VDM++仕様・Javaコード間のカスタマイズ・トレース可能な変換

石　川　冬　樹[†1]

形式仕様の幅広い活用のためには，様々な実装方針に対応した実装を効率よく得て，以後変更の際にも，仕様と実装との対応を把握し維持していく必要がある．本論文では，変換ルールと双方向変換理論に基づく，VDM++仕様から Java コードへの変換の枠組みを提案する．この枠組みでは，コード生成のカスタマイズ，共通テストによる仕様とコードとの検証対応，また仕様とコードとのトレーサビリティ管理を支援する．

Customizable and Traceable Transformation between VDM++ Specification and Java Program

Fuyuki Ishikawa[†1]

For wide applications of formal specification, it is essential to derive the implementation with various strategies efficiently, then understand and maintain the correspondences between the specification and the implementation through later modifications. This paper proposes a framework for transformation from a VDM++ specification to a Java program based on transformation rules and a bidirectional transformation theory. The framework supports customization of code generation, corresponding verification of specification and code, and traceability management of specification and code.

1. はじめに

開発の早い段階における成果物の品質を高め，手戻りのコストを防ぐことは重要と考えられてきた．このためのアプローチの一つである形式仕様記述においては，システムの状態や操作に関する仕様を形式的にモデル化・記述し，分析を行う．形式仕様記述の代表的な手法の一つである VDM は，プログラムに近い文法での記述や，インタプリタによる実行・テストを通した分析を行うため，一般的な開発者にとって導入しやすいライトウェイトなものである[1),2)]．形式仕様記述においては，開発の早い段階における効率的な記述・分析のため，抽象データ型や宣言的な構文が利用され，不要な実現詳細を捨象しての記述・分析を行う．

形式仕様記述の活用においては，最終的な成果物である実装コードを，効率的に，信頼できる形で得ることが重要である．また，仕様記述と実装コードとの間の対応関係を明確に把握するとともに，変更の際にも整合性ある形で維持していく必要がある．

本研究では上記の過程を支援するために，VDM++仕様から Java コードへの変換ルールとして実装方針を明示的に記述するアプローチ，およびそれに基づくコード生成の枠組みを提案する．このアプローチは下記の 3 つの観点における問題解決を目指している．

コード生成のカスタマイズ Java コードを得る際には，VDM++仕様に記述された決定事項を満たしつつ，実現方針を定め反映する必要がある．ここで自動コード生成を用いると，実装方針を反映できず特に非機能的要求を満たせない．一方で，人間がすべてコード記述を行うと，多くの共通することを形式的に改めて書き直すコストが高い．これに対し本研究では，変換ルール，特に部分的に上書き可能なものを用いてコード生成を制御することにより，部分的にコード生成をカスタマイズすることを可能とする．

共通テストケース生成 B メソッド[3)] 等一部の形式手法では，証明と詳細化を通して正しいことが保証されたコードを生成する．一方，VDM のツールにおいてはライトウェイトな利用を想定し，インタプリタ実行を通したテストを主な検証手段とし，段階的詳細化も想定していない．本研究においても，Java コードへの比較的自由な構文上の変換を許し，VDM に対して設定し確認したテストケースと同等のテストケースを，Java コードに対しても生成することによる検証を行うことを考える．この際に，変換ルールとして明示された VDM++仕様と Java コードとの差異に関する情報を利用する．

トレーサビリティ 様々なソフトウェアシステム開発への適用を考えると，仕様の変更や派生開発への対応，また実装時に見つかった誤りに対応した修正を行う必要がある．この際には仕様と実装の差異を踏まえつつ，それらの整合性を保って変更を反映する必要がある．本研究では，仕様と実装の差異を変換ルールとして明示しておくことにより，片方のみに存在する要素が変換によって上書きされ消えてしまうといった事態を防ぐ．また，変換ルールの実現においては双方向グラフ変換理論[4)] を用い，どの変換ルールを通して VDM++仕様と Java コードのどの部分がどの部分に対応しているかを把握できるようにするとともに，

†1 GRACE センター，国立情報学研究所
GRACE Center, National Institute of Informatics

```
1  class TestClass
2  instance variables
3    private values : set of real;
4    private user : token;
5    private state : history;
6    inv history.isValid();
7  operations
8    public extract :
9      real ==> set of real
10   extract(x) ==
11     return
12       { v | v in set values & v > x }
13   pre x > 0;
14 end TestClass
```

```
public class TestClass {
  private values : HashSet<Double>;
  private user : User;
  private static Logger logger;

  public Set<Double> extract(double x)
    throws InvalidArgException{
    if (!x > 0) {
      throw new InvalidArgException();
    }
    HashSet<Double> ret
      = new HashSet<Double>();
    for(double v: values){
      if(v > x) { ret.add(v); }
    }
    return ret;
  }
}
```

図 1　VDM++による形式仕様記述と対応する Java コードの例
Fig. 1　Example of Formal Specification in VDM++ and Corresponding Java Program

Java コード上の変更も VDM++仕様に反映させることもできるようにする．

本論文ではまず第 2 章において，VDM における抽象化について説明する．次に提案フレームワークの概要を第 3 章で示した後，変換ルール言語とそれに基づいた変換機構，およびそれらの実装についてそれぞれ第 4 章，第 5 章，第 6 章にて説明する．最後に第 7 章，第 8 章において，提案アプローチの利点・限界について議論しまとめる．

2. 背　　景

VDM++は手法としての VDM を実現する言語の一つであり，オブジェクト指向に基づく仕様記述を行うことができる．VDM++では**図 1** の左側に示す例にあるように，変数とメソッド（操作）を含むクラスを定義する．この際にはシステムが満たすべき性質として，不変条件（6 行目）や事前条件（10 行目）等も定義する．このような形式記述により曖昧さや不完全さが取り除かれ，不整合等も顕在化しやすくなる．また，型チェックやインタプリタを用いたテスト，静的解析，定理証明等の分析が，特に自動的に，行えるようになる．

図 1 の例を用いて，VDM++仕様における抽象化について下記に説明する．

- 3 行目では *set of real* という抽象的な型を用いて変数を定義している．仕様の時点では一般に，Java での *HashSet<Double>* といったように，メモリ上にどうデータを配置し操作するかについて定める必要がないためである．
- 4 行目では，*token* 型の変数 *user* を定義している．*token* 型は，等値判定のみの対象となる識別子を表すために用意された型である．実装においては，ユーザ情報は氏名，住所等多くの情報を含み，またユーザの識別子も特定の形式を持つかもしれない．しかしそういった詳細は，機能の分析には本質的でない場合には捨象されうる．
- 5，6 行目では，変数 *history* とそれに関する不変条件を定義している．例えば，受け取ったリクエストから一つずつ選んでの処理が定義されているとする．その正当性は，便宜的に処理されたメッセージの履歴を保持し，個々の待ち時間や優先度等を調べることにより判定できるが，実現上は不要のため実装には現れないかもしれない．
- 8〜10 行目では操作（メソッド）を定義している．その振る舞い（9 行目）では，ループ処理等のアルゴリズムは捨象され，集合から条件を満たす値を抜き出す宣言的な構文が用いられている．10 行目にて定義されているメソッドの事前条件は，一般に実装においては，現れないか，操作開始時にチェックを行い成り立たない場合に例外を投げるような処理として記述される．
- 実装プログラムにおいては，ロガー変数およびその呼び出し（ロギング）等，VDM++仕様に現れない変数やメソッドが定義される可能性がある．

図 1 の右側は，上記のような VDM++仕様における抽象化方針を踏まえて得られる Java プログラムの一例である．文法の違いだけではなく，本質的に新しい情報，すなわち実装方針に関する決定が反映されている．具体的には，型が具体化されており（*HashSet*，*User*，*double*），変数が追加または削除されている（*logger*，*history*）．また操作定義においては，事前条件が引数チェックの振る舞いに変わり，宣言的な記述がループに展開されている．

このような VDM による仕様モデルと実装との対応を明確にし管理していくためには，下記のような側面を明示的に区別し，管理していく必要がある．

- どの側面・要素が本質的な決定であり，そのまま，もしくは（実現型など）追加の決定を伴って，実装コードに引き継がれるのか．
- どの側面・要素が，検証等のための仮のものであり，実装コードでは用いられないのか
- どの側面・要素が実装コードのみにおいて導入されるのか（ロガー等）

VDM に関する既存ツールでは上記のような差異に関する支援が考慮されていない．VDM-Tools[5] におけるコード生成器では，複数のオプションが設定可能であるのみで，プロジェクトごとに固有の実装方針を反映することはできない．すると生成した実装コードを書き換

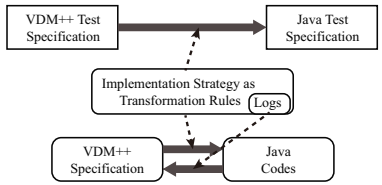

図 2　提案フレームワークにおける変換機能
Fig. 2　Transformation Features of Proposed Framework

える必要が発生しうるが，その場合はコード再生成の際の上書きを防ぐため，該当部分を生成の対象から外し，手動で対応関係を管理する必要がある．このため，変数定義やメソッドのシグネチャのみをスケルトンとして生成することも可能となっている．しかし上記の例題に見るように，インターフェースレベルでも VDM++仕様と Java コードの間には差異が存在する．例えば，近年注目されている UML クラス図と VDM++仕様との変換ツール[5),6)]を用いると，VDM++の語彙を用いた抽象モデルに関するクラス図を作成，VDM++仕様記述と連動することができる．しかし，通常の開発プロセスで用いるような，Java 等の語彙を用いた実装詳細を含むクラス図との関連をどう明確にし管理していくが不明である．

3. 提案する枠組みの概要

第 1 章にて述べたように，本研究においては，VDM++仕様から Java コードを実装方針に応じてテストケースも含めて得るとともに，その対応付けを明確に把握，管理しつつ変更を加えていくための枠組みを提案する．**図 3** に提案フレームワークの扱う変換を示す．

VDM++仕様は，多少の抽象化を伴い，ただし実装コードに反映される決定事項を多く含んだ形で（実装コードと同様な構造にて）記述されるものとする．記述した VDM++仕様に対しては，型チェックやレビュー，テスト等の分析を行い適宜修正する．これらの過程は通常の VDM の利用と同等であり，既存の VDM ツール（エディタ，インタプリタ，デバッガ，テストフレームワーク，UML エディタ，UML-VDM 等）を用いることができる．

提案フレームワークではその後，デフォルトのコード生成をカスタマイズしたい場合の実装方針として変換ルールの記述を行い，それを用いて Java コードを生成する．フレームワーク内部では，この変換ルールは構文木に対する処理として実現されており，実際に変換を行うとともに，構文木上での変換前後のノードの対応を保持するようになっている．また，変換ルールを用いて VDM++におけるテスト仕様（実行可能な VDM++仕様としてのテストケース定義）から，同等な Java テストコードを生成することもできる．これにより，VDM++仕様において確認した性質を，Java コードにおいても確認することができる．

VDM++仕様において変更があった場合，もしくは実装方針としての変換ルールを変更した場合，Java プログラムは再度生成される．Java コード独自の要素等は変換ルールとして明示的に記述されているので，変換結果にはそれらも含まれる（上書きされて消えてしまうことはない）．また，Java コード記述後に見つかった誤りの修正等に応じ Java コード側に変更があった場合でも，基本的には VDM++仕様側にその変更を反映することができる．ただしその反映方法が一意に定まらない場合は開発者が明示的に指示する必要がある．

本論文では変換ルールが明示的に与えられたときに，上記のような作業を支援する枠組みを扱う．既存の VDM++仕様と Java コードから，もしくはそれらに対する開発者の自由な編集結果から変換ルールを抽出する等，より発展的な支援については対象外とする．

4. 変換ルール言語

4.1 アプローチ

提案フレームワークにおける変換ルールでは，VDM++仕様の要素をどう Java コードの要素に構文的に変換するかを定義する．変換ルールは実装方針を表現するものであり，純粋な文法の差異は別に扱われる．例として VDM++仕様における下記の変数定義を考える．

```
private x : real;
```

もし変換ルールを与えないとすると，この記述要素は下記の Java コードに変換される．

```
private real x;
```

このように，同等な構文木と個々の文法に合わせたテキスト表現との対応は，変換ルールではなくシリアライザ・デシリアライザ（パーサ）が扱う．

上記は説明のために用いたが，実際には *real* という型は Java には存在しない．Java では，数学的概念としての実数ではなく，*double* や *float*，もしくはユーザ定義のクラスのいずれかとして実現する必要がある．もしすべての *real* 型を *double* 型として実装するとした場合の変換ルールは下記のようになる．

```
type-implement: real by double
```

このルールを導入することにより，上記の変換結果は次のように変わる．

```
private double x;
```

ルールにおける最初の要素（*type-implement*）は実装方針の指定パターンを表している（本章の後半で詳述する）．実際には，開発者がルールを指定しなくとも正しい Java コードが

得られるよう，デフォルトのルールを定めている．現在は，既存のコード生成器[5] に従い，上記の *double* を用いたものをデフォルトとしている．

もしも例外的に，変数 x に対してのみ *float* 型を用いたいとする．この際には，開発者は下記のルールを与えデフォルトを上書きすることができる．

```
type-implement: real by float in variable x
```

このルールは最初のものと同じように書かれているが，どの変数を対象とするかが明示されている．このように，適用範囲を明示した上書きによるカスタマイズが可能となっている．

4.2 変換ルールの記述能力

上述のアプローチに従い，既存の VDM 文献（主に書籍[1),2)]）等から抽出した抽象化方針に対応するよう変換ルールの記述能力を定めている．本論文では詳細は省略するが，上述のような具体型の指定のほか，変数やメソッド，メソッドの内の文の追加や削除を行うこともできる．また，不変条件等の性質記述を，チェックして例外を投げるような処理や，コメントに変換することもできる．こういったルールの適用対象については，個別のクラス，変数やメソッドを指定するほか，例えば全ての public 要素といった指定も可能である．

4.3 テストケースの継承

これまでに述べた変換を通し，各テストケースを表す VDM++のテスト仕様（実行可能なテスト動作の定義）を，同等なテストケースを実行する Java のテストコードに変換することができる．ここでメソッド呼び出しの引数が追加されている場合には，その値を各テストケースにおいて定める必要がある．この情報は，メソッド呼び出しを行っている文の置き換えを行うルールとして指定することもできるが，設定ファイルにてまとめて指定することもできるようになっている．例えば,「あるテストケースを実現する特定のクラスにおいて，メソッド呼び出しの追加された引数の値を常にある値に設定する」といった記述が可能である．こういった設定の詳細については，著者の以前の取り組みにおいて議論している[7)]．

5. 変 換 機 構

5.1 基 本 方 針

第 2 章にて述べた既存のコード生成器の問題を受け，提案フレームワークの変換機構においては下記のような性質を満たすことを目指す．

> 生成された Java コードに変更を加えた際，それを VDM++仕様に逆変換し，さらにそのまま Java コードを生成すると同等の Java コードが得られる．

このため提案フレームワークにおいては，同等の性質を保証するグラフ（構文木）の双方向変換に関する基礎理論・ツールに基づき変換を実現する．本研究では，ツールの機能および活発的な開発・支援状況から[4),8)] のものを採用している．この変換理論・ツールにおいては，UnCAL という専用の関数型言語にて表現されたグラフ変換に対して，上記のような双方向性が議論，保証，実装されている．

これらのグラフ変換においては，構文木上のノードに対して，特定のラベルの置換，特定のサブグラフの選択抽出（select）および削除（delete），また特定のノードの子としての新しいグラフの追加（extend）を行うことができ，第 4 章で挙げたような変換ルールを実現することができる．この際，複数の変換を順次適用することにより複雑な変換を定義することができる．本研究の変換ルールを UnCAL に変換し適用する際には，より特定の範囲に対するルール（デフォルトを上書きするルール）を先に適用するようにする．例えば第 4.1 節の例では，一部の *real* が *float* に変換された後，残ったものが *double* に変換される．

5.2 仕様とコードの対応関係の把握

基盤となっている変換理論・ツールにおいては，変換時のログとして，変換後のグラフ（構文木）内の各ノードが，変換前のどのノードに由来するか，もしくは新しく導入されたのか，を記録する．このログを利用することにより下記の情報が抽出できる．

- Java コードにおける要素がどの VDM++仕様における要素に由来するか
- Java コードのみに新しく導入された要素が，どのルールによって導入されたか

また変換を表す UnCAL を変形することにより，名前等の変更や削除を実際に行わず該当部分だけを抜き出すような変換（選択）処理を定義することもできる．これにより，下記の情報が抽出できる．

- VDM++仕様におけるどの要素が，あるルールによって変更や削除されるか

以上により VDM++仕様，変換ルール，Java コード間の対応関係を抽出できる．

5.3 Java コード側の変更の反映

Java コードにおいて変更があった場合，それは実装に固有の事項かもしれないし，VDM++仕様にも反映されるべき本質的な変更かもしれない．この点については，何らかの方法で開発者に入力を求める必要がある．前者の場合，その修正に対応する変換ルールを定義することとする．このように明示的に情報を残しておくことにより，以後 VDM++仕様から Java コードを再生成した場合にもその修正が反映される．

Java コードにおける修正が本質的な変更である場合に，その修正を VDM++仕様に反映することを考える．基本的には基盤となる変換理論・ツールが保証する双方向性によって，変更の反映は可能である．ただし以下二つの制限がある．

- 変換理論・ツールは構文木上の変更を扱うものであるため，コードから構文木を生成する際に，変更前後で同じ要素に同じ識別子を与えることが必要となる．このため実際のツールインターフェースでは，名前の変更といった専用のコマンドを用意し変更を明示させたり，開発者による編集を適宜検出したりする必要がある．
- Java コード側に新しく追加された要素の反映の方法は一意に定まらない．例えば，VDM++仕様上での *int* 型も *nat* 型も，Java コードでは *int* 型に変換されうる．このため，Java コード上で *int* 型の変数が追加された際，VDM++仕様にてどちらの型を用いるべきかは自明でない．デフォルトの逆変換方法を定める（上の例で *int* を用いると決めてしまう）ことも考えられるが，独自の変換ルールを導入した場合や独自の逆変換方法を用いたい場合に対応できない．このため現在は，本質的な要素の追加はVDM++仕様に対してまず行いそれを Java コードに反映することとしている．

6. GUI ツールのプロトタイプ実装

これまで述べたフレームワークを，基盤となる変換理論を実装したツール GRoundTram（バージョン 0.9.2）[8] を用いて実装した．GRoundTram は UnQL+クエリによる VDM++構文木から Java 構文木への変換，その際のログ出力，および Java 構文木上の変更の VDM++構文木への反映（逆変換）の機能を提供している．これに対し，VDM++仕様および Java コードとそれぞれの構文木表現との対応付け，変換ルールから UnQL+クエリの生成，第 5.2 節で述べた対応付けの抽出等の機能を加え，フレームワークの機能を実現している．

VDM++仕様，Java コード，変換ルール間の対応付けについては，GUI による支援も実装している．具体的には，**図 6** にあるような三つのエディタからなる GUI において，これら三つの記述の一つのある部分にカーソルを置くと，残り二つの記述のそれぞれにおいて，その部分に対応する部分が強調表示されるようになっている．

7. 議　　論

本論文においては，VDM++仕様から Java コードへの変換フレームワークを提案した．このフレームワークにおいては，実装方針を表す変換ルールを明示的に記述し，また双方向グラフ変換理論に基づき実現することにより，第 1 章で挙げたコード生成のカスタマイズ，共通テストケースの生成，およびトレーサビリティを実現している．

本研究のアプローチは，完全に自動化されたコード生成と，完全に手動でのコーディングとの二つの両極端の間であると言える．例えば一時的なアプリケーションを手早く立ち上げる際など，非機能的要求が少ないような場合には完全に自動化されたコード生成が役に立つ．一方，例えば資源制約が大きいデバイス向けのアプリケーションを構築する際など，特別なコーディング形式が求められる場合には開発者によるコーディングが適している．本研究のアプローチは，それ以外の多くの場合，部分的に特定の実装方針を採用する必要がある場合に適していると考えられる．例えば，一部のクラスのみ並列アクセスに対応できるようにしたり，一部の要素を既存ライブラリにて実現したりといった場合が挙げられる．

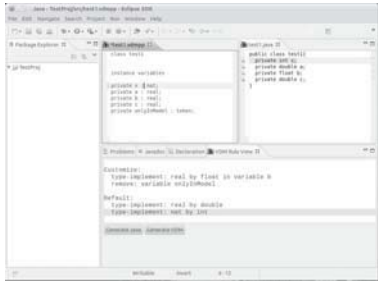

図 3　GUI ツールのスクリーンショット
Fig. 3　Screenshot of GUI-based Tool

また本研究のアプローチにおいては，VDM++仕様と Java コード間の対応関係を明示，管理し，変更を行っていくことを考えている．これは，従来の形式手法が対象としていた，高信頼性が求められる一方で仕様が明確に固められるシステムの開発[9] よりも，要求変更や派生開発の多い一般的なシステム開発へと適用を広げていくことを目指している．

以降では提案フレームワークの特徴に関し，その利点と限界について議論する．

7.1 構 文 変 換

提案アプローチにおいては構文的に変換を定義し扱っている．このアプローチでは開発者が意味論定義や証明を行う必要がなく，気軽に実装方針の記述を行うことができる．これは，厳密な定理証明等よりも，厳密な記述や，テストによる検証を主とする VDM 自体のアプローチとも合致する．なお，こういった記述や実行を主とするアプローチにおいても，自然言語の限界，つまり曖昧さや不完全さ，不整合を解消する効果は大きいとされている[10]．

一方で，一般に形式手法といった場合，B メソッド[3] のように，定理証明による厳密な検証や，正しいプログラムを導くための段階的詳細化を扱うことがある．段階的詳細化においては，モデルとその詳細化との関係を，宣言的な定理として記述し，定理証明を通して正しさを保証する．このようなアプローチは高い信頼性を保証する一方で，通常の開発プロセス

とは大きく異なる専門的なプロセスを要求し，鉄道制御[9] 等，高信頼性が求められるシステムのごく一部にのみ適用されることが多い．これに対し，本研究のアプローチは本章はじめでも議論したように，より広いシステム開発への適用を目指している．

ただし，提案フレームワークにおいても，デフォルトの変換ルールの正しさ，すなわち生成される Java コードの妥当性（例えば文法・型エラーのないこと）の保証は必須である．この点については，現在の実装で扱っている VDM++および Java の文法要素，それらに関する検証が代表的なものに限られているため，今後進めていく必要がある．また特に，開発者による変換ルールの追加・上書きを許すため，それを含めて変換ルールの正しさを分析，検証，保証していく枠組みを整備する必要がある．

7.2　VDM への特化

前節で議論したライトウェイトさに加え，本研究の変換ルールの記法は VDM における言語設計の思想に依存している．具体的には，VDM++と Java において共通した構造にて記述が行われることを想定しており，その結果，追加される実装方針に関する情報のみを変換ルールとして与えればよいとしている．例えば，状態遷移ベースのモデルや，性質ベースの宣言的なモデルから Java コードへの変換を考えた場合，大きな構造変化を伴い，変換結果を予測しつつ実装方針を指示することが難しい．一方，VDM ++と同様の変数と操作による仕様記述言語，Java と同様の手続き型プログラミング言語であれば，同じアプローチによって変換フレームワークを構築することができると考えられる．

本研究では VDM における抽象化方針のパターンに注目した変換ルール言語を定めることにより，汎用的な言語変換ツール等に比べて直感的な記法を用いることができている．一方で，より特化したドメイン依存のルールのライブラリ，例えば VDM++の写像型とデータベースによる実現との間の変換等の有効性は高いと考えられる．

7.3　変換の実現

本論文においては，ツールの開発状況を踏まえ，[4),8)] における双方向グラフ変換理論を採用した．同様の双方向性を満たす変換（[11)] 等）を用いても同様の枠組みが実現できる可能性がある．また，こういった変換理論に基づきつつ提案フレームワークに特化した変換手続きを定義することにより，変換性能を向上する等の改善も検討すべきである．

また，本論文においては，開発者が変換ルールをすべて与えるという前提での支援を議論した．実際には，意識せずに記述した Java コードから VDM++仕様との対応を推測し変換ルールを抽出するような機構の有用性が高いであろう．その際には推測の精度の問題等もあるため，実用的にはインタラクティブな，洗練されたインターフェースが必要となる．

8. おわりに

本論文では，VDM++と Java を対象として形式仕様と実装コードの差異を踏まえた変換フレームワークを提案した．このフレームワークでは，上書き可能な変換ルールを用いることにより，実装方針を反映してのコード生成，および共通テストケースの生成を行うことができるようになっている．またその実現にグラフの双方向変換理論を用い，記述要素間の対応付けや，実装コードの変更の仕様への反映が可能となっている．今後，様々な文法への対応，洗練された GUI の構築等多くの課題が残されているが，本研究を第一歩として，ライトウェイトな VDM における仕様と実装を連動させての管理が促進されると期待している．

参　考　文　献

1) Fitzgerald, J. and Larsen, P.G.: *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press (1998).
2) Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M.: *Validated Designs For Object-oriented Systems*, Springer (2005).
3) B Method - Presentation of B Method, B Language, and formal methods, `http://www.bmethod.com/`.
4) Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K. and Matsuda, K.: Bidirectionalizing Graph Transformations, *The 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)* (2010).
5) VDM information web site, `http://www.vdmtools.jp/`.
6) Lausdahl, K., Lintrup, H.K. and Larsen, P.G.: Connecting UML and VDM++ with Open Tool Support, *The 16th International Symposium on Formal Methods (FM 2009)*, pp.563–578 (2009).
7) Ishikawa, F. and Murakami, Y.: Challenges in Inheriting Test Cases Configurations from VDM to Implementation, *The 7th VDM-Overture Workshop* (2009).
8) The BiG Project, `http://www.biglab.org/`.
9) Behm, P., Benoit, P., Faivre, A. and Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project, *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, p.712 (1999).
10) Hall, A.: Seven Myths of Formal Methods, *IEEE Software*, Vol.7, No.5, pp.11–19 (1990).
11) Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C. and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol.29, No.3 (2007).