設計が挙げられる。また、形式仕様記述言語の 1 つに VDM++(Vienna Development Method) がある [4]。近年、形式仕様記述言語を用いた仕様記述が重要視されてきている [2]。

一方で、作成したソフトウェアにはテストが必要であるが、人手によるテストケースの設計には手間と時間がかかる。そのため、VDM++ 仕様を対象としたテストケース自動生成を目的としたツール BWDM(Boundary Value/Vienna Develop Method) を、立山らが開発した [5, 6]。

既存の BWDM は、VDM++ 仕様を入力として、境界値分析と記号実行を行い、テストケースを自動生成する。自動生成したテストケースによって、境界値テストと、if-then-else 式の構造認識に基づいたテストを実施できる。

しかし、既存の BWDM には、以下の 5 つの問題点がある。

(a) 境界値分析時にすべての組合せを用いてテストケースを生成するため、組合せ爆発を起こす可能性がある

(b) 複数の変数を含む条件式を持つ関数のテストケースを生成できない

(c) 1 つの関数しかテストケースを生成できない

(d) 定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できない

(e) 操作のテストケースを生成できない

そこで、本研究では、BWDM の有用性の向上を目的として、BWDM の拡張を行う [7, 8, 9, 10, 11, 12]。具体的には、既存の BWDM が持つ、上記 5 つの問題点を解決する。また、拡張後の BWDM の名称を、BWDM(Verification tool for Vienna Development Method) に変更する。

まず、本研究では、BWDM が組合せ爆発を起こす可能性 (問題 (a)) の排除を目的として、BWDM によるテストケース生成にペアワイズ法を適用し、BWDM を拡張する。ペアワイズ法とは、組合せテストの総数を減らした上で、効果的なテストを行う方法である [13]。ソフトウェアについて、3 つ以上の因子の組合せで発生する欠陥は、ほとんど存在しないことが知られている [14]。したがって、組合せテストは 2 つの因子の組合せに対して効果的である。(3 つ以上の因子で発生している欠陥を見つけるには、他の方法でテストする必要がある。) そして、2 つの因子のみの組合せをテストする手法のことをペアワイズ法と呼ぶ。ペアワイズ法を適用するに当たって、Microsoft 社が開発した PICT(Pairwise Independent Combinatorial Testing tool)[15] を利用する。PICT は CLI(Command Line Interface) ツールであるが、API(PICT ライブラリと呼称する) も提供しており、C++ から利用できる。しかし、既存の BWDM は Java で記述しており、PICT ライブラリを呼び出すことができない。そこで、PICT と BWDM を接続するためのインタフェースとして、pict4java を開発する [16]。そして、既存の BWDM に pict4java を埋め込むことで、BWDM を拡張する。

次に、本研究では、既存の BWDM における、複数の変数を含む条件式を持つ仕様に対してテストケース生成ができないという問題 (問題 (b)) の解決を目的として、BWDM によるテストケース生成にドメイン分析テストのためのテストケース生成手法を適用し、BWDM を拡張する。ドメイン分析テストとは、境界値分析において、複数の変数を含む条件式を持つ関数をテストす

る方法である [17][18]。ここで、本研究における、複数の変数を含む条件式とは、片方の辺にのみ複数の変数が含まれ、もう片方の辺は定数のみの条件式を指すこととする。すなわち、両辺に変数を含む条件式を持つ仕様は、本研究の対象としない。

加えて、本研究では、既存の BWDM における、1 つの関数しかテストケースを生成できないという問題 (問題 (c))、定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できないという問題 (問題 (d))、操作のテストケースを生成できないという問題 (問題 (e)) の解決を目的として、BWDM における構文解析部の抽象構文木解析処理を修正し、BWDM を拡張する。

以下、本論文の構成は次のとおりである。

第 2 章では、BWDM を実装するために必要となる前提知識について説明する。

第 3 章では、既存の BWDM について説明する。

第 4 章では、拡張した BWDM について説明する。

第 5 章では、適用例を用いて、拡張した BWDM が正しく動作することを検証する。

第 6 章では、拡張した BWDM について考察する。

第 7 章では、本研究のまとめと今後の課題を示す。

第 2 章

研究の準備

本章では、BWDM を拡張するにあたり、必要となる前提知識を説明する。

2.1 VDM++

形式手法の 1 つに VDM(Vienna Development Method)[19] がある。1970 年代にウィーンの IBM 研究所で考案され、1990 年代前半にかけて開発された。1996 年には形式仕様記述言語 VDM-SL が ISO 標準となっている [20]。VDM++ は、VDM-SL にオブジェクト指向拡張を施した形式仕様記述言語である。本研究で拡張する BWDM はこの VDM++ で記述された仕様のテストケースを生成する。また、本研究では、Overture Project が発行している VDM-10 Language Manual で定義されている VDM++ を対象とする [4]。

VDM には、VDMTools[21] や Overture IDE[22] などの支援ツールが揃っており、仕様の検証などを他の形式手法に比べ比較的行いやすいという利点がある。

VDM++ では、関数や操作の定義を定義ブロック (`definition block`) で行う。それぞれの定義と、各定義に対応する日本語での呼び方を、表 2.1 に示す。

また、本研究で拡張する BWDM は、表 2.1 における、定数定義、操作定義、インスタンス変数定義に、新たに対応する。詳細を、以下に示す。

- 定数定義

  定数定義は、プログラミングにおける定数を表す。定義した定数は、関数定義、操作定義で利用できる。拡張する BWDM は、定数定義に対応することで、関数定義、または、操

表 2.1: VDM++ における定義および各定義に対応する日本語での呼び方

VDM++ における定義	日本語での呼び方
type definitions	型定義
state definition	状態定義
value definitions	定数定義
function definitions	関数定義
operation definitions	操作定義
instance variable definitions	インスタンス変数定義
synchronization definitions	同期定義
thread definitions	スレッド定義
traces definitions	トレース定義

作定義内で定数を利用できる。ただし、本研究で定義の対象とする定数の型は、int、nat、nat1 のいずれかとする。

- 操作定義

  操作定義は、プログラミングにおけるメソッドを表す。定義した操作は、同クラス内のインスタンス変数を参照できる。拡張する BWDM は、操作定義に対応することで、if-then-else 式の構造認識に基づいたテストを実施するためのテストケースを自動生成する。ただし、本研究で定義の対象とする操作内で使用する型は、int、nat、nat1 と sec of char(戻り値のみ) のいずれかとする。また、事前条件と事後条件には対応しない。

- インスタンス変数定義

  インスタンス変数定義は、プログラミングにおけるフィールドを表す。定義した定数は、関数定義、操作定義で利用できる。拡張する BWDM は、インスタンス変数定義に対応することで、操作定義内でインスタンス変数を利用できる。ただし、本研究で定義の対象とするインスタンス変数の型は、int、nat、nat1 のいずれかとする。

表 2.2: VDM++ における定義および定義に対応する VDMJ での呼び方

VDM++ における定義	VDMJ での呼び方
関数定義	explicit function
操作定義	explicit operation
定数定義	value
インスタンス変数定義	instance variable

2.2 VDMJ

VDMJ[23] は、Nick Battle 氏が開発している VDM の支援ツールである。GNU GPL3 ライセンスでソースコードを公開している VDM 支援ツールであり、構文解析器、型チェッカー、デバッガなどの機能を持つ。VDM を用いた開発を支援する IDE である Overture Tool[22] の内部にも用いられている。

本研究では、VDM++ 仕様の静的解析において、VDMJ の字句解析機能と構文解析機能を利用する。

また、本研究で利用する VDMJ のクラスを、以下に示す。

- 抽象クラス `VDMJ::TCDefinition`

  VDM++ における各定義ブロックを表す抽象クラスである。フィールド `name` の値は定義名を示す。メソッド `kind()` を実行することで、定義の種類を取得できる。VDM++ における定義の種類を、表 2.2 に示す。

- クラス `VDMJ::TCExplicitFunctionDefinition`

  VDM++ における関数定義ブロック functions 内にある定義を表すクラスである。

- クラス `VDMJ::TCExplicitOperationDefinition`

  VDM++ における操作定義ブロック operations にある定義を表すクラスである。

- クラス `VDMJ::TCValueDefinition`

  VDM++ における定数定義ブロック values にある定義を表すクラスである。

表 2.3: 性別と年齢を入力として出力が決定される関数の因子と水準

因子	水準	因子の取り得る値
性別	3	男性
		女性
		その他
年齢	2	20 歳未満
		20 歳以上
国籍	3	日本
		アメリカ
		その他

- クラス `VDMJ::TCInstanceVariableDefinition`

  VDM++ におけるインスタンス変数定義ブロック instance variables にある定義を表すクラスである。

2.3 因子と水準

因子とは、テスト対称に影響を与える要因である。本研究においては、関数、または、操作内の変数を意味する。水準とは、因子の取り得る値の数である。本研究においては、関数、または、操作内の変数が取り得る値の数を意味する [24]。

例えば、性別 {“男性”, “女性”, “その他”} と年齢 {“20 歳未満”, “20 歳以上”}、国籍 {“日本”, “アメリカ”, “その他”} を入力として出力が決定される関数の場合の因子と水準を、表 2.3 に示す。

2.4 ペアワイズ法

ペアワイズ法とは、組合せテストの総数を減らした上で、効果的なテストを行う方法である [13]。ソフトウェアについて、3 つ以上の因子の組合せで発生する欠陥は、ほとんど存在しないことが知られている [14]。したがって、組合せテストは 2 つの因子の組合せに対して効果的であ

表 2.4: 性別と年齢と国籍を入力として出力が決定される関数のペアワイズ法を用いた場合の組合せの例

性別	年齢	国籍
女性	20 歳以上	日本
その他	20 歳未満	日本
女性	20 歳未満	その他
その他	20 歳以上	その他
男性	20 歳以上	アメリカ
女性	20 歳未満	アメリカ
その他	20 歳未満	アメリカ
男性	20 歳未満	日本
男性	20 歳以上	その他

る。(3 つ以上の因子で発生している欠陥を見つけるには、他の方法でテストする必要がある。) 狭義においては、2 つの因子のみの組合せをテストする手法のことをペアワイズ法と呼ぶ。

ペアワイズ法の例を示す。2.3 節の例の関数は、因子が 3 で水準が (3, 2, 3) である。この関数の総組合せでテストケースを作成した場合、全組合せ総数は $3 \times 2 \times 3 = 18$ 個 となる。これに対して、ペアワイズ法を用いてテストケースを作成した場合のテストケース例を、表 2.4 に示す。表 2.4 から、テストケースの数を 9 個程度に抑えることができることが確認できる。

2.5 ドメイン分析テスト

本研究における、ドメイン分析テストの定義と、ドメインテストが必要となる仕様の例を、以下で示す。

2.5.1 ドメイン分析テストの定義

本研究におけるドメイン分析テストとは、関係性がある複数の変数を同時にテストする方法である [17][18]。ドメインとは、入力するデータの定義域である。ドメイン分析とは、入力する変

数と条件式を分析し、ドメインを抽出することである。ドメイン分析テストでは、ドメインごとに on ポイント、off ポイント、in ポイント、out ポイントと呼ばれる入力値、および、それを元にしたテストケースを作成し、テストする。境界値とは、同値分割した領域の端、あるいは端のどちらか側で最小の増加的距離にある入力値、または、出力値である [18]。本研究では、境界値の中でも、条件式を満たす境界値を利用する。これを、TB(True Boundary) と命名する。TB の定義を、以下に示す。ここで、本研究における、複数の変数を含む条件式とは、片方の辺にのみ複数の変数が含まれ、もう片方の辺は定数のみの条件式を指すこととする。すなわち、両辺に変数を含む条件式を持つ仕様は、本研究の対象としない。

- *condition* が $exp = int$ のとき、$TB = int$ とする。
- *condition* が $exp < int$ のとき、$TB = int - 1$ とする。
- *condition* が $exp > int$ のとき、$TB = int + 1$ とする。
- *condition* が $exp <= int$ のとき、$TB = int$ とする。
- *condition* が $exp >= int$ のとき、$TB = int$ とする。

なお、*condition* を条件式、*exp* を左辺、*int* を右辺とし、*condition* は、*exp operator*{=, <, >, <= , >=} *int* の形式でなければならない。また、*int* は、整数でなければならない。

それぞれのポイントの定義を、以下に示す。

- on ポイント : 着目条件式 (後述) の TB である。ドメインを決定づける条件式に付き 1 つ生成する。他の on ポイントと重複してはならない。*targetExp* を着目条件式とすると、on ポイントは、$targetExp = True$ かつ $exp = TB$ となる値でなければならない。
- off ポイント : 着目する on ポイントに隣接し、TB でない値である。on ポイントに付き複数 (着目条件式に含まれる変数 $* 2$) 個存在する。*targetVar* を着目変数 (後述) とすると、off ポイントは、$targetVar + 1$ となる値、または、$targetVar - 1$ となる値でなければならない。
- in ポイント : ドメインを決定づけるすべての条件式を満たす値である。ドメインに付き 1 つ生成する。on ポイントや off ポイントと重複してはならない。*onPoints* を on ポイントの集合とし、$offPoints$ を off ポイントの集合、*inPoint* を in ポイントとすると、in ポイントは、

($condition1 \wedge condition2 \wedge ... \wedge conditionN$) = $True$ かつ $inPoint \notin (onPoints \cup offPoints)$ となる値でなければならない。

- out ポイント：着目条件式のみを満たさない値である。ドメインを決定づける条件式に付き 1 つ生成する。off ポイントと重複してはならない。*outPoint* を out ポイントとすると、out ポイントは、($\neg targetExp \wedge condition1 \wedge condition2 \wedge ... \wedge conditionN$) = $True$ かつ $outPoint \notin offPoints$ となる値でなければならない。

また、それぞれのポイントは、以下のパラメータを持つ。

- 正常系判定値
  - “正常系” とは、ポイントの期待出力がドメインの期待出力と一致する状態のことを言う。
  - “非正常系” とは、ポイントの期待出力がドメインの期待出力と一致しない状態のことを言う。
  - “正常系判定値” とは、正常系であるかどうかを保持する値である。正常系か非正常系かの 2 つの状態を持つ。
- 着目条件式

  on ポイント、off ポイント、out ポイントのみが持つ。どの条件式に着目してポイントを生成したかの情報である。
- 着目変数

  off ポイントのみが持つ。どの変数に着目して、on ポイントに隣接するポイントを生成したかの情報である。

2.5.2 ドメイン分析テストの例

ドメイン分析テストが必要となる仕様の例として、遊園地チケット割引機能をテストすることを考える。遊園地チケット割引機能は、夫婦である夫と妻それぞれの年齢を入力とし、割引価格が適用されるかどうかを判定する関数であり、以下のルールを持つ。

コード 2.1: ドメインテストが必要となる仕様 (遊園地チケット割引機能)

```
1  class 遊園地チケット
2
3  functions
4
5  static public 割引判定 : int * int -> seq of char
6    割引判定(夫の年齢, 妻の年齢) ==
7      if(夫の年齢 + 妻の年齢 <= 50) then
8        if(夫の年齢 >= 18) then
9          if(妻の年齢 >= 16) then
10           ''割引価格となる''
11         else
12           ''割引価格とならない(妻の年齢 $<$ 16)''
13       else
14         ''割引価格とならない(夫の年齢 $<$ 18)''
15     else
16       ''割引価格とならない(夫の年齢 + 妻の年齢 > 50)'';
17
18 end 遊園地チケット
```

A) 以下の条件をすべて満たすとき、遊園地チケットは割引価格となる。

- 夫の年齢と妻の年齢の合計が 50 歳以下である。
- 夫の年齢は 18 歳以上である。
- 妻の年齢は 16 歳以上である。

B) A) でない場合、遊園地チケットは割引価格とならない。

この仕様を VDM++ で記述した仕様を、コード 2.1 に示す。7 行目に、複数の変数を左辺に含む条件式があるため、既存の BWDM ではテストケース生成ができない。また、“割引価格となる” というドメインの各ポイントを生成した例を、図 2.1 に示す。ドメインを決定づける 3 つの条件式は、“夫の年齢 + 妻の年齢 <= 50”、“夫の年齢 >= 18”、“妻の年齢 >= 16” である。on ポイントは、“On1”～“On3” の 3 つであり、それぞれ、条件式の境界線上に存在する。off ポイントは、“Off11”～“Off32” の 8 つである。“Off11”～“Off14” は “夫の年齢 + 妻の年齢 <= 50” という条件式に着目した “On1” に隣接する off ポイントであり、4 つ存在する。これは、“夫の年齢” を正負の方向にそれぞれずらした off ポイントが 2 つ存在し、同じように、“妻の年齢” を正負の方向にそれぞれずらした off ポイントが 2 つ存在するためである。しかし、“On2” の off ポイントは “Off21”、 “Off22” の 2 つのみである。これは、“夫の年齢” をずらした off ポイントが “妻の年齢 >= 16” となり、TB となるからである。in ポイントは、“In” の 1 つであり、“割引価格となる”

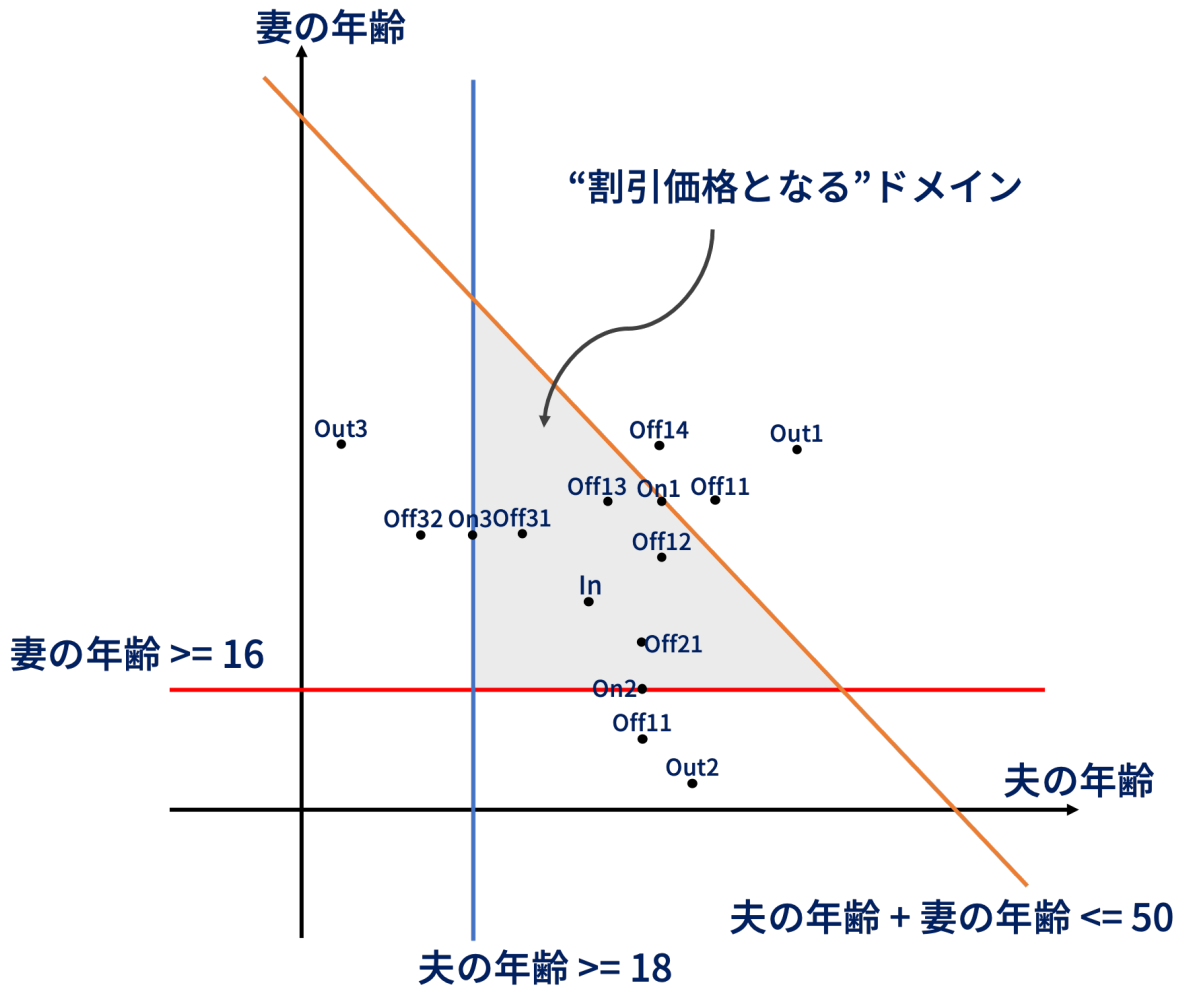

図 2.1: 遊園地チケット割引機能 (コード 2.1) の割引になる条件にドメイン分析を適用した例

ドメインのすべての条件式を満たす値を持つ。out ポイントは、“Out1”〜“Out3” の 3 つであり、それぞれの着目条件式のみを満たさない値を持つ。

第 3 章

既存の BWDM

本章では、既存の BWDM(Boundary Value/Vienna Develop Method) について説明する。

3.1 機能

既存の BWDM(Boundary Value/Vienna Develop Method) は、立山氏が開発した、VDM++ 仕様を対象としたテストケース生成ツールである。関数定義に含まれる、if-then-else 式の構造認識に基づいたテストを実施するためのテストケースを自動生成する [6]。

既存の BWDM には、以下の機能がある。

- 記号実行によるテストケース生成
- 境界値分析によるテストケース生成

記号実行によって生成したテストケースは、すべての実行フローを網羅できることが期待できる。境界値分析によって生成したテストケースは、境界値テストに使用することができる。既存の BWDM を使用することにより、VDM++ 仕様を用いたソフトウェア開発効率を改善できる。

既存の BWDM の構造を、図 3.1 に示す。

3.2 入出力例

3 つの引数 (a, b, c) を入力とする VDM++ 仕様ファイルの例をコード 3.1 に、このファイルから生成したテストケースの例をコード 3.2 に、それぞれ示す。また、テストケースの出力フォー

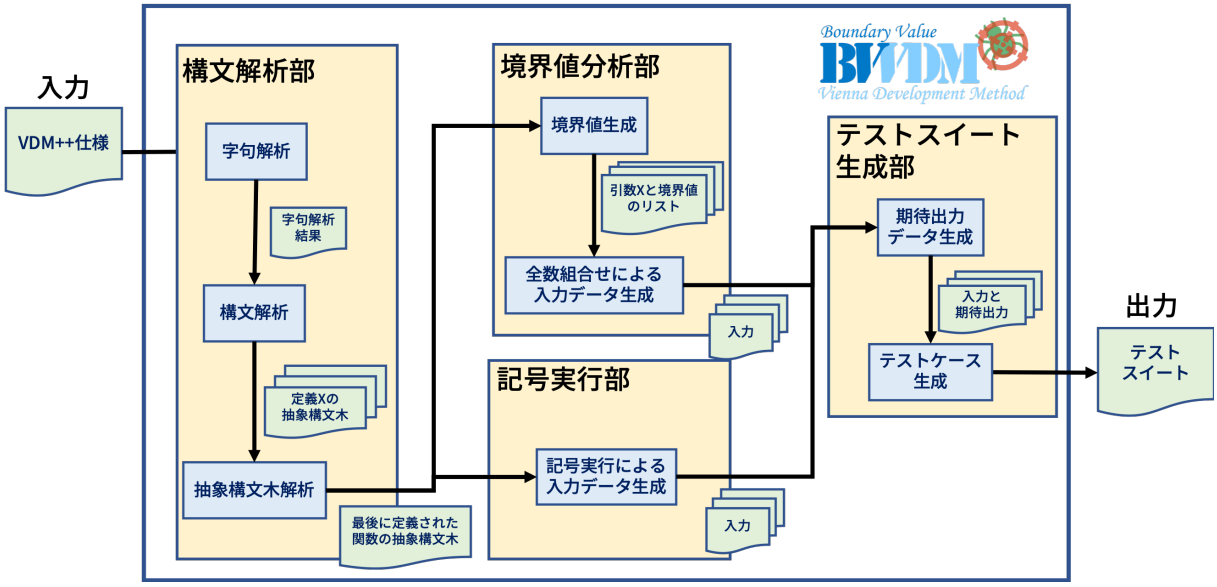

図 3.1: 既存の BWDM の構造

コード 3.1: VDM++ 仕様の例

```
1  class SampleClass
2
3  functions
4
5  sampleFunction : int*nat*nat -> seq of char
6    sampleFunction(a, b, c) ==
7      if(a < 100) then
8        if(b > 2018) then
9          "a は 100 未満かつ b は 2018 より大きい"
10       else
11         "a は 100 未満かつ b は 2018 以下"
12     elseif(c < 12) then
13       "a は 100 以上かつ c は 12 未満"
14     else
15       "a は 100 以上かつ c は 12 以上";
16
17 end SampleClass
```

コード 3.2: 既存の BWDM のテストケース出力例 (コード 3.1 を適用)

```
1
2  関数名 : sampleFunction
3  引数の型 : a:int b:nat c:nat
4  戻り値の型 : seq of (char)
5  生成テストケース数 : 220件 (境界値分析:216/記号実行:4)
6
7  各引数の境界値
8  a : 2147483648 2147483647 -2147483648 -2147483649 99 100
9  b : 4294967295 4294967294 0 -1 2019 2018
10 c : 4294967295 4294967294 0 -1 11 12
11
12 記号実行情報
13 戻り値の数 : 4
14 制約 : b > 2018 and a < 100, 戻り値 : "a は 100 未満かつ b は 2018 より大きい"
15 制約 : !( b > 2018 ) and a < 100, 戻り値 : "a は 100 未満かつ b は 2018 以下"
16 制約 : c < 12 and !( a < 100 ) , 戻り値 : "a は 100 以上かつ c は 12 未満"
17 制約 : !( c < 12 ) and !( a < 100 ) , 戻り値 : "a は 100 以上かつ c は 12 以上"
18
19 境界値分析によるテストケース
20 No.1 : 2147483648 4294967295 4294967295 -> Undefined Action
21 No.2 : 2147483647 4294967295 4294967295 -> Undefined Action
22 No.3 : -2147483648 4294967295 4294967295 -> Undefined Action
23 No.4 : -2147483649 4294967295 4294967295 -> Undefined Action
24 No.5 : 99 4294967295 4294967295 -> Undefined Action
25
26 ~~~中略~~~
27
28 No.212 : 2147483647 2018 12 -> "a は 100 以上かつ c は 12 以上"
29 No.213 : -2147483648 2018 12 -> "a は 100 未満かつ b は 2018 以下"
30 No.214 : -2147483649 2018 12 -> Undefined Action
31 No.215 : 99 2018 12 -> "a は 100 未満かつ b は 2018 以下"
32 No.216 : 100 2018 12 -> "a は 100 以上かつ c は 12 以上"
33
34 記号実行によるテストケース
35 No.1 : 99 2019 1 -> "a は 100 未満かつ b は 2018 より大きい"
36 No.2 : 99 2018 1 -> "a は 100 未満かつ b は 2018 以下"
37 No.3 : 100 b 11 -> "a は 100 以上かつ c は 12 未満"
38 No.4 : 100 b 12 -> "a は 100 以上かつ c は 12 以上"
```

コード 3.3: 既存の BWDM のテストケースの出力フォーマット

```
1
2  各引数の境界値
3  <引数 1>: <入力値 1> ... <入力値N>
4  ・
5  ・
6  ・
7  <引数N>: <入力値 1> ... <入力値N>
8
9  境界値分析によるテストケース(ペアワイズ法適用)
10 <ID> : <入力値 1> ... <入力値N> -> <期待出力>
11
12 記号実行によるテストケース
13 <ID> : <入力値 1> ... <入力値N> -> <期待出力>
```

マットを、コード 3.3 に示す。コード 3.2 において、“各引数の境界値” には、引数 a、b、c それぞれの境界値の集合を出力している。そして、“境界値分析によるテストケース” には、各引数の境界値の総組合せのテストスイートを出力している。また、“記号実行によるテストケース” には、関数のすべての実行フローを網羅できるテストスイートを出力している。

3.3 問題点

既存の BWDM には 5 つの問題点がある。問題点を、以下で説明する。

3.3.1 組合せ爆発に関する問題点

境界値分析によるテストケース生成において、生成するテストケース数は、因子が取り得るそれぞれの値の数 (水準) を掛け合わせることにより決定する。たとえば、因子が 6 で水準が (6, 6, 2, 4, 5, 7) の場合、既存の BWDM は、$6\times6\times2\times4\times5\times7 = 10,080$ 個 のテストケースを生成する。したがって、組合せ爆発を起こす可能性があるという問題がある。

本研究では、組合せ爆発を起こす可能性があるという問題を解決するために、テストケース生成時にペアワイズ法を適用することで、BWDM を拡張する。

3.3.2 複数変数を含む条件式に関する問題点

既存の BWDM には、左辺に複数の変数を含む条件式を持つ仕様のテストケースを生成できないという問題がある。コード 2.1 に示した VDM++ 仕様には、7 行目の条件式の左辺に複数の変数が含まれているため、この仕様を既存の BWDM に適用しても、テストケースを生成できない。

この理由を、既存の BWDM に コード 2.1 の仕様を入力した場合を例に、以下で説明する。

- コード 2.1 の 6 行目にて、入力する 2 つの変数は “夫の年齢” と “妻の年齢” と定義している。7 行目の if 条件式は “夫の年齢” と “妻の年齢” の 2 つの変数を利用しているが、既存の BWDM は “夫の年齢 + 妻の年齢” という 1 つの変数だと解釈する。“夫の年齢 + 妻の年齢” という引数は仕様に定義されていないため、既存の BWDM はエラーを出力し、動作を停止してしまう。

- コード 2.1 の 7 行目の “夫の年齢 + 妻の年齢 <=50” という条件式の TB を求めるには、「制

約を満たす入力があるかどうか」という、充足可能性問題 (Satisfiable Problem、SAT)[25] を解かなければならない。しかし、既存の BWDM は、充足可能性問題を解くことができない。

本研究では、左辺に複数の変数を含む条件式を持つ仕様のテストケース生成ができないという問題を解決するために、ドメイン分析テストのためのテストケース生成手法を提案し、既存の BWDM に適用することで、BWDM を拡張する。

3.3.3 1 つの関数しかテストケースを生成できない問題点

既存の BWDM には、1 つの関数しかテストケースを生成できないという問題がある。したがって、複数の関数を含む仕様のテストケースを生成した際に、最後に定義した関数のテストケースしか出力できない。複数の関数を含む VDM++ 仕様を、コード 3.4 に示す。この VDM++ 仕様には「うるう年判定」関数と「成人判定」関数が定義されている。コード 3.4 を既存の BWDM に適用した際の出力を、コード 3.5 に示す。

既存の BWDM では、図 3.1 の構文解析処理で定義ごとの抽象構文木を生成し、それらを抽象構文木解析処理に渡す。抽象構文木解析処理では、受け取った抽象構文木の集合から、関数定義だけを抽出する。受け取る抽象構文木の集合は、2.2 節で示した抽象クラス `TCDefinition` を継承したクラスの集合である。既存の BWDM の抽象構文木解析処理を、以下に示す。

1. `TCExplicitFunctionDefinition` 型の変数 `tcFunctionDefinition` を宣言する。
2. 構文解析処理で生成した抽象構文木の集合が空であるか確認する。空でない場合、3. へ進む。空である場合、変数 `tcFunctionDefinition` を構文木解析処理の出力とし、処理を終了する。
3. 抽象構文木の集合の先頭を取り出し、変数 `astDefinition` に格納する。
4. `astDefinition.kind()` を実行し、定義の種類を取得する。
5. 4. で取得した定義の種類 (表 2.2 参照) が “explicit function” であるか確認する。“explicit function” である場合、6. へ進む。“explicit function” でない場合、2. へ進む。

コード 3.4: 複数の関数を含む VDM++ 仕様

```
class 判定

functions

public うるう年判定 : int -> seq of char
  うるう年判定(年) ==
    if(年 mod 4 = 0) then
      if(年 mod 100 = 0) then
        if(年 mod 400 = 0) then
          "うるう年"
        else
          "平年"
      else
        "うるう年"
    else
      "平年";

public 成人判定 : nat -> seq of char
  成人判定(年齢) ==
    if(年齢 <= 20) then
      "未成年"
    else
      "成人";

end 判定
```

6. 変数 astDefinition を TCExplicitFunctionDefinition に型変換し、変数 tcFunctionDefinition に代入する。その後、2. に進む。

このように、抽象構文木解析処理では、受け取った定義の集合から、関数定義のみを抽出する。抽出した関数定義ごとに、TCExplicitFunctionDefinition 型の変数に関数定義を代入するが、この変数が 1 つしか用意されていないため、最終的に、VDM++ 仕様で最後に定義された関数のみが抽象構文木解析処理の出力となってしまう。したがって、既存の BWDM は、1 つの関数しかテストケース生成ができない。

本研究では、1 つの関数しかテストケースを生成できないという問題を解決するために、既存の BWDM の構文解析処理を修正し、複数の関数のテストケース生成に対応し、BWDM を拡張する。

3.3.4 定数定義ブロック内の定義を含む関数のテストケースを生成できない問題点

既存の BWDM には、定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できないという問題点がある。定数定義ブロック内の定義を含む関数を含む VDM++ 仕様を、コード 3.6 に示す。この VDM++ 仕様には「偶数判定」関数が定義されている。また、定数

コード 3.5: 複数の関数を含む VDM++ 仕様 (コード 3.4) を既存の BWDM に適用した際の出力

```
関数名 : 成人判定
引数の型 : 年齢:nat
戻り値の型 : seq of (char)
生成テストケース数 : 8件 (境界値分析:6/記号実行:2)

各引数の境界値
年齢 : 4294967295 4294967294 0 -1 20 21

記号実行情報
戻り値の数 : 2
制約 : 年齢 <= 20 , 戻り値 : "未成年"
制約 : !( 年齢 <= 20 ) , 戻り値 : "成人"

境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> "成人"
No.3 : 0 -> "未成年"
No.4 : -1 -> Undefined Action
No.5 : 20 -> "未成年"
No.6 : 21 -> "成人"

記号実行によるテストケース
No.1 : 1 -> "未成年"
No.2 : 21 -> "成人"
```

「even」を定義している。しかし、既存の BWDM は、定数定義ブロック内の定義を解析しない。したがって、「偶数判定」関数内の even を引数と判断するが、even という引数は存在しないため、エラーを出力してしまう。

本研究では、定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できないという問題を解決するために、既存の BWDM の構文解析処理を拡張し、定数定義ブロック内の定義を含む関数のテストケース生成に対応することによって、BWDM を拡張する。

3.3.5 操作のテストケースを生成できない問題点

既存の BWDM には、操作のテストケースを生成できないという問題点がある。操作を含む VDM++ 仕様を、コード 3.7 に示す。この VDM++ 仕様には、「西暦設定」操作と「うるう年判定」操作が定義されている。また、インスタンス変数「current_year」を定義している。既存の BWDM は、インスタンス変数定義ブロック、および、操作定義ブロック内の定義を解析しない。したがって、「西暦設定」操作と「うるう年判定」操作のテストケース生成を行わない。

本研究では、操作のテストケース生成ができないという問題を解決するために、既存の BWDM の構文解析処理を拡張し、操作を含む仕様のテストケース生成に対応することによって、BWDM

コード 3.6: 定数定義ブロック内の定義を含む関数を含む VDM++ 仕様

```
1  class 判定
2
3  values
4
5  public static even : int = 2;
6
7  functions
8
9  public 偶数判定 : int -> seq of char
10   偶数判定(数) ==
11     if(数 mod even = 0) then
12       "偶数"
13     else
14       "奇数";
15
16 end 判定
```

を拡張する。ただし、拡張する BWDM は、if-then-else 式を含まない操作についてはテストケースを生成しない。これは、BWDM が if-then-else 式のみに基づいてテストケースを生成するためである。

コード 3.7: 操作を含む VDM++ 仕様

```
class 判定

instance variables

private current_year : nat := 2000;

operations

public 西暦設定 : int ==> int
  西暦設定(年) == current_year := 年;

public うるう年判定 : () ==> seq of char
  うるう年判定() ==
    if(current_year mod 4 = 0) then
      if(current_year mod 100 = 0) then
        if(current_year mod 400 = 0) then
          return "うるう年"
        else
          return "平年"
      else
        return "うるう年"
    else
      return "平年";

end 判定
```

第 4 章

BWDM の拡張

本章では、BWDM の拡張について説明する。既存の BWDM には 5 つの問題点がある (3.3 節参照)。これらの問題点を解決するために、本研究では、3 つの拡張を行う。また、拡張後の BWDM は、名称を BWDM(Verification tool for Vienna Development Method) に変更する [12]。

本研究で行う 3 つの拡張を、以下に示す。

- ペアワイズ法の適用によるテストケース数削減
- ドメイン分析テストの適用による複数変数を含む条件式を含む関数のテストケース生成
- 複数の定義への対応

ここで、本研究で行う拡張とそれぞれが対応する問題を、表 4.1 に示す。また、本研究で拡張する BWDM の構造を、図 4.1 に示す。

以降、各拡張について説明する。

4.1 ペアワイズ法の適用によるテストケース数削減

本研究では、pict4java を開発する [7, 8, 9, 11, 16]。pict4java は、PICT と BWDM を接続するためのインタフェースである。そして、既存の BWDM に pict4java を組込むことで、BWDM を拡張する。詳細を、以下に示す。

表 4.1: 本研究で行う拡張とそれぞれが対応する問題

本研究で行う拡張	解決する問題
ペアワイズ法の適用による テストケース数削減	組合せ爆発に関する問題点 (3.3.1 節参照)
ドメイン分析テストの適用による 複数変数を含む条件式を含む関数の テストケース生成	複数変数を含む条件式に関する問題点 (3.3.2 節参照)
複数の定義への対応	1 つの関数しか テストケースを生成できない問題点 (3.3.3 節参照)
	定数定義ブロック内の定義を含む関数を含む仕様の テストケースを生成できない問題点 (3.3.4 節参照)
	操作のテストケースを生成できない問題点 (3.3.5 節参照)

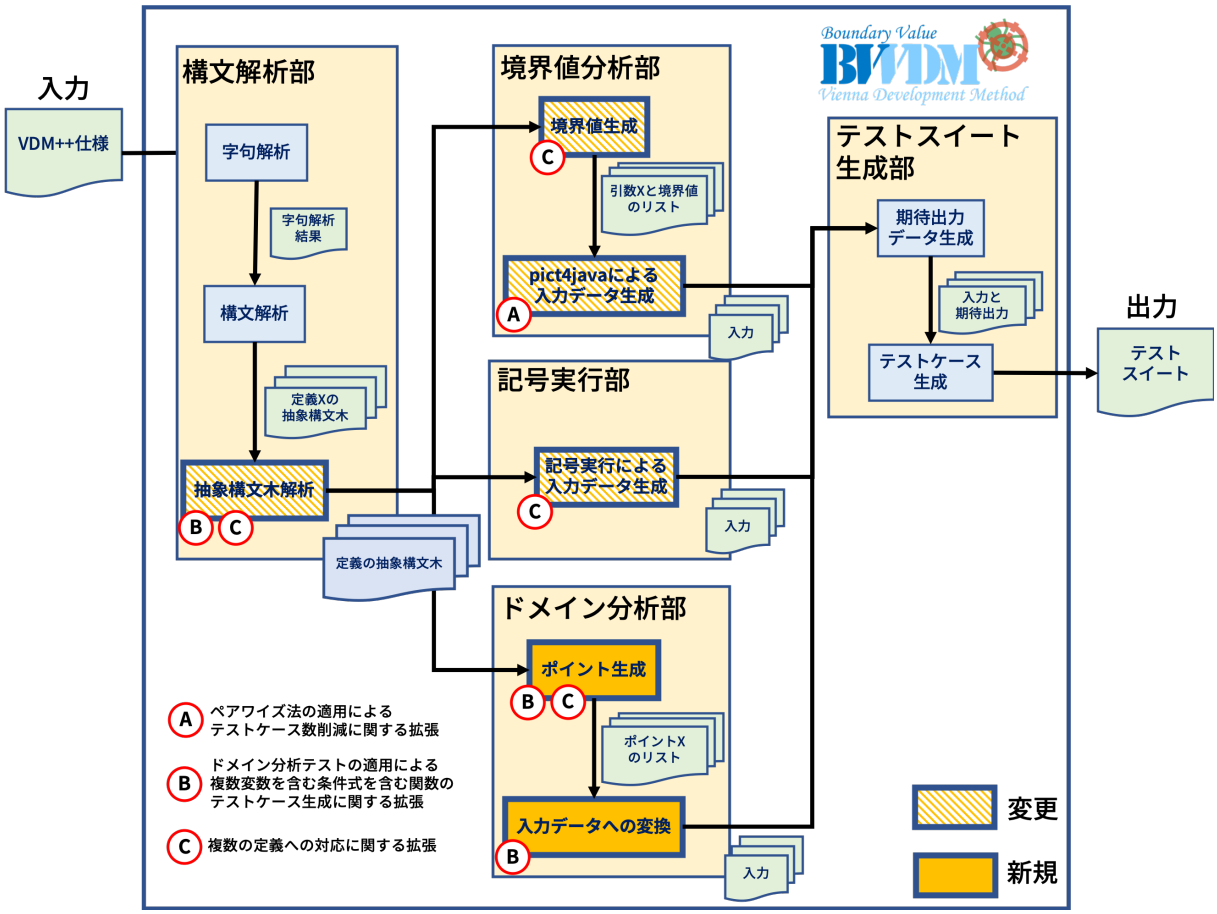

図 4.1: 本研究で拡張する BWDM の構造

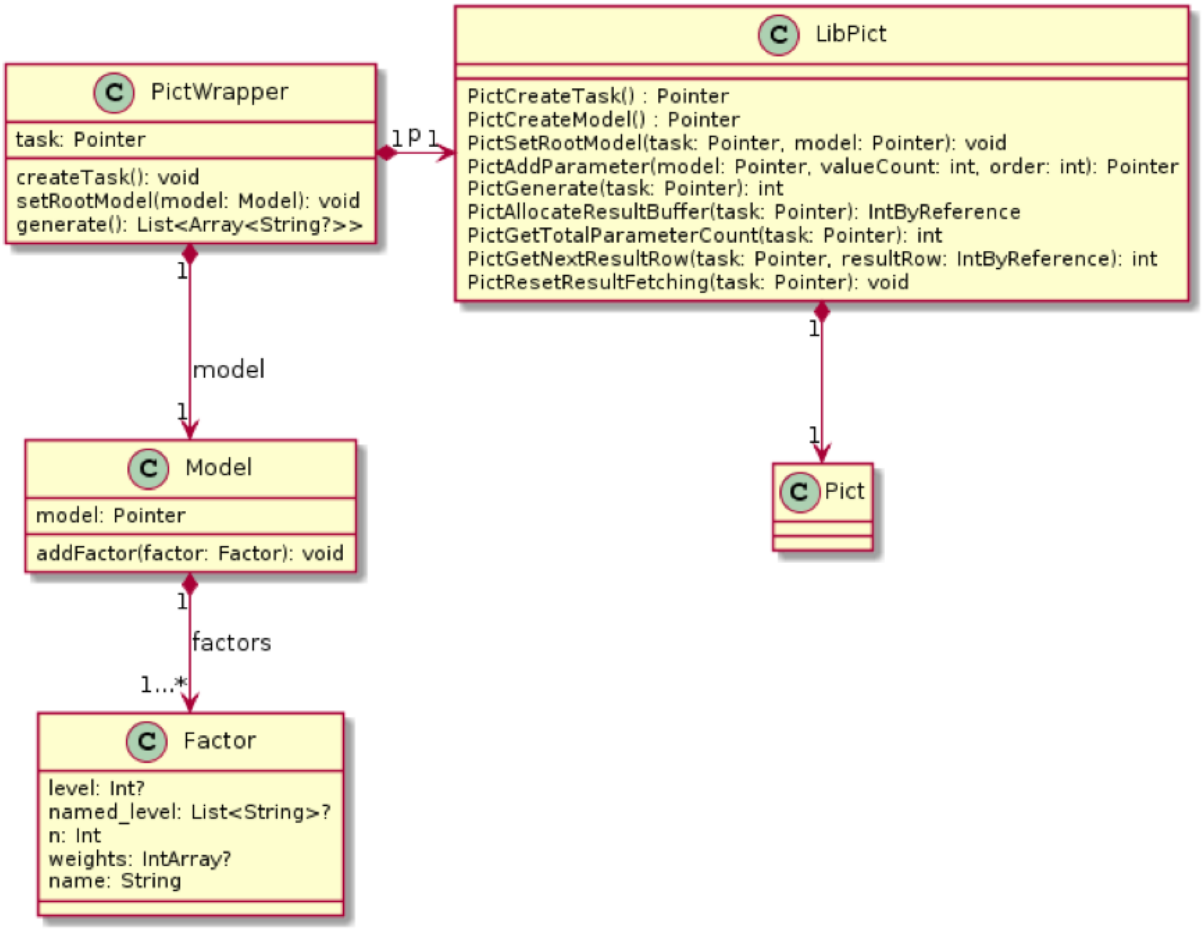

図 4.2: pict4java のクラス図

4.1.1 pict4java の概要

PICT は、CLI(Command Line Interface) ツールである。API(PICT ライブラリと呼称する) も提供しており、C++ からのみ利用できる。既存の BWDM は Java で記述しており、PICT ライブラリを呼び出すことができない。そのため、BWDM の拡張の準備として、pict4java を開発する。pict4java は、JNA(Java Native Access)[26] を利用し、Java から呼び出すことのできる PICT ライブラリである。

図 4.2 に、pict4java のクラス図を示す。それぞれのクラスの説明を、以下に示す。

- クラス `Pict`

  Microsoft 社が開発した C++ で記述された PICT ライブラリである。

- クラス `LibPict`

JNA(Java Native Access)[26] を用いて Java で記述した PICT ライブラリのインタフェースである。メソッド名はすべて、PICT ライブラリの持つ関数名と同じである。

主に使用する PICT の関数を、以下に示す。

PictAddParameter() PICT への因子と水準の登録

PictGenerate() 組合せデータの生成

PictGetNextResultRow() 生成データの取得

- クラス PictWrapper

  PICT を操作するためのクラスである。Kotlin[27] で記述する。以下の機能を持つ。

  createTask() Task の生成と初期化をする。Task は、PICT の組合せ生成処理の最小単位である。PICT ライブラリにおける PictCreateTask に相当する。

  setRootModel() Task に Model の登録をする。Model は因子の集合である。PICT ライブラリにおける PictSetRootModel に相当する。

  generate() ペアワイズ法を適用した組合せの生成をする。PICT ライブラリにおける PictGenerate に相当する。

- クラス Model

  因子の集合を保持するためのクラスである。Kotlin で記述する。

  コンストラクタ () PICT ライブラリにおける Model を生成する。PictCreateModel に相当する。

  addFactor() 因子 (Factor) を Model に登録する。PICT ライブラリにおける PictAddParameter に相当する。

- クラス Factor

  level 水準

  named_level 因子の取り得る値の集合

  n 最低限組合せるペア数 (デフォルトは 2)

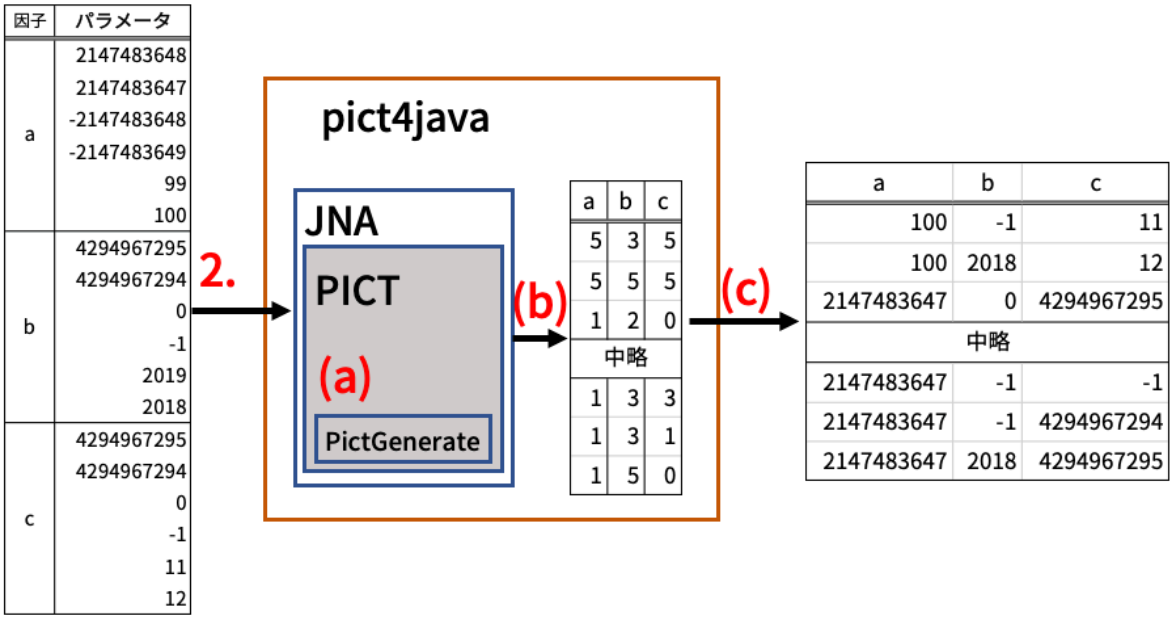

図 4.3: pict4java の処理の流れ

weights 因子の取り得る値の重みの集合

name 因子の名前

メンバ変数 n の値を変えることによって、その因子については、n 個の組合せを網羅する入力データを作成できる。

図 4.1 の「A」の文字の箇所が、本節で拡張した処理である。境界値分析部ではまず、テストケースの入力データとして、VDM++ 仕様内の引数ごとにおける、不等式、剰余式などに合わせた境界値、および、型の最小値、最大値の境界値を、それぞれ抽出する。既存の BWDM では、引数ごとに生成した境界値のすべての組合せを生成し、境界値テストの入力データとしていた。

拡張後の BWDM では、すべての組合せを生成するのではなく、4.1.1 節で述べた pict4java を用いてペアワイズ法を適用し、入力データ生成を行う。具体的には、境界値分析で得た因子が取り得る値を pict4java に入力する。

境界値分析後の境界値データを受けとった pict4java の入力データ生成アルゴリズムを、以下に示す。また、このアルゴリズムを用いた pict4java の処理の流れを、図 4.3 に示す。図中の赤文字は、アルゴリズムの項目と対応している。

1. 因子と、因子の取り得る値を元に、クラス `Factor` のインスタンスを因子の数だけ生成する。

2. クラス Model の addFactor メソッドを用いて、PictAddParameter 関数を呼び出し、PICT に因子と因子ごとの水準を登録する。

3. クラス PictWrapper の generate メソッドを用いて、ペアワイズ法を適用した組合せデータのリストを生成する。組合せデータは文字列型の配列のリストである。詳細の処理を、以下に示す。

   (a) PictGenerate 関数を用いて、PICT にペアワイズ法を適用した組合せデータを生成させる。

   (b) PictGetNextResultRow 関数を用いて (a) で生成した組合せデータを 1 件取得する。取得したデータは、因子が取り得るパラメータ群のインデックスとなる。因子の数だけ (b) の処理を繰り返す。

   (c) (b) で取得したデータのインデックスに該当するパラメータを用いて、組合せデータのリストを生成する。

4. 3. で生成したリストを、pict4java の出力データとする。

4.2 ドメイン分析テストの適用による複数変数を含む条件式を含む関数のテストケース生成

図 4.1 の「B」の文字の箇所が、本節で拡張した処理である。既存の BWDM では、入力した VDM++ 仕様を構文解析し、その結果を、境界値分析部と記号実行部に渡し、テストケースにおける入力データを生成する。

拡張する BWDM には、境界値分析部、記号実行部に加えて、ドメイン分析部を追加する [10]。ドメイン分析部では、2.5.1 節で記述した、in ポイント、out ポイント、on ポイント、off ポイントを生成する。また、複数の変数が条件式に含まれる VDM++ 仕様の解析に対応できるように、構文解析部を一部修正する。さらに、ドメインテストに必要な、正常系判定値と着目条件式、着目変数の情報をテストケースに含めるために、テストケース生成部において、ドメインテストによるテストケースを出力する際、既存の出力に加えて、正常系判定値と着目条件式、着目変数の情報も出力するよう処理を追加する。

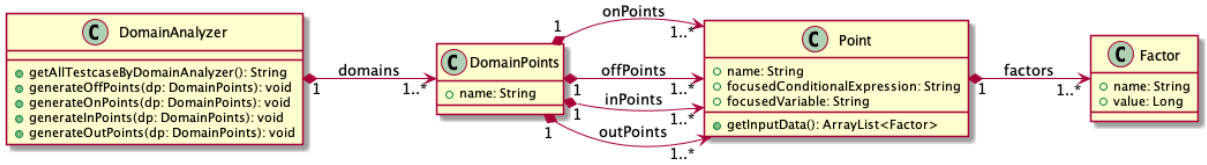

図 4.4: ドメイン分析部のクラス図

4.2.1 構文解析部

既存の BWDM は、各条件式の左辺に複数の変数を含む場合、エラーを出力してしまう (3.3 節参照)。左辺に複数の変数が条件式に含まれる VDM++ 仕様の構文解析を可能とするために、if 条件式の左辺と右辺の式を構文解析し、条件式内の変数を抽出する。

4.2.2 ドメイン分析部

拡張する BWDM はドメイン分析テストにおける、on ポイント、off ポイント、in ポイント、out ポイントを生成するために、ドメイン分析部を持つ。各ポイントは、テストケースの入力データであり、引数名と値のタプルの配列を保持している。各ポイントの生成については、4.2.3 節にて説明する。

作成するドメイン分析部のクラス図を、図 4.4 に示す。各クラスの詳細を、以下に示す。

- `Factor` クラスは、引数の情報を持つクラスである。変数名 (`name`) と値 (`value`) を保持する。
- `Point` クラスは、テストケースの入力データの情報を持つクラスである。テストケース名 (`name`)、そのポイントが着目条件式 (`forcusedConditionalExpression`)、そのポイントの着目変数 (`forcusedVariable`)、そして複数の `Factor`(`factors`) を保持する。
- `DomainPoints` クラスは、ドメインの情報を持つクラスである。ドメインの期待出力 (`name`)、on ポイントの集合 (`onPoints`)、off ポイントの集合 (`offPoints`)、in ポイント (`inPoints`)、out ポイントの集合 (`outPoints`) を保持する。
- `DomainAnalyzer` クラスは、ドメイン分析を行うクラスである。ドメインの集合 (`domains`)

を保持している。また、on ポイント、off ポイント、in ポイント、out ポイントを生成する機能 (generateXXPoints メソッド)、および、各ドメインの各ポイントを、期待出力生成部に入力できるデータ構造として抽出する機能 (getAllTestCaseByDomainAnalyzer メソッド) を持つ。

4.2.3 各ポイントの生成手法の提案と適用

拡張する BWDM は、入力する仕様のドメインごとに、on ポイント、off ポイント、in ポイント、out ポイントを生成する。

各ポイントを満たす変数の値を求めるために、SMT ソルバ (Satisfiable Modulo Theories)[25] を利用する。充足可能性問題 (SAT) を解くアルゴリズムを実装したソフトウェアを SAT ソルバと呼び、SAT ソルバを算術演算に対応させたソフトウェアを SMT ソルバと呼ぶ。幅広く知られている SMT ソルバの 1 つに、Microsoft Research が開発を進めている Z3[28] がある。Z3 は、C、C++、Java、Python などのプログラミング言語から利用できる API を提供する。拡張する BWDM は、この Z3 を利用し、条件式を満たす入力値を生成する。

各ポイントの生成方法を、以下に示す。

on ポイント

on ポイントは、着目条件式の TB である。ドメインを決定づける条件式に付き 1 つ生成し、他の on ポイントと重複してはならない。ドメインに関わる条件式の数だけ以下の手順を繰り返し、各条件式に着目した on ポイントを生成する。

1. もし、着目条件式の比較演算子が、“>=”、または、“<=” である場合、“=” に置き換える。“>” である場合、“=” で置き換え、右辺を “+1” する。“<” である場合、“=” で置き換え、左辺を “+1” する。

2. 1. で修正した条件式と、その他の条件式を Z3 に入力し、解 (引数と値のタプルの集合) を求める。解が求まらなかった場合、現在の着目条件式における on ポイントが存在しないため、生成を行わずに次の着目条件式の on ポイントの生成を行う。

3. 配列 onPoints(4.2.2 節参照) を参照し、他の on ポイントと値が重なるかどうかを判定する。

(a) 重なっている場合、条件式に、” 重なっている変数 ! = 重なった値 “という条件式を加え、2. に戻る。

(b) 重なっていない場合、解を元に、Point インスタンス (4.2.2 節参照) を作り、メンバ forcusedConditionalExpression には、着目条件式を格納する。作成したインスタンスを配列 onPoints に格納する。

off ポイント

off ポイントは、着目する on ポイントに隣接し、TB でない値である。on ポイントに付き複数 (着目条件式に含まれる変数 * 2) 個存在する。配列 onPoints(4.2.2 節参照) を参照し、on ポイントの数だけ以下の手順を繰り返し、on ポイントに着目した off ポイントを生成する。

1. 着目する on ポイントのメンバ forcusedConditionalExpression を参照し、着目する on ポイントがどの条件式に着目していたかを保持する。

2. 1. の条件式から、引数を抽出する。抽出した引数の数だけ以下を繰り返す。

   (a) 以下の処理を 2 回繰り返す。1 回目は “N=−1” と定義し、2 回目は “N=1” と定義する。

      i. 着目する on ポイントインスタンスをコピーする。

      ii. i. でコピーした Point インスタンスのメンバ配列 factors から、“Factor.name == 引数名” となる Factor インスタンスを検索する。

      iii. ii. で検索して見つかった Factor インスタンスのメンバ value を “+N” する。

      iv. Point インスタンスのメンバ forcusedVariable に引数名を格納する。

      v. Point インスタンスを配列 offPoints(4.2.2 節参照) に格納する。

in ポイント

in ポイントは、ドメインを決定づけるすべての条件式を満たす値である。ドメインに付き 1 つ生成し、他の on ポイント、off ポイントと重複してはならない。以下の手順を行い、in ポイントを生成する。

1. もし、条件式の比較演算子が、“>=” である場合、“>” で置き換える。“<=” である場合、“<” で置き換える。これを、すべての条件式に対して行う。

2. 1. で修正した条件式を Z3 に入力し、解 (引数と値のタプルの集合) を求める。解が求まらなかった場合、in ポイントが存在しないため、生成を行わない。

3. 配列 `onPoints` と配列 `offPoints`(4.2.2 節参照) を参照し、他の on ポイント、off ポイントと値が重なるかどうかを判定する。

   (a) 重なっている場合、条件式に、” 重なっている変数 ! = 重なった値 “という条件式を加え、2. に戻る。

   (b) 重なっていない場合、解を元に、`Point` インスタンス (4.2.2 節参照) を作り、配列 `inPoints` に格納する。

out ポイント

out ポイントは、着目条件式のみを満たさない値である。ドメインを決定づける条件式に付き 1 つ生成し、他の off ポイントと重複してはならない、ドメインに関わる条件式の数だけ以下の手順を繰り返し、各条件式に着目した out ポイントを生成する。

1. 着目条件式を “!(着目条件式)” に置き換え、否定する。

2. 1. で否定した条件式と、その他の条件式を Z3 に入力し、解 (引数と値のタプルの集合) を求める。解が求まらなかった場合、現在の着目条件式における out ポイントが存在しないため、生成を行わずに次の着目条件式の out ポイントの生成を行う。

3. 配列 `onPoints` と配列 `offPoints`(4.2.2 節参照) を参照し、他の on ポイント、off ポイントと値が重なるかどうかを判定する。

   (a) 重なっている場合、条件式に、” 重なっている変数 ! = 重なった値 “という条件式を加え、2. に戻る。

コード 4.1: ドメイン分析テストのためのテストケースの出力フォーマット

```
1
2  ドメインテストによるテストケース
3  - <ドメインの期待出力>
4  -- On ポイント
5  <ID> : <入力値 1> ... <入力値N>, <正常系かどうか>, <着目する条件式> -> <期待出力>
6
7  -- Off ポイント
8  <ID> : <入力値 1> ... <入力値N>, <正常系かどうか>, <着目する条件式>, <着目する条件式> -> <期待出力>
9
10 -- In ポイント
11 <ID> : <入力値 1> ... <入力値N>, <正常系かどうか> -> <期待出力>
12
13 -- Out ポイント
14 <ID> : <入力値 1> ... <入力値N>, <正常系かどうか>, <着目する条件式> -> <期待出力>
15
16 - <ドメイン名 2>
17
18 ・
19 ・
20 ・
21
22 - <ドメイン名N>
```

(b) 重なっていない場合、解を元に、Point インスタンス (4.2.2 節参照) を作り、メンバ forcusedConditionalExpression には、着目条件式を格納する。作成したインスタンスを配列 outPoints に格納する。

4.2.4 テストケースの生成

既存の BWDM は、期待出力生成部において、入力データを元に期待出力を生成する機能を持つ。DomainAnalyzer クラス (4.2.2 節参照) の getAllTestCaseByDomainAnalyzer メソッドを呼び出すことにより、4.2.3 節で生成した各ドメインの各ポイントを、期待出力生成部に入力できるデータ構造として抽出できる。そのため、既存の BWDM の期待出力生成部をそのまま用いて、期待出力生成とテストケース生成は可能である。拡張する BWDM では、各ドメインの各ポイントのテストケースを出力する。

しかし、既存の BWDM のテストケース生成部には、ドメインテストに必要な、正常系判定値と着目条件式、着目変数の情報を出力テストケースに加える処理が存在しない。これに対応するため、テストケース生成部において、ドメインテストによるテストケースを出力する際、正常系判定値と着目条件式、着目変数の情報についても出力するよう処理を追加する。着目条件式の出力には、Point インスタンスのメンバ forcusedConditionalExpression を参照

する。off ポイントの場合、着目変数の出力を行う。出力には、Point インスタンスのメンバ forcusedVariable を参照する。正常系判定値は、出力するテストケースの期待出力と、ドメインの期待出力 (DomainPoints.name) を比較して判断する。等しければ、“正常系” と出力する。等しくなければ “非正常系” と出力する。

最後に、出力テストケースのフォーマット (コード 4.1 参照) の < と > で囲まれている部分を、それぞれの情報で置き換える。置き換えたファイルをテストスイートとして出力する。

4.3 複数の定義への対応

図 4.1 の「C」の文字の箇所が、本節で拡張した処理である。既存の BWDM では、入力した VDM++ 仕様を構文解析部に入力し、字句解析、構文解析、抽象構文木解析を順に行い、最後に定義された定義の抽象構文木を抽出する。

拡張する BWDM には、3.3.4 節と 3.3.5 節で述べた、関数定義以外の定義に対応していないという問題を解決するために、抽象クラス Definition、クラス FunctionDefinition、クラス OperationDefinition を新規に作成し、追加する。また、3.3.3 節で述べた、1 つの関数しかテストケース生成できないという問題を解決するため、構文解析部の抽象構文木解析処理を一部修正する。

4.3.1 定義クラス群の作成

関数定義以外の定義に対応するために、抽象クラス Definition、クラス FunctionDefinition、クラス OperationDefinition を作成する。これらの、定義クラス群の関係を、図 4.5 に示す。図中にある、VDMJ パッケージのクラスは VDMJ であらかじめ用意されているクラス (2.2 節参照) である。

それ以外のクラスの説明を、以下に示す。

- 抽象クラス Definition

  テストケース生成を行う定義を表す抽象クラスである。TCDefinition 型のオブジェクトを元に、if-then-else 式から木構造を生成する [6]。以下のフィールドとメソッドを持つ。

  name 定義の名前

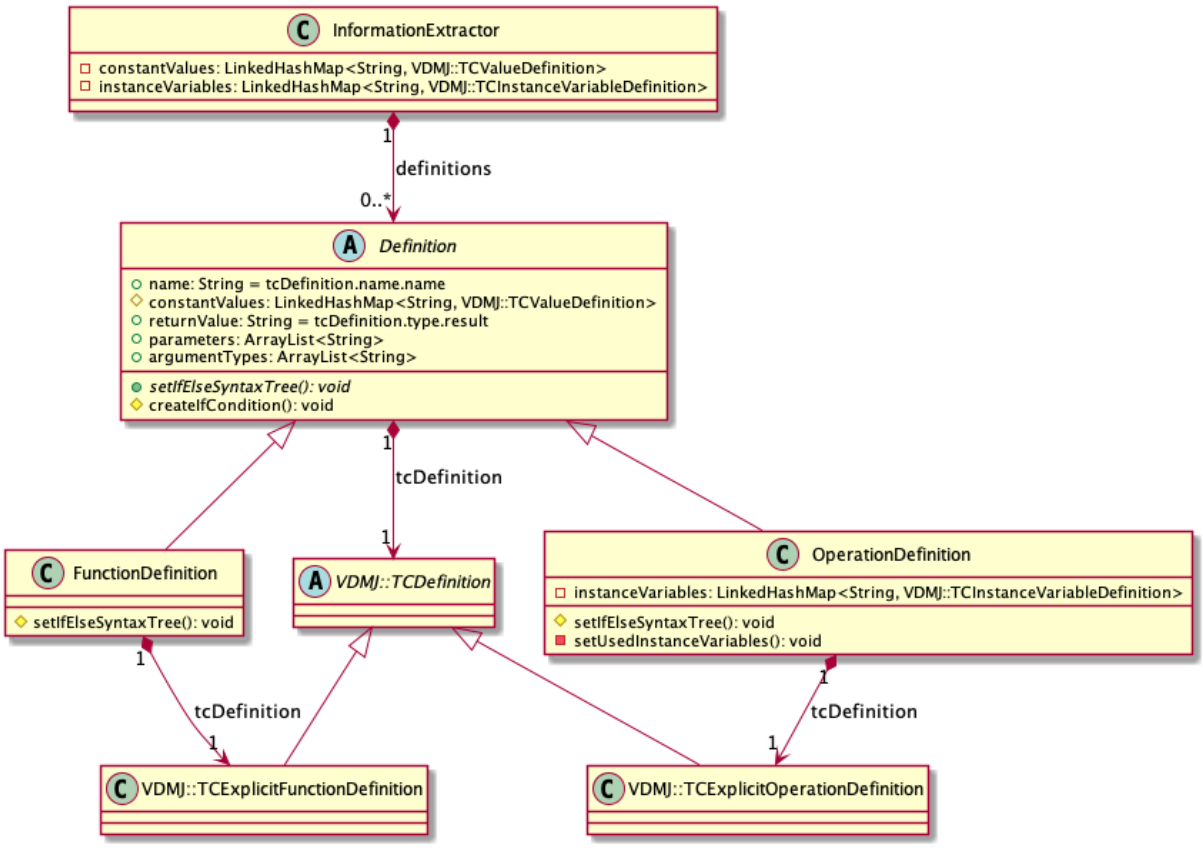

図 4.5: 定義クラス群とそれに関連するクラスのクラス図

constantValues 定数定義の集合

returnType 戻り値の型

argumentTypes 引数の型の集合

parameters 仮引数名の集合

setIfElseSyntaxTree() if-then-else 式から木構造を生成する抽象メソッド

createIfCondition() if-then-else 式を構文解析する

- クラス FunctionDefinition

  テストケース生成を行う関数を表すクラスである。TCExplicitFunctionDefinition 型のオブジェクトを元に、if-then-else 式から木構造を生成する。以下のフィールドとメソッドを持つ。

  setIfElseSyntaxTree() if-then-else 式から木構造を生成する。インスタンス生成時にコンストラクタから呼び出される

- クラス OperationDefinition

  テストケース生成を行う操作を表すクラスである。TCExplicitOperationDefinition 型のオブジェクトを元に、if-then-else 式から木構造を生成する。以下のフィールドとメソッドを持つ。

  instanceVariables インスタンス変数定義の集合

  setUsesInstanceVariables() if-then-else 式で使用されるインスタンス変数を argumentTypes と parameters に追加する

  setIfElseSyntaxTree() if-then-else 式から木構造を生成する。インスタンス生成時にコンストラクタが呼び出す

- クラス InformationExtractor

  抽象構文木解析処理を行うクラスである。構文解析処理から受け取った、定義を表す抽象構文木の集合から、定義クラスの集合を生成する。

  constantValues 定数定義の集合

instanceVariables インスタンス変数定義の集合

4.3.2 抽象構文木解析処理の修正

既存の BWDM では、図 4.1 に示した構文解析処理で定義ごとの抽象構文木を生成し、それらを抽象構文木解析処理に渡す。抽象構文木解析処理では、受け取った抽象構文木の集合から、関数定義であるものだけを抽出する。受け取る抽象構文木の集合は、4.3.1 節で示した抽象クラス TCDefinition を継承したクラスの集合である。

拡張する BWDM では、抽象構文木解析処理にて、受け取った抽象構文木を、4.3.1 節で作成した定義クラス群に変換し、連想配列に格納する。拡張する BWDM の抽象構文木解析処理を以下に示す。以下の処理は InformationExtractor インスタンスにて行う。

1. FunctionDefinition 型の連想配列 explicitFunctions をインスタンス化する。
2. OperationDefinition 型の連想配列 explicitOperations をインスタンス化する。
3. TCValueDefinition 型の連想配列 constantValues をインスタンス化する。
4. TCInstanceVariableDefinition 型の連想配列 instanceVariables をインスタンス化する。
5. 構文解析処理で生成した抽象構文木の集合が空であるか確認する。空でない場合、6. へ進む。空である場合、連想配列 explicitFunctions と連想配列 explicitOperations を構文木解析処理の出力とし、処理を終了する。
6. 抽象構文木の集合の先頭を取り出し、変数 astDefinition に格納する。
7. astDefinition.kind() を実行し、抽象構文木の種類を取得する。
8. 7 で取得した抽象構文木の種類を確認し、種類に応じて以下のいずれかを実行する。
   (a) “explicit function” である場合、関数定義なので、以下を実行する。
      i. 変数 astDefinition を TCExplicitFunctionDefinition に型変換し、クラス FunctionDefinition に渡し、インスタンス化する。

ii. インスタンス化したクラス FunctionDefinition を、連想配列 explicitFunctions に、キーを定義名 (astDefinition.name) として追加する。

(b) “explicit operation” である場合、操作定義なので、以下を実行する。

i. 変数 astDefinition を TCExplicitOperationDefinition に型変換し、クラス OperationDefinition に渡し、インスタンス化する。

ii. インスタンス化したクラス OperationDefinition を、連想配列 explicitOperations に、キーを定義名 (astDefinition.name) として追加する。

(c) “value” である場合、定数定義なので、astDefinition を TCValueDefinition に型変換し、tcValueDefinition に追加する。

(d) “instance variable” である場合、インスタンス変数定義なので、astDefinition を TCInstanceVariableDefinition に型変換し、tcInstanceVariableDefinition に追加する。

(e) (a)、(b)、(c)、(d) のいずれでもない場合、5. へ進む。

4.3.3 境界値分析部と記号実行部とドメイン分析部で定義クラス群を入力とするように修正

既存の BWDM では、境界値分析部、記号実行部、ドメイン分析部に入力するオブジェクトの型が TCExplicitFunctionDefinition 型のみであった。したがって、拡張する BWDM では、4.3.1 節で作成した定義クラス群を、境界値分析部、記号実行部、ドメイン分析部に入力できるように、それぞれの部を修正する。

第 5 章

適用例

本章では、本研究で拡張した BWDM が正しく動作することを、適用例を用いて確認する。

5.1 因子と水準の組合せ数が大きい仕様

4.1 節で拡張した BWDM が正しく動作することを検証するため、拡張した BWDM に因子と水準の組合せ数が大きい VDM++ 仕様を適用した。適用例として用いる、因子が 3 で水準が (6, 6, 6) の関数の VDM++ 仕様を、コード 5.1 に示す。適用結果を、コード 5.2 に示す。

この例の場合、既存の BWDM では $6 \times 6 \times 6 = 216$ 個 のテストケースを生成する。これに対して、拡張後の BWDM では、コード 5.2 より、テストケースの生成を 40 個に抑えており、かつ、この 40 個のテストケースは、2 個の因子のペアの組合せをすべて網羅できていることが確認できる。すなわち、拡張した BWDM が、VDM++ 仕様から、ペアワイズ法を適用した境界値テストケースを正しく出力できていることが確認できた。

5.2 複数変数を含む条件式を用いた仕様

拡張した BWDM に、遊園地チケット割引機能 (コード 2.1 参照) を入力として適用した結果の一部を、コード 5.3 に示す。なお、コード 5.3 では、“割引価格となる” 期待出力以外の 3 つの期待出力を持つドメイン (“割引価格とならない (妻の年齢 < 16)”、“割引価格とならない (夫の年齢 < 18)”、“割引価格とならない (夫の年齢 + 妻の年齢 > 50)”) のテストケースは省略してある。

コード 5.1: 因子が 3 で水準が (6, 6, 6) の関数を持つ VDM++ 仕様

```
class SampleClass
functions

sampleFunction : int*nat*nat -> seq of char
sampleFunction(a, b, c)==
  if(a < 100) then
    if(b > 2018) then
      "a は 100 未満かつ b は 2018 より大きい"
    else
      "a は 100 未満かつ b は 2018 以下"
  elseif(c < 12) then
    "a は 100 以上かつ c は 12 未満"
  else
    "a は 100 以上かつ c は 12 以上";

end SampleClass
```

“割引価格となる” ドメインの 3 つの on ポイントは、ドメインを決定づける 3 つの条件式 (“夫の年齢 + 妻の年齢 <= 50”、“夫の年齢 >= 18”、“妻の年齢 >= 16”) にそれぞれ着目し、着目条件式の TB が入力となっており、かつ、期待出力と正常系判定値 (ポイントの期待出力とドメインの期待出力が一致するかどうか) が適切であることが確認できる。off ポイントは、各 on ポイントに隣接しており、TB ではない値が入力となっており、かつ、着目変数、期待出力、正常系判定値が適切であることが確認できる。in ポイントは、期待出力とドメインの期待出力が一致しており、各条件式の TB でない値が入力となっており、かつ、正常系であることが確認できる。out ポイントは、関係する 3 つの条件式に着目し、着目条件式のみを否定する TB でない値が入力となっており、かつ、期待出力と正常系判定値が適切であることが確認できる。

また、コード 5.3 では省略している、“割引価格となる” 期待出力以外の期待出力を持つドメイン (“割引価格とならない (妻の年齢 < 16)”、“割引価格とならない (夫の年齢 < 18)”、“割引価格とならない (夫の年齢 + 妻の年齢 > 50)”) に対しても、テストケースが適切に生成できていることを確認した。

したがって、拡張した BWDM は、既存の BWDM の問題点である、条件式内に複数の変数がある VDM++ 仕様を解析できること、かつ、ドメインテストによるテストケース生成が適切にできることを確認できた。

コード 5.2: 拡張した BWDM に因子が 3 で水準が (6, 6, 6) の関数を持つ仕様 (コード 5.1) を適用した際の出力

```
1
2  関数名 : sampleFunction
3  引数の型 : a:int b:nat c:nat
4  戻り値の型 : seq of (char)
5  生成テストケース数 : 44件 (境界値分析:40/記号実行:4)
6
7  各引数の境界値
8  a : 2147483648 2147483647 -2147483648 -2147483649 99 100
9  b : 4294967295 4294967294 0 -1 2019 2018
10 c : 4294967295 4294967294 0 -1 11 12
11
12 記号実行情報
13 戻り値の数 : 4
14 制約 : b > 2018 and a < 100, 戻り値 : "a は 100 未満かつ b は 2018 より大きい"
15 制約 : !( b > 2018 ) and a < 100, 戻り値 : "a は 100 未満かつ b は 2018 以下"
16 制約 : c < 12 and !( a < 100 ) , 戻り値 : "a は 100 以上かつ c は 12 未満"
17 制約 : !( c < 12 ) and !( a < 100 ) , 戻り値 : "a は 100 以上かつ c は 12 以上"
18
19 境界値分析によるテストケース(ペアワイズ法適用)
20 No.1 : 100 4294967295 -1 -> Undefined Action
21 No.2 : 2147483648 2019 -1 -> Undefined Action
22 No.3 : 100 2019 11 -> "a は 100 以上かつ c は 12 未満"
23 No.4 : 2147483648 2018 4294967294 -> Undefined Action
24 No.5 : 2147483647 -1 4294967294 -> Undefined Action
25 No.6 : -2147483649 0 0 -> Undefined Action
26 No.7 : 2147483648 -1 4294967295 -> Undefined Action
27 No.8 : 100 -1 12 -> Undefined Action
28 No.9 : -2147483649 2019 12 -> Undefined Action
29 No.10 : 99 -1 0 -> Undefined Action
30 No.11 : 99 4294967295 11 -> Undefined Action
31 No.12 : 99 2019 4294967294 -> "a は 100 未満かつ b は 2018 より大きい"
32 No.13 : -2147483648 2018 -1 -> Undefined Action
33 No.14 : 100 2018 0 -> "a は 100 以上かつ c は 12 未満"
34 No.15 : 99 4294967294 -1 -> Undefined Action
35 No.16 : -2147483649 2018 4294967295 -> Undefined Action
36 No.17 : 2147483647 4294967295 4294967295 -> Undefined Action
37 No.18 : 100 0 4294967295 -> Undefined Action
38 No.19 : 2147483647 0 12 -> "a は 100 以上かつ c は 12 以上"
39 No.20 : 99 2018 12 -> "a は 100 未満かつ b は 2018 以下"
40 No.21 : -2147483648 4294967294 4294967295 -> Undefined Action
41 No.22 : 2147483647 4294967294 0 -> "a は 100 以上かつ c は 12 未満"
42 No.23 : 99 0 4294967295 -> Undefined Action
43 No.24 : -2147483649 4294967294 4294967294 -> Undefined Action
44 No.25 : 2147483647 2019 4294967295 -> Undefined Action
45 No.26 : -2147483648 0 11 -> "a は 100 未満かつ b は 2018 以下"
46 No.27 : 2147483648 4294967294 11 -> Undefined Action
47 No.28 : 2147483647 -1 -1 -> Undefined Action
48 No.29 : -2147483649 -1 11 -> Undefined Action
49 No.30 : 100 4294967294 12 -> "a は 100 以上かつ c は 12 以上"
50 No.31 : 2147483648 4294967295 0 -> Undefined Action
51 No.32 : 100 4294967295 4294967294 -> Undefined Action
52 No.33 : 2147483647 2018 11 -> "a は 100 以上かつ c は 12 未満"
53 No.34 : 2147483648 0 4294967294 -> Undefined Action
54 No.35 : -2147483648 2019 0 -> "a は 100 未満かつ b は 2018 より大きい"
55 No.36 : 2147483648 4294967295 12 -> Undefined Action
56 No.37 : -2147483648 4294967295 4294967294 -> Undefined Action
57 No.38 : -2147483648 -1 12 -> Undefined Action
58 No.39 : -2147483649 0 -1 -> Undefined Action
59 No.40 : -2147483649 4294967295 0 -> Undefined Action
```

コード 5.3: 拡張した BWDM に遊園地チケット割引機能 (コード 2.1) を適用した際の出力の一部

```
ドメインテストによるテストケース
- "割引価格となる"
-- On ポイント
No.1 : 20 16, 正常系, 妻の年齢>=16 -> "割引価格となる"
No.2 : 18 18, 正常系, 夫の年齢>=18 -> "割引価格となる"
No.3 : 32 18, 正常系, 夫の年齢 +妻の年齢<=50 -> "割引価格となる"

-- Off ポイント
No.1 : 20 17, 正常系, 妻の年齢>=16, 妻の年齢 -> "割引価格となる"
No.2 : 20 15, 非正常系, 妻の年齢>=16, 妻の年齢 -> "割引価格とならない (妻の年齢 < 16)"
No.3 : 32 19, 非正常系, 夫の年齢 +妻の年齢<=50, 妻の年齢 -> "割引価格とならない (夫の年齢 + 妻の年齢 > 50)"
No.4 : 32 17, 正常系, 夫の年齢 +妻の年齢<=50, 妻の年齢 -> "割引価格となる"
No.5 : 33 18, 非正常系, 夫の年齢 +妻の年齢<=50, 夫の年齢 -> "割引価格とならない (夫の年齢 + 妻の年齢 > 50)"
No.6 : 31 18, 正常系, 夫の年齢 +妻の年齢<=50, 夫の年齢 -> "割引価格となる"
No.7 : 19 18, 正常系, 夫の年齢>=18 -> "割引価格となる"
No.8 : 17 18, 非正常系, 夫の年齢>=18 -> "割引価格とならない (夫の年齢 < 18)"

-- In ポイント
No.1 : 20 18, 正常系 -> "割引価格となる"

-- Out ポイント
No.1 : 20 0, 非正常系, 妻の年齢>=16 -> "割引価格とならない (妻の年齢 < 16)"
No.2 : 20 33, 非正常系, 夫の年齢 +妻の年齢<=50 -> "割引価格とならない (夫の年齢 + 妻の年齢 > 50)"
No.3 : 0 18, 非正常系, 夫の年齢>=18 -> "割引価格とならない (夫の年齢 < 18)"

- "割引価格とならない(妻の年齢<16) "
-- On ポイント
No.1 : 20 15, 正常系, 妻の年齢>=16 -> "割引価格とならない(妻の年齢<16) "
No.2 : 18 10, 正常系, 夫の年齢>=18 -> "割引価格とならない(妻の年齢<16) "
No.3 : 40 10, 正常系, 夫の年齢 + 妻の年齢 <= 50 -> "割引価格とならない(妻の年齢<16) "

-- Off ポイント
=======省略=======
```

5.3 複数の関数を含む仕様

拡張した BWDM に、複数の関数を含む仕様 (コード 3.4 参照) を入力として適用した結果を、コード 5.4 に示す。コード 5.4 では、「成人判定」関数 (37 行目以降) だけでなく、「うるう年判定」関数 (1-35 行目) についてもテストケース生成が適切にできていることが確認できる。

したがって、拡張した BWDM は、既存の BWDM の問題点である、1 つの関数しかテストケース生成できないという問題点を解決し、複数の関数のテストケース生成ができることを確認できた。

5.4 定数定義ブロック内の定義を含む関数を含む仕様

拡張した BWDM に、定数定義ブロック内の定義を含む関数を含む仕様 (コード 3.6) を入力として適用した結果を、コード 5.5 に示す。コード 5.5 では、12 行目にて「数 $mod\ 2 = 0$」となっており、even を 2 に正しく変換できていることが確認できる。また、13 行目も同様に確認できる。また、入力と出力の組合せが正しいことから、テストケース生成が適切にできていることが確認できる。

したがって、拡張した BWDM は、既存の BWDM の問題点である、定数定義ブロック内の定義を含む関数を含む仕様のテストケース生成ができないという問題点を解決し、定数定義ブロック内の定義を含む関数を含む仕様のテストケース生成ができることを確認できた。

5.5 操作定義を含む仕様

拡張した BWDM に、操作定義を含む仕様 (コード 3.7 参照) を入力として適用した結果を、コード 5.6 に示す。コード 5.6 では、インスタンス変数「current_year」を「うるう年判定」操作の引数としてテストケース生成できていることが確認できる。また、入力と出力の組合せが正しいことから、適切なテストケース生成ができていることが確認できる。

したがって、拡張した BWDM は、既存の BWDM の問題点である、操作のテストケース生成ができないという問題点を解決し、操作のテストケース生成ができることを確認できた。

コード 5.4: 拡張した BWDM に複数の関数を含む仕様 (コード 3.4) を適用した際の出力

```
関数名 : うるう年判定
引数の型 : 年:int
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
年 : 2147483648 2147483647 -2147483648 -2147483649 3 4 5 99 100 101 399 400 401

記号実行情報
戻り値の数 : 4
制約 : 年 mod 400 = 0 and 年 mod 100 = 0and 年 mod 4 = 0, 戻り値 : "うるう年"
制約 : !( 年 mod 400 = 0 ) and 年 mod 100 = 0and 年 mod 4 = 0, 戻り値 : "平年"
制約 : !( 年 mod 100 = 0 ) and 年 mod 4 = 0, 戻り値 : "うるう年"
制約 : !( 年 mod 4 = 0 ) , 戻り値 : "平年"

境界値分析によるテストケース
No.1 : 2147483648 -> Undefined Action
No.2 : 2147483647 -> "平年"
No.3 : -2147483648 -> "うるう年"
No.4 : -2147483649 -> 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 : 500 -> "平年"
No.3 : 104 -> "うるう年"
No.4 : 1 -> "平年"

関数名 : 成人判定
引数の型 : 年齢:nat
戻り値の型 : seq of (char)
生成テストケース数 : 8件 (境界値分析:6/記号実行:2)

各引数の境界値
年齢 : 4294967295 4294967294 0 -1 20 21

記号実行情報
戻り値の数 : 2
制約 : 年齢 <= 20 , 戻り値 : "未成年"
制約 : !( 年齢 <= 20 ) , 戻り値 : "成人"

境界値分析によるテストケース
No.1 : 4294967295 -> Undefined Action
No.2 : 4294967294 -> "成人"
No.3 : 0 -> "未成年"
No.4 : -1 -> Undefined Action
No.5 : 20 -> "未成年"
No.6 : 21 -> "成人"

記号実行によるテストケース
No.1 : 1 -> "未成年"
No.2 : 21 -> "成人"
```

コード 5.5: 拡張した BWDM に定数定義ブロック内の定義を含む関数を含む仕様 (コード 3.6) を適用した際の出力

```
1
2  関数名 : 偶数判定
3  引数の型 : 数:int
4  戻り値の型 : seq of (char)
5  生成テストケース数 : 9件 (境界値分析:7/記号実行:2)
6
7  各引数の境界値
8  数 : 2147483648 2147483647 -2147483648 -2147483649 1 2 3
9
10 記号実行情報
11 戻り値の数 : 2
12 制約 : 数 mod 2 = 0 , 戻り値 : "偶数"
13 制約 : !( 数 mod 2 = 0 ) , 戻り値 : "奇数"
14
15 境界値分析によるテストケース
16 No.1 : 2147483648 -> Undefined Action
17 No.2 : 2147483647 -> "奇数"
18 No.3 : -2147483648 -> "偶数"
19 No.4 : -2147483649 -> Undefined Action
20 No.5 : 1 -> "奇数"
21 No.6 : 2 -> "偶数"
22 No.7 : 3 -> "奇数"
23
24 記号実行によるテストケース
25 No.1 : 2 -> "偶数"
26 No.2 : 1 -> "奇数"
```

コード 5.6: 拡張した BWDM に操作定義を含む仕様 (コード 3.7) を適用した際の出力

```
1
2  関数名 : うるう年判定
3  引数の型 : current_year:nat
4  戻り値の型 : seq of (char)
5  生成テストケース数 : 17件 (境界値分析:13/記号実行:4)
6
7  各引数の境界値
8  current_year : 4294967295 4294967294 0 -1 3 4 5 99 100 101 399 400 401
9
10 記号実行情報
11 戻り値の数 : 4
12 制約 : current_year mod 400 = 0 and current_year mod 100 = 0and current_year mod 4 = 0, 戻り値 : "うるう年"
13 制約 : !( current_year mod 400 = 0 ) and current_year mod 100 = 0and current_year mod 4 = 0, 戻り値 : "平年"
14 制約 : !( current_year mod 100 = 0 ) and current_year mod 4 = 0, 戻り値 : "うるう年"
15 制約 : !( current_year mod 4 = 0 ) , 戻り値 : "平年"
16
17 境界値分析によるテストケース
18 No.1 : 4294967295 -> Undefined Action
19 No.2 : 4294967294 -> "平年"
20 No.3 : 0 -> "うるう年"
21 No.4 : -1 -> Undefined Action
22 No.5 : 3 -> "平年"
23 No.6 : 4 -> "うるう年"
24 No.7 : 5 -> "平年"
25 No.8 : 99 -> "平年"
26 No.9 : 100 -> "平年"
27 No.10 : 101 -> "平年"
28 No.11 : 399 -> "平年"
29 No.12 : 400 -> "うるう年"
30 No.13 : 401 -> "平年"
31
32 記号実行によるテストケース
33 No.1 : 400 -> "うるう年"
34 No.2 : 100 -> "平年"
35 No.3 : 4 -> "うるう年"
36 No.4 : 1 -> "平年"
```

第 6 章

考察

本章では、拡張した BWDM の有用性について考察し、関連研究と、拡張した BWDM の問題点について述べる。

6.1 拡張した BWDM の有用性について

本節では、拡張した BWDM の有用性についての考察を述べる。

6.1.1 水準の積が膨大となるテストケース生成の比較検証

本研究で拡張した BWDM が、既存の BWDM に比べて、テストケース総数を削減できることを確認する。膨大な数のテストケースを生成するために、因子が 7 で水準が (6, 8, 6, 8, 8, 6, 6) の関数を持つ VDM++ 仕様を、既存の BWDM と拡張後の BWDM にそれぞれ適用する。適用した VDM++ 仕様を、コード 6.1 に示す。また、生成結果の比較を、表 6.1 に示す。実行環境は、macOS 10.13.6(CPU: Intel Core i5 2.3GHz, RAM: 16GB) である。比較に用いる式を、以下に示す。

$$\text{削減率 (\%)} = \frac{A - B}{A} \times 100 \tag{6.1}$$

A: 既存の BWDM によって生成したテストケース総数

B: 拡張した BWDM によって生成したテストケース総数

コード 6.1: 因子が 7 で水準が (6, 8, 6, 8, 8, 6, 6) の関数を持つ VDM++ 仕様

```
class ProblemClass
functions

problemFunction : nat*nat*nat*nat*nat*nat*nat -> seq of char
  problemFunction(a, b, c, d, e, f, g) ==
    if(a > 4) then
      if(b mod 10 = 3) then
        if(c < 13) then
          if(b > 11) then
            "a>4 and b>11 and c<13"
          else
            if(g < 11) then
              "g<11"
            else
              "a>4 and b>10 and c<13"
        else
          if(d > 10) then
            "d>10"
          else
            if(e < 10) then
              "e<10"
            elseif(f > 10) then
              "f>10"
            else
              "a>4 and b>10 and c>=13"
      else
        "a>4 and b<=10"
    else
      "a<=4";
end ProblemClass
```

表 6.1: 因子が 7 で水準が (6, 8, 6, 8, 8, 6, 6) の関数を持つ VDM++ 仕様 (コード 6.1) のテストケース生成結果の比較

	生成テストケース数	実行時間 (秒)
既存の BWDM	663,552	6.767146152
拡張した BWDM	78	0.88973152

表 6.1 および式 (6.1) より、生成テストケース数を $(663552 - 78)/663552 \times 100 = 99.98(\%)$ 削減できた。既存の BWDM では、膨大な数のテストケースを生成したのに対して、拡張後の BWDM では、実用的な数のテストケースを生成した。

したがって、拡張後の BWDM は、境界値分析結果から生成するテストケース数が組合せ爆発を起こす可能性を排除できたと言える。また、表 6.1 から、テストケース生成時間についても 86.85% 短縮できた。以上から、拡張後の BWDM は実用性が高いと言えることから、BWDM は有用性が向上したと考える。

6.1.2 人手によるドメイン分析テストのためのテストケース作成との比較検証

人手によるドメイン分析テストのためのテストケース作成と、拡張した BWDM によるドメイン分析テストのためのテストケース生成で、作成 (生成) に要した時間の比較検証を行った。その結果を、表 6.2 に示す。

対象とした VDM++ 仕様は、2.5 節で用いた コード 2.1 である。“割引価格となる” を期待出力に持つドメインに対するドメイン分析テストのためのテストケースを作成する時間を計測した。生成するテストケースとしては、以下を基準とした。

1. on ポイント、off ポイント、in ポイント、out ポイントを出力 (記述) する

2. on ポイント、off ポイント、out ポイントには、着目条件式も出力 (記述) する

3. off ポイントには、着目変数も出力 (記述) する

4. 各ポイントには、期待出力と正常系であるかどうかも出力 (記述) する

検証に参加したメンバーは本研究室の大学院生 3 人と学部 4 年生 1 人であり、普段からソースコードの読み書きを行い、基本的なプログラミングの知識を有している。VDM++ の文法の知識を持たない者も含まれるが、今回の検証に必要な文法は、事前に他の VDM++ の例を用いてレクチャーした。また、ドメイン分析テストのためのテストケース生成についても、事前に他の VDM++ 仕様とテストケースの例を用いてレクチャーした。

人手による検証では、コード 2.1 を印刷した紙を渡し、仕様を確認後、テストケースを書き始めてから、テストケースを記述し終えるのに要した時間を計測した。入力データと戻り値の組合

表 6.2: コード 2.1 のドメイン分析テストのためのテストケース作成に要した時間の比較

被験者	時間
被験者 A	8m 16s
被験者 B	10m 23s
被験者 C	30m(制限時間超過)
被験者 D	24m 04s

	時間
被験者 (平均)	18m 10s
BWDM	0m 15s

せが不正確な場合、間違いを指摘し、被験者が正しい組合せを記述した時点で時間計測終了とした。また、制限時間を 30 分とし、制限時間を超えた場合、その場で時間計測終了とした。

拡張した BWDM による検証では、コマンドライン上での命令操作で、拡張した BWDM によるテストケース生成を行うのに要した時間を計測した。また、実験に用いたコンピュータは、OS:macOS 10.14.5、CPU:2.3GHz Intel Core i5、メモリ:16GB である。

なお、Java の System.nanoTime[29] メソッドを用いて、命令操作を省いた純粋なテストケース生成処理に BWDM が要した時間を計測した結果、1.25 秒であった。

人手による作成と比較した結果、平均で 18 分程の時間短縮を確認できた。対象にした VDM++ 仕様には、VDM++ 独特の文法等は含まれないため、VDM++ に対する慣れなどの影響は無視できるものと思われる。また、人手によるテストケース生成の場合、ヒューマンエラーも見られた。具体的には、off ポイントの記述時に、条件式の解釈を間違え、誤った期待出力を記述してしまった。(例：入力 (17、 20) の期待出力を “遊園地チケットは割引価格とならない。(妻の年齢 < 16)” と記述した。) 仕様の規模が拡大すると、人手とコンピュータとの処理効率の差に加えて、ヒューマンエラーの有無などにより、テストケース生成に要する時間の差は更に拡大していくと思われる。以上から、拡張した BWDM は有用性が向上したと考える。

6.1.3 複数定義対応に関する評価

拡張した BWDM では、複数の関数のテストケースを生成できるようになり (5.3 節参照)、また、定数定義ブロックで定義した定数を、関数定義や操作定義内で参照できるようになった (5.4 節参照)。加えて、インスタンス変数を含む操作定義のテストケースを生成できるようになった (5.5 節参照)。

これにより、BWDM の対応範囲が広がり、より多くの VDM++ 仕様のテストケース生成が可能となった。したがって、拡張した BWDM の有用性が向上したと考える。

6.2 関連研究

ドメイン分析に基づいたテストケースを自動生成する手法としては、丹野らの研究 [30] がある。この手法では、変数同士に依存関係がある場合でも、制約ソルバ [25] を用いることで、依存関係を考慮した境界値等のテストケースを網羅的に生成する。入力として、ソフトウェアの設計情報をモデル化した設計モデルと呼ばれるテキストを入力する必要がある。設計モデルは [30] で定義されている。

設計モデルは、仕様書を元に人手で記述する必要があるため、設計モデルの作成に時間と手間がかかってしまうという問題がある。これに対して、拡張した BWDM は、VDM++ 仕様を元に、自動でテストケースを生成できるため、設計モデルを作成する必要がないという利点がある。しかし、仕様書が自然言語のみで記述されている場合、VDM++ 仕様を記述しなければ、拡張した BWDM を使うことができない。そのため、自然言語で記述された仕様書からドメイン分析テストのためのテストケースを作成する場合、設計モデルを記述するか、VDM++ 仕様を記述するかで対応が分かれることとなる。なお、設計モデルの記述と VDM++ 仕様の記述に必要な要素がほぼ同じなため、記述量はほとんど変わらないと考える。設計モデルは丹野らの手法でしか利用できないが、VDM++ 仕様を用いた場合は、BWDM 以外にも、VDMTools[21] や Overture IDE[22] などの支援ツールが揃っており、仕様の検証や記述を行いやすいという利点がある。

6.3 拡張した BWDM の問題点

以下に、今回拡張した BWDM の問題点を示す。

- 整数型以外の型に対応していない

  拡張した BWDM は、整数型である、int 型と nat 型と nat1 型のみにしか対応していない。実数値を表す real 型や有理数を表す rat 型、複数の型から構成される合成型などには対応していない。そのため、それらを用いた仕様からテストケースを作成できない。それらの未対応の型への対応は、VDM++ 仕様を静的解析する際に型情報を読み込み、境界値分析時に、

読み込んだ型情報から境界値の生成処理を BWDM に追加することによって、解決可能であると考える。

- 型定義に対応していない

  拡張した BWDM は、型定義ブロックに記述した型定義を用いた関数や操作のテストケースを生成できない。この問題は、型定義部で定義した型について静的解析を行い、境界値分析を行う処理の際に、それらの情報を用いて境界値を生成することで、解決可能であると考える。

- 関数定義と操作定義内で対応している構文が少ない

  拡張した BWDM は、if 条件式境界値の生成を VDM++ 仕様の関数定義内の if-then-else 式のみから行っており、それ以外の構文には対応していない。また、if 条件式も、不等号 (< 、<= 、> 、>=) と剰余 (*mod*) のみの対応である。if-then-else 式同様に、条件と動作の記述が可能な cases 式などに対応していないため、ツールの適用可能な範囲はまだ狭く、実用性が高いとは言えない。この問題は、新たな構文への対応のために、新たな構文の情報抽出方法と、境界値分析手法の提案を行うことで解決可能であると考える。

- インスタンス変数を操作する操作定義のテストケースを生成できない

  拡張した BWDM は、インスタンス変数を操作する操作定義のテストケース生成ができない。なぜなら、if-then-else 式を使用していない操作の入力と期待出力を推定できないからである。この問題は、インスタンス変数定義ブロック内で定義した不変条件に対応することで解決可能と考える。具体的には、不変条件を元に生成した境界値を入力とし、不変条件を満たすかどうかの真偽値を期待出力とするテストケースを生成することで、インスタンス変数を操作する操作定義のテストケースを生成できると考える。

- 他の関数または操作を呼び出す関数定義と操作定義のテストケースを生成できない

  拡張した BWDM では、他の関数または操作を利用する関数定義と操作定義のテストケースを生成できない。この問題は、抽象構文木解析処理において解析した関数、および、操作を Java コードに変換し、スタックを構築して関数呼び出しの仕組みを作ることで解決可能と考える。

第 7 章

おわりに

本研究では、BWDM の有用性の向上を目的として、BWDM の拡張を行った。具体的には、既存の BWDM が持つ 5 つの問題点を解決した。また、拡張後の BWDM の名称を BWDM(Verification tool for Vienna Development Method) に変更した。本研究で解決した 5 つの問題点を、下記に示す。

(a) 境界値分析時にすべての組合せを用いてテストケースを生成するため、組合せ爆発を起こす可能性がある

(b) 複数の変数を含む条件式を持つ関数のテストケースを生成できない

(c) 1 つの関数しかテストケースを生成できない

(d) 定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できない

(e) 操作のテストケースを生成できない

上記 5 つの問題点を解決するために、既存の BWDM に、以下の 3 つの機能拡張を行った。

- ペアワイズ法の適用によるテストケース数削減
- ドメイン分析テストの適用による複数変数を含む条件式を含む関数のテストケース生成
- 複数の定義への対応

まず、既存の BWDM における、境界値分析結果から生成するテストケース数が組合せ爆発を起こす可能性 (問題 (a)) の排除を目的として、BWDM の拡張を行った。拡張した BWDM は、VDM++ 仕様に対して境界値分析を行い、ペアワイズ法を適用し、テストケースを自動生成する。これにより、境界値分析結果から生成するテストケース数が組合せ爆発を起こす可能性を排除することができ、テスト工程の作業効率化を見込めると考えられる。

次に、本研究では、既存の BWDM における、複数の変数を含む条件式を持つ仕様に対してテストケース生成ができないという問題 (問題 (b)) の解決を目的として、ドメイン分析テストのためのテストケース生成を、BWDM に適用する拡張を行った。まず、複数の変数を含んだ条件式を解析できるように、構文解析部を拡張した。次に、ドメイン分析部を追加し、on ポイント、off ポイント、in ポイント、out ポイントを生成できるようにした。また、SMT ソルバを用いて、充足可能性問題を解けるようにした。さらに、テストケースに、正常系判定値、着目条件式、そして着目変数の情報を含めるために、テストケース生成部を拡張した。今回の拡張により、既存の BWDM では生成できなかった、複数の変数を含む条件式を持つ VDM++ 仕様の構文解析と、ドメイン分析テストのためのテストケース生成ができるようになったことを確認した。また、20 行ほどの VDM++ 仕様に対して、ドメイン分析テストのためのテストケース生成に要した時間を人手と比較検証した結果、18 分程の時間短縮を確認できた。

加えて、本研究では、既存の BWDM における、1 つの関数しかテストケースを生成できないという問題 (問題 (c))、定数定義ブロック内の定義を含む関数を含む仕様のテストケースを生成できないという問題 (問題 (d))、操作のテストケースを生成できないという問題 (問題 (e)) の解決を目的として、BWDM における構文解析部の抽象構文木解析処理を修正し、BWDM を拡張した。これにより、BWDM の対応範囲が広がり、より多くの VDM++ 仕様のテストケース生成が可能となった。

以上により、本研究で行った拡張によって、既存の BWDM が持つ 5 つの問題点を解決し、BWDM の有用性が向上したと言える。また、BWDM がテストケースを自動生成できる VDM++ 仕様の構文が増えたことにより、テストケース作成作業の自動化率が向上するため、テスト工程の作業効率が向上したと言える。

以下に、今後の課題を示す。

- 整数型以外の型への対応

現状、実数を表す real 型や、有理数を表す rat 型、複数の型から構成する合成型などの、VDM++ の多くの型に未対応である。この問題は、VDM++ 仕様を静的解析する際に型情報を読み込み、境界値分析時に、読み込んだ型情報から境界値の生成処理を BWDM に追加することによって、解決可能であると考えられる。

- 型定義の対応

  拡張した BWDM は、型定義ブロックに記述した型定義を用いた関数や操作のテストケースを生成できない。この問題は、型定義部で定義した型について静的解析を行い、境界値分析を行う処理の際に、それらの情報を用いて境界値を生成することで、解決可能と考える。

- 関数定義と操作定義内で利用できる構文の対応範囲拡大

  拡張した BWDM は、if 条件式境界値の生成を VDM++ 仕様の関数定義内の if-then-else 式のみから行っており、それ以外の構文には対応していない。この問題は、対応していない構文の情報抽出方法と、境界値分析手法の提案を行うことで解決可能と考える。

- インスタンス変数を操作する操作定義のテストケースの生成への対応

  拡張した BWDM は、インスタンス変数を操作する操作定義のテストケース生成ができない。この問題は、インスタンス変数定義ブロック内で定義した不変条件に対応することで解決可能と考える。

- 他の関数または操作を呼び出す関数定義と操作定義のテストケース生成への対応

  拡張した BWDM では、他の関数または操作を利用する関数定義と操作定義のテストケース生成ができない。この問題は、抽象構文木解析処理において解析した関数、および、操作を Java コードに変換し、スタックを構築して関数呼び出しの仕組みを作ることで解決可能と考える。

謝辞

本研究において、数多くのアドバイス、叱咤激励、知識、そして食料の差し入れをいただきました、宮崎大学工学教育研究部の片山徹郎教授に、心から感謝申し上げます。

また、本論文の執筆にあたり、時にはユーモアを交え、貴重なご意見とご指摘を頂きました、宮崎大学工学教育研究部の山森一人教授と油田健太郎准教授に、深く感謝いたします。

そして、既存の BWDM についての知識、ノウハウを教えていただいた、偉大なる BWDM の生みの親、立山博基氏に、次あったときにありがとうございましたとお伝えしたいです。

他にも、何の知識もない私に、技術的知識、研究とは何かなどの、さまざまなことを教えていただき、時には、アニメ鑑賞会、漫才、ジョイフルを共に行っていただいた、片山徹郎研究室の偉大なる先輩、偉大なる同輩、偉大なる後輩の皆様、ありがとうございました。皆様のおかげで、辛いとき、悩めるときも、前向きに生活することができました。

最後に、私を成長させてくれた、すべての方々、本当にありがとうございました。6 年間という長い長い大学生活を、とても満足のいくものとすることができました。胸を張って修了できます。

参考文献

[1] 失敗知識データベース. 失敗事例 – みずほファイナンシャルグループ大規模システム障害. http://www.shippai.org/fkd/cf/CA0000623.html. Accessed: 2020-1-18.

[2] IPA 情報処理推進機構. なぜ形式手法か. http://sec.ipa.go.jp/users/seminar/seminar_tokyo_20130918-1.pdf. Accessed: 2020-1-18.

[3] 荒木啓二郎, 張漢明. プログラム仕様記述論. オーム社, 2002.

[4] Overture Project. Manuals. http://overturetool.org/documentation/manuals.html. Accessed: 2020-1-19.

[5] Hiroki Tachiyama, Tetsuro Katayama, Tomohiro Oda. Automated generation of decision table and boundary values from VDM++ specification. *The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering Technical Report Series, No. CS-TR-1513-2017*, pp. 89–103, 2017.

[6] 立山博基, 片山徹郎. VDM++ 仕様を用いたテストケース自動生成ツール BWDM における if-then-else 式の構造認識手法の提案. ソフトウェアエンジニアリングシンポジウム 2017 論文集, pp. 130–137, 2017.

[7] 平木場風太, 片山徹郎. VDM++ 仕様を対象としたテストケース自動生成ツール BWDM における境界値分析結果へのペアワイズ法の適用. 平成 30 年度 電気・情報関係学会 九州支部連合大会, pp. 135–136, 2018.

[8] Futa Hirakoba, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, Kentaro Aburada, and Naonobu Okazaki. Prototype of Test Cases Automatic Generation Tool BWDM Based on

Boundary Value Analysis with VDM++. *The 2019 International Conference on Artificial Life and Robotics(ICAROB 2019)*, pp. 161–164, 2019.

[9] 平木場風太, 片山徹郎. VDM++ 仕様を対象としたテストケース自動生成ツール BWDM への PICT の適用. 宮崎大学工学部紀要 第 48 号, pp. 143–148, 2019.

[10] 平木場風太, 片山徹郎. VDM++ 仕様を対象としたテストケース自動生成ツール BWDM へのドメイン分析テストの適用. ソフトウェアエンジニアリングシンポジウム 2019 論文集, pp. 126–134, 2019.

[11] Tetsuro Katayama, Futa Hirakoba, Yoshihiro Kita, Hisaaki Yamaba, Kentaro Aburada, and Naonobu Okazaki. Application of pairwise testing into BWDM which is a test case generation tool for the VDM++ specification. *Journal of Robotics, Networking and Artificial Life*, Vol. 6, No. 3, pp. 143–147, 2019.

[12] korosuke613. BWDM: Verification tool for vienna development method. `https://github.com/korosuke613/BWDM`. Accessed: 2020-1-27.

[13] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pp. 254–261. IEEE, 1998.

[14] D. Kuhn, Dolores Wallace, and A.M. Jr. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, Vol. 30, pp. 418 – 421, 2004.

[15] Microsoft. Pairwise independent combinatorial testing. `https://github.com/microsoft/pict`. Accessed: 2020-1-18.

[16] korosuke613. pict4java. `https://github.com/korosuke613/pict4java`. Accessed: 2020-1-27.

[17] Lee Copeland(訳:宗雅彦). はじめて学ぶソフトウェアのテスト技法. 日経 BP 社, 2005.

[18] ISTQB. Istqb glossary. `https://glossary.istqb.org`. Accessed: 2020-1-18.

[19] Overture Project. http://overturetool.org/method/. Accessed: 2020-1-22.

[20] International Organization for Standardization. ISO/IEC JTC 1/SC 22/WG 19. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, 1996.

[21] FMVDM. VDMTools. http://fmvdm.org/vdmtools/index.html. Accessed: 2020-1-18.

[22] The Overture Project. Overture Tools. http://overturetool.org. Accessed: 2020-1-18.

[23] nickbattle. VDMJ. https://github.com/nickbattle/vdmj. Accessed: 2020-1-18.

[24] 五味弘, 辻村浩, 小池宏道. 直交表とオールペア法の並行運用によるソフトウェアテスト. JaSST'14 Tokyo. http://jasst.jp/symposium/jasst14tokyo/pdf/C4-1-1.pdf. Accessed: 2020-1-19.

[25] 宋剛秀, 田村直之. SAT ソルバーの最新動向と利用技術. 第 19 回プログラミングおよびプログラミング言語ワークショップ PPL2017. https://ppl2017.ipl-e.ai.kyutech.ac.jp/slides/ppl2017_c4_soh.pdf. Accessed: 2020-1-18.

[26] java-native access. Java native access (jna). https://github.com/java-native-access/jna. Accessed: 2020-1-18.

[27] Kotlin Foundation. Kotlin. https://kotlinlang.org/. Accessed: 2020-1-18.

[28] Z3 Prover. Z3. https://github.com/Z3Prover/z3. Accessed: 2020-01-18.

[29] Oracle. クラス System. https://docs.oracle.com/javase/jp/8/docs/api/java/lang/System.html. Accessed: 2020-1-18.

[30] 丹野治門, 張暁晶. ドメインテスト技法に基づく網羅的なテストデータ自動生成手法の提案. 研究報告ソフトウェア工学 (SE), Vol. 2014, No. 6, pp. 1–8, 2014.