A-020

# モデル規範型形式手法 VDM++ による 組織の整合性・制約条件に対する記述と検証手法

## Description and Verification Methods for Organizational Consistency and Constraints using the Model Reference Formal Method VDM++

水野 駆 †　　和崎 克己 ††
Kakeru Mizuno　Katsumi Wasaki

### 1　はじめに

手続き型のテストではシステムのある部分にエラーがあることや有限個の組み合わせの中でエラーがないことは証明できるが，システム全体として仕様に反していることは証明できない．形式手法を使うことで数学的に設計対象が仕様に則っていることを証明できる．本研究では，モデル規範型形式手法によりオブジェクト指向性がある対象の抽象的な仕様を厳密に記述し，複雑な条件は一階述語論理式で列挙して条件を変えながらテストすることを目指す．設計対象としては，複雑制約が課されている組織の勤務シフト表の検査システムとした．

### 2　VDM とは

VDM とは 1970 年代に IBM のウィーン研究所において開発されたモデル規範型の形式手法である．VDM を用いて上流工程時の形式的な仕様を厳密に記述できる [1][2]. 形式仕様記述とは，システムの構造や機能に関する仕様や設計を厳密かつ抽象的にモデル化・記述し，分析を行うことを言う．VDM とは手法の名前であり，記述言語としては VDM-SL または VDM++ を用いる．

### 3　複雑な制約条件が課される課題に対するモデル規範型アプローチ

本研究における要求とは,「その組織の仕事とメンバーの人数を含んでいる概念」のことである．勤務シフト表とは，時系列順で部門に所属するメンバーが表になっているものである．対象としている勤務シフト表の検査システムにおいては，ある職場は複数の部門を所有する．その部門に対して上で述べた要求をいくつか出すという構図になる (図 1). 要求の内容によって様々なバリエーションの制約条件が生まれる．

#### 3.1　核となる制約条件

勤務者や組織に課される様々な制約条件を一階述語論理式で表現する．その中で核となる制約条件を自然語で記述すると例えば以下のようになる．

a1. ある職場の部門に要求される人数分が勤務シフト表に存在する．
a2. ある職場の部門に一人は仕事に対する熟練度が最高な人が存在する．
a3. ある職場の部門に仕事に対する能力が無い人が含まれない．

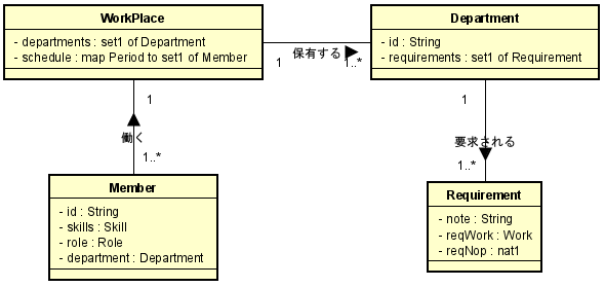


図 1　勤務シフトにおける制約条件を検査するシステムのクラス図

```
protected CheckReqNop: set1 of Department * Period
          * map Period to set1 of Member -> bool
CheckReqNop(deps,p,sch) ==
  (forall d in set deps &
    forall r in set d.GetReqs() &
      r.GetReqNop() <= GetNopByReq(r, sch(p)));
```

図 2　核となる制約条件「a1」を VDM++ で記述

#### 3.2　習熟度と制約条件

核となる制約条件の他に，メンバーに対して習熟度レベルを割り振り，その習熟度に応じて勤務シフト表におけるメンバーのバランスを考える制約条件を記述する．なお，習熟度は 0 から 5 の 6 段階評価とし，それぞれにラベルを割り振る．0 はその仕事に対する能力なし，1 はビギナー，5 はリーダーそれ以外はベテランとする．

b1. その区切りの時点で，リーダー + ベテラン ⩾ ビギナーを満たす．
b2. その区切りの時点で，ベテラン ⩾ ビギナーを満たす．
b3. その区切りの時点で，ベテラン ⩾ ビギナー ×2 を満たす．

区切りとは，主に週や時刻単位などの時間的な区切りを想定している．

```
CheckNopBlance1: set1 of Department * Period
     * map Period to set1 of Member -> bool
CheckNopBlance1(deps,p,sch) ==
  (forall d in set deps &
    forall r in set d.GetReqs() &
      d.GetValidFuncBool()("CheckNopBlance1") =>
        (GetNopBySkills(d, sch(p), r, <Leader>)
          + GetNopBySkills(d, sch(p), r, <Veteran>)
            >= GetNopBySkills(d, sch(p), r, <Beginner>)));
```

図 3　習熟度と制約条件「b1」を VDM++ で記述

† 信州大学大学院総合理工学研究科, Graduate School of Science and Technology, Shinshu University
†† 信州大学工学部電子情報システム工学科, Department of Electrical and Computer Engineering, Faculty of Engineering, Shinshu University

### 3.3　整合性の判定方法

現時点では制約条件の結びつきは AND 条件でそれぞれ結びついていて，積の左結合により先頭から評価していく「ふるい落とし」の形を採用している．3.1 と 3.2 の a1 から b3 まで順に，if-elseif 文で制約条件の判定を行う．制約に反した部分でエラーとなるのでふるい落としとなっている．それぞれの制約条件を評価し，反していたら false を返し，すべてパスしたら true を返す関数となる．if の評価部分に制約条件を有効化/無効化するスイッチ (sw1〜6) を設けることで職場や部門ごとに制約条件を変えながらテストすることができる．以下は，「ふるい落とし」の形を表した疑似コードである．

**function** WorkPlaceValid
  **if not** ($a1$ **or not** $sw1$) **then return** false
  **else if not** ($a2$ **or not** $sw2$) **then return** false
  **else if not** ($a3$ **or not** $sw3$) **then return** false
  **else if not** ($b1$ **or not** $sw4$) **then return** false
  **else if not** ($b2$ **or not** $sw5$) **then return** false
  **else if not** ($b3$ **or not** $sw6$) **then return** false
  **else return** true
  **end if**
**end function**

```
class TestInshoku

instance variables
req1: Requirement := new Requirement(mk_token("id_1"),"料理",1);
req2: Requirement := new Requirement(mk_token("id_2"),"皿洗い",1);
req3: Requirement := new Requirement(mk_token("id_3"),"清掃",1);
req4: Requirement := new Requirement(mk_token("id_3"),"清掃",2);
req5: Requirement := new Requirement(mk_token("id_4"),"レジ打ち",1);
req6: Requirement := new Requirement(mk_token("id_5"),"接客",2);

kitchen: Department := new Department("1",{req1,req2,req3}
,{"CheckLeaderBySkills" |-> false});
hall: Department := new Department("2", {req4,req5,req6},{"None" |-> false});

A : Member := new Member("1", {mk_token("id_1")|->0}, <ビギナー>, kitchen);
B : Member := new Member("2", {mk_token("id_2")|->1}, <ベテラン>, kitchen);
C : Member := new Member("3",
{mk_token("id_1")|->4,mk_token("id_2")|->4,mk_token("id_3")|->4}, <リーダー>, kitchen);
D : Member := new Member("4",
{mk_token("id_3")|->5,mk_token("id_4")|->5,mk_token("id_5")|->5}, <リーダー>, hall);
E : Member := new Member("5",
{mk_token("id_3")|->1,mk_token("id_4")|->1}, <ベテラン>, hall);
F : Member := new Member("6", {mk_token("id_5")|->1}, <ベテラン>, hall);
G : Member := new Member("7", {mk_token("id_5")|->1}, <ビギナー>, hall);

inshoku : WorkPlace := new WorkPlace({kitchen, hall},{
                        "CheckLeaderBySkills" |-> true,
                        "CheckZeroBySkills" |-> true,
                        "CheckNopBlance1" |-> true,
                        "CheckNopBlance2" |-> true,
                        "CheckNopBlance3" |-> false},
                        {
                        p1 |-> {C,D,F},
                        p2 |-> {C,D,G},
                        p3 |-> {B,C,D,E,F}});

values
p1: CommonType`Period = "Monday";
p2: CommonType`Period = "Tuesday";
p3: CommonType`Period = "Wednesday";
p4: CommonType`Period = "Thursday";
p5: CommonType`Period = "Friday";

operations
pure public Run: () ==> map CommonType`Period to set of Member
Run() == izakaya.GetSchedule();

end TestInshoku
```

図 4　飲食店勤務シフトの VDM++ での記述例

## 4　勤務シフトの整合性チェックシステム

以下は勤務シフトシステムを構成する要素である．

- 職場の情報 DB
- 勤務シフト表
- 検証にかけるために利用する GUI
- シフトの仕様を記述した VDM Project

**(1) 勤務シフトシステムの記述**
GUI と DB の連携，Excel ファイルの読み込み，VDM++ のテストファイルの生成までを Python で実装した．GUI 部分と DB 連携部分は，それぞれ Python のライブラリである Tkinter と SQLite を使用した．VDM Project の記述と検証は，統合開発環境である Overture を使用した [3].

**(2) 勤務シフトシステムの概要**
作成した GUI からチェックに必要な各職場の情報を DB に保存できる．そして，別に作成した勤務シフト表 (Excel ファイル) を読み込み DB の情報と併せてその職場のテストファイルを自動で作成できる．さらに，そのテストファイルの実行後にメッセージを GUI に表示する．

**(3)VDM++ で書かれたテストファイル**
対象として飲食店アルバイトを例に，勤務シフトのチェックを行った．図 4 が実行するためのテストファイルである．本システムに必要なメンバのインスタンス化を行い，最後に「inshoku」をインスタンス化している．もし，シフトの整合性チェックにおいて制約条件に背く部分があった場合，インスタンス化は失敗しエラーを取得することができる．職場のインスタンス化時の引数として，三つ必要なものがある．一つ目は，その職場に存在する部門のインスタンスである．二つ目は，制約条件の関数名と制約条件を加味するかどうかの真偽値の写像 (スイッチ部分) である．指定しない場合デフォルトが適応されるが，指定した場合その制約条件をスキップできる．これは仕様を直接編集することなく複数の条件でのテストを可能にする．三つ目は，時間的区切りとメンバーの集合の写像である．この部分が，勤務シフト表に対応している部分となる．

## 5　まとめと今後の課題

本研究でモデル規範型形式手法 VDM++ によって複数の制約を課されているモデルである，組織の勤務シフトについて仕様を記述した．また，ローコードでの検証を可能にするための検証環境を構築し，シミュレーションまで行った．検証に必要な組織の情報は DB に保存し，勤務シフト表を作成し検証するところまでノーコードで実装した．制約をスイッチできる部分の GUI からの変更は未実装であるが，テストファイルの一部分を編集することでスイッチすることが可能である．また，組織全体に制約を課していたが部門ごとに不要な制約はカットすることも可能である． 本システムに存在する制約条件で職場が成り立つことを仮定している．もし制約条件間での包含関係の不整合や矛盾が起きていると，成立することがないモデルとなってしまう．それを防ぐための，制約条件間の数学的な検証はできていない.VDM++ での実現は困難であると考えられるので別の定理証明系を導入することが今後の課題である．

### 参考文献

[1] 石川冬樹（2011），VDM++ における形式仕様記述，近代科学社
[2]「VDM++ による形式仕様記述」Web サイト
http://research.nii.ac.jp/~f-ishikawa/vdm/overture.html
[3] OvertureTool https://www.overturetool.org/