社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2011-65 (2012-3)

制約記述言語OCLとJMLのモデル駆動開発技法に基づいた双方向の変換手法の提案

榛葉　浩章† 　花田健太郎† 　岡野　浩三† 　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †h-shimba@ics.es.osaka-u.ac.jp, ††{k-hanada,okano,kusumoto}@ist.osaka-u.ac.jp

あらまし　OCL (Object Constraint Language) は UML 記述に対しさらに詳細に性質記述を行うために設計された言語である. 近年 MDA 関連技術の発展により，UML からプログラム言語への変換技術が着目を浴びており，OCL から JML (Java Modelling Language) のようなプログラムレベルの仕様記述言語への変換技術も研究されつつある. 研究グループでは，OCL から JML への変換及び，その逆変換である JML から OCL への変換について研究しており，双方向の変換を実現することで，仕様記述レベルでのラウンドトリップエンジニアリングによる開発の支援や，様々な制約記述言語間での相互変換を実現していきたいと考えている．本稿では，モデル変換的手法に基づいた OCL から JML 及び，JML から OCL への変換ツールの試作型の実装について述べる.

キーワード　OCL，JML，RTE，モデル変換

Bi-directinal Translation Method between OCL and JML based on Model Driven Development

Hiroaki SHINBA†, Kentaro HANADA†, Kozo OKANO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-Shi, Osaka, 565–0871 Japan
E-mail: †h-shimba@ics.es.osaka-u.ac.jp, ††{k-hanada,okano,kusumoto}@ist.osaka-u.ac.jp

Abstract　OCL (Object Constraint Language) is an annotation language for UML, which can describe specication more precisely.In recent years, MDA based techniques have emerged, thus translation techniques such as translation from OCL to JML (Java Modelling Language) as well as UML to some program languages, have gained a lot of attention.Our research group has been studying not only a translation method from OCL to JML but also a translation method from JML to OCL.By implementing bi-directional translation between OCL and JML, we can support the following two items.One is development by round trip engineering at the design level.The other one is translations among various formal specification languages.In this report, we present two implementations based on model translation techniques.Translations from OCL to JML and from JML to OCL.

Key words　OCL，JML，RTE，Model Translation

1. はじめに

近年 MDA (Model Driven Architecture) [1] 関連技術の発展により，UML (Unified Modelling Language) からプログラム言語への変換技術が注目を浴びている．UML クラス記述から Java スケルトンコードを自動生成する方法についてはすでに既存研究で多くの方法が提案されており [2], [3], 自動変換ツールも EMF フレームワークを用いた Eclipse プラグインなどの形で公開されている．OCL (Object Constraint Language) [4] から JML (Java Modeling Language) [5] への変換技術も研究が行われている．OCL は UML 記述に対し，さらに詳細に性質記述を行うために設計された言語で，OMG (Object Management Group) によって標準化されている．より実装に近い面での制約記述言語として，Java プログラムに対して JML が提案されている．JML，OCL ともに DbC (Design by Contract) [6] の概念に基づきクラスやメソッドの仕様を与えることができる.

過去に著者の所属する研究グループは，OCL 記述が付加されたクラス図に対して，JML 記述への変換法を具体的に提示

し[7]，文献[8]において実装を示している．しかし，既存ツールはMDAの主流であるモデル変換的な実装ではなく，抽象構文木を介したマッピングで実装されている．現在ではOCLからJML以外の複数の言語への変換の実現を目標としているが，既存手法ではそれぞれの変換を独立に実装する形になるため実装の再利用性が低く，複数言語への変換の実装には向かない．また，既存の実装は変換の達成を主な目的としており，ツールそのもののユーザビリティは低い．

本稿では，Xtext[9]を用いてOCLが付加されたUMLクラス図の構文定義を行い，その構文定義からJMLが付加されたJavaのスケルトンコードへの変換を定義する方法での実装を試みた．Xtextでは構文定義と変換部分が独立しており，構文定義の再利用性が高いことが利点として挙げられる．また，構文定義に対応した高機能なエディタを生成することができるため，ツールそのもののユーザビリティの向上も見込めると考えられる．

また，仕様記述レベルでのラウンドトリップエンジニアリング(以下，RTE)[10],[11]による開発の支援，様々な制約記述言語間での相互変換を目的として，OCLからJMLへの変換と同様の方法で，JMLからOCLへの変換を試みた．RTEとは，フォワードエンジニアリングと，リバースエンジニアリングを反復しながら，モデルとソースコードを徐々に詳細化していくことで開発を行う手法であり，OCLとUML，JMLとJavaのスケルトンコードを徐々に詳細化していく形で仕様記述レベルでの開発を支援していきたいと考えている．

以降，2章で背景について述べ，3章で実装について述べ，4章で評価について述べ，5章でまとめる．

2. 準　　備

本章では，背景知識や関連研究などについて簡単に触れる．

2.1 OCL

OCL (Object Constraint Language) はUMLモデルに対し，さらに詳細に性質記述を行うために設計された言語であり，UMLと同様にOMGによって標準化されている．UMLでは，実装時にモデルがどのように開発されるべきか，といった詳細な情報を表すことができない．このような問題を解決するため，OCLが導入された．

2.2 JML

JML (Java Modeling Language) は，Javaのメソッドやオブジェクトに対して制約を記述する言語である．記述においてはJavaの文法を踏襲し，初心者でも記述しやすい特徴を持つ．また，記述はJavaコメント中に記述できるため，プログラムの実装，コンパイルや実行に影響がない．

JMLには，コード実行時にJML記述に違反しないかをチェックするランタイムアサーションチェッカや，JUnit用のテストケースのスケルトンやテストメソッドを自動で出力するJMLUnit[12]，JML記述に対するJavaプログラムの実装の正しさをメソッド単位で静的検査できるESC/Java2など，コードの検証を効率化するための様々なツールがサポートされている．

2.3 モデル変換

代表的なモデル変換としてQVT[13]やATL[14]などが挙げられる．これらは，MDAにおけるモデル変換である．

モデル変換にはモデルからモデルへの変換であるModel2Model変換 (M2M)，モデルからコードへの変換であるModel2Text(M2T)が存在する．M2T変換機能を提供するツールとして，UML2Java[15]などが挙げられる．

2.4 RTE

RTE (Round Trip Engineering) とは，フォワードエンジニアリングとリバースエンジニアリングを反復しながら，モデルとソースコードを徐々に詳細化していくことで開発を行う手法であり，機能の変更や要求の変更に強いという特徴を持つ．[10],[11] IDE等のRTE開発支援ツールを用い，設計・コードのいずれかの変更の一部を他方に自動的に反映していくものが一般的である．RTEを行う際には，モデルとソースコードの整合性を保つ必要性があるが，自動的に整合性を保つ様々なツールが存在する．

2.5 Xtext

Xtext[9]とは，モデルの構文定義や，構文定義に従ったモデルからテキストへの変換ルールの構築などをサポートするためのフレームワークである．モデルの構文定義を行うことで，コード補完機能やエラー検出機能などを備えたエディタを生成することができる．生成されたエディタ上で入力となるテキスト型モデルを記述すると，自動的に変換ルールを用いてモデルをテキストにM2T変換する．

2.6 関 連 研 究

研究グループでは，OCLからJMLへの変換に関する多くの既存研究[16][17]において非対応だったiterate演算に対応した変換手法の提案・実装を行っている[8]．例えば，式(1)のようなiterateの演算が与えられた場合，この演算に対応したJavaのメソッドを生成し，JML側から呼び出すことで変換を実現している．

$$\mathrm{Set}\{1,2,3\} \text{-> iterate}(i : \mathrm{Integer};\ sum : \mathrm{Interger} = 0 \mid sum + i) \qquad (1)$$

具体的には，式(1)に対応するメソッドは図1となる．

```
private int mPrivateUseForJML01(){
    int sum = 0;
    for (int i : set){
        sum = sum + i;
    }
    return sum;
}
```

図1　iterate演算を変換したメソッドの例

しかし，既存ツールはMDAの主流であるモデル変換的な実装ではなく，抽象構文木を介したマッピングで実装されている．今後OCLからJML以外の複数の言語への変換の実現を目標としているが，既存手法ではそれぞれの変換を独立に実装する形になるため実装の再利用性が低く，複数言語への変換の実装には向かない．また，既存の実装は変換の達成を主な目的としており，ツールそのもののユーザビリティは低い．

また，要求変更に対してRTEによる開発を実施することで，

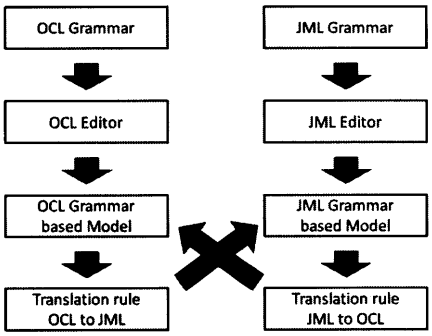

図 2　Xtext による実装概要

開発期間を短縮し，効率化することができる．Web アプリケーションを対象に RTE を支援するための研究などが存在している [18].

本稿では，Xtext を用いて OCL が付加された UML クラス図の構文定義を行い，その構文定義から JML が付加された Java のスケルトンコードへの変換を定義する方法で実装することで，研究グループの既存ツールの課題点を改善することを試みた．

3. 実　　装

この章では，実装に関して述べる．

3.1 実装方針

Xtext を用いて，モデルの構文定義を行い，モデルの構文定義からコードへの変換ルールを定義することでそれぞれの変換を実現する．図 2 に実装の概要を示す．

この実装方法には以下のような利点が挙げられる．

- モデルの構文定義は，テキストへの変換ルールの実装から独立しているため構文定義そのものの再利用性が高い．
- Xtext ではモデルの構文定義を行うことで，コード補完や文法エラーの検出を行えるユーザビリティの高いエディタが得られる．

3.2 OCL から JML への変換

ここでは，OCL から JML への変換の実装について述べる．

3.2.1 OCL が付加された UML の構文定義

OCL を付加できる UML クラス図のモデルの構文定義を行った．UML の部分に関しては，Xtext で既存のものが存在するので流用し，OCL 部分を追加する形で構文定義を行った．OCL 部分に関しては，メソッド名や引数の数，戻り値の型などの場合分けを特に考慮した．変換ルールの作成は構文定義に依存するので，構文定義の段階で詳細な場合分けを行うことで意味的な解析にかかるコストが削減され，構文定義を再利用することの有用性が高まると考えられる．また，生成されるエディタのエラー検出やコンテンツアシストは構文定義に依存するので，構文定義の段階で場合分けを厳密に定義するほどコンテンツアシスト機能などが充実したエディタが得られる．これらのことから，メソッド名なども考慮した構文定義を行うことは，ユーザビリティの観点や実装の再利用性の観点から有用であると考えられる．

3.2.2 OCL から JML への変換ルールの定義

表 1 に変換ルールの一部を記載する．OCL 式から JML 式への変換関数の表記を μ で与え，a_i は Integer, Real, Boolean の任意の型，c_i は Collection 型の任意の型を表している．

表 1　OCL から JML への μ 変換表の一部

$\mu(a_1 = a_2)$	=	$\mu(a_1) == \mu(a_2)$
$\mu(a_1 <= a_2)$	=	$\mu(a_1) <= \mu(a_2)$
$\mu(a_1 <> a_2)$	=	$\mu(a_1)! = \mu(a_2)$
$\mu(c_1 \text{->size())}$	=	$\mu(c_1)$.size()
$\mu(c_1 \text{->includes}(a_1))$	=	$\mu(c_1)$.has($\mu(a_1)$)
$\mu(c_1 \text{->isEmpty())}$	=	$\mu(c_1)$.isEmpty()

OCL-JML 間の変換は既存研究とほぼ同様に定義している [8]. exists 演算や forall 演算のようなコレクションループは表 2 のように iterate 演算に置き換えることが可能なため，既存研究では iterate 演算以外のコレクションループも iterate 演算に置き換え，その OCL 式と意味的に等価なメソッドを生成して，その値を JML 側から参照する方法で変換していた．しかし，この変換方法ではコードの可読性が低下するといった問題点が挙げられていたため，本稿の実装では JML にメソッドが存在している exists 演算や forall 演算などは直接的にマッピングを行っている．

表 2　Collection-Iterate 対応表の一部

$c_1 \text{->exists}(a_1 \mid a_2)$	=	c_1->iterate(a_1; res : Boolean = false | res or a_2)
$c_1 \text{->forAll}(a_1 \mid a_2)$	=	c_1->iterate(a_1; res : Boolean = true | res and a_2)
$st_1 \text{->select}(a_1 \mid a_2))$	=	st_1->iterate(a_1; res : Set(T) = Set {} | if a_2 then res ->includeing (a_1) else res endif)
$st_1 \text{->reject}(a_1 \mid a_2))$	=	st_1->select(a_1 | not a_2)

また，DSL に従ったモデルの例を図 3 に，モデルからの変換結果を図 4 に例として示す．

```
entity Sample {
    inv : sampleVariable >= 0
    sampleVariable : Integer
}
```

図 3　DSL に従ったモデル

```
package ;
public class Sample {
/*@
invariant ((sampleVariable)>=0);
@*/
    private Integer sampleVariable;

    public Integer getSampleVariable() {
        return sampleVariable;
    }

    public void setSampleVariable(Integer sampleVariable) {
        this.sampleVariable = sampleVariable;
    }
}
```

図 4　モデルから変換された出力結果

3.2.3 Oclvoid 型の扱いについて

OclVoid 型は未定義値である Undefined 定数のみをもつクラスである．Undefined はオブジェクトがサポートしていない型にキャストしたり，空のコレクションから要素を取得しようとすると戻り値として得られる値である．これは JML では null として扱われるので，ほとんどの変換においては null として扱っている．

null との違う点として，例えば True or Undefined などの一部の論理演算においては，OclVoid は式そのものを未定義としては扱わず，正当な評価（上記の場合は True）を返すというものがある．これに対応するためには，Undefined をこの場合は単純に null として扱うのではなく，以下のように扱えば対応できる．

```
(a_1 == null ? false :
        throw new JMLTranslationException())
```

左辺の boolean の評価値により，or 演算の評価値に影響を与えないため，右辺が null であった場合は false を返す．null でない場合は or 演算の型不整合であるので，検証時に例外を出力するようにしている．

3.3 JML から OCL への変換

ここでは JML から OCL への変換の実装について述べる．

3.3.1 JML が付加された Java スケルトンコードの構文定義

JML が付加された Java のスケルトンコードが記述できるモデルの構文定義を行った．Java については，クラス宣言, 修飾子, クラスのフィールド変数とメソッドの宣言を対象として構文を定義した．これは正しく変換を行うために，変数の型の情報やそれぞれの名前が必要となるためである．

JML の部分は今回は試作型として JML の Reference Manual で定義されている式の部分の構文と演算のみを扱った．JML から OCL への変換を実現する際に JML は OCL と比べて実装に近い言語なので具体性が高く, OCL では表現できないものが存在する．例として，JML では代入演算やシフト演算が存在するが，OCL ではそれに対応する演算が存在しない．このような変換を定義できない構文や演算は，構文定義の時点で予め除いている．Xtext で生成されるエディタは構文定義に従ったエディタを生成することから，対応していない構文を除去しておくことで，変換可能なコードのみが入力できるようになり，ユーザは容易に対応している JML 式が理解できる．

3.3.2 JML から OCL への変換ルールの定義

JML から OCL への変換ルールの定義の一部を表 3 に示す．

JML から OCL への変換は，基本演算についてはほぼ 1 対 1 で OCL・JML 間の対応を取ることができ，JML の演算子を OCL の演算子へと変換するだけでよい. ただし，一部の演算子については，被演算子の順序を入れ替えるなどする必要があるものも存在する．

また，JML の文法は，Java の文法を踏襲している．例えば “+演算子”については，数値型 + 数値型 など型が同じもの同士の演算以外にも，文字列型 + 数値型 という形も記述可能である. しかし，OCL では異なる型同士での演算には対応していないため，対応することができない．また，JML では 文字列型+文字列型 のように数値型以外も “+演算子”を使う事ができる．

表 3 μ 変換ルール

$\mu(b_1?b_2:b_3)$	=	if $\mu(b_1)$ then $\mu(b_2)$ else $\mu(b_3)$ endif
$\mu(b_1<==>b_2)$	=	$\mu(b_1)= \mu(b_2)$
$\mu(b_1<=!=>b_2)$	=	$\mu(b_1) <> \mu(b_2)$
$\mu(b_1==>b_2)$	=	$\mu(b_1)$ implies $\mu(b_2)$
$\mu(b_1<==b_2)$	=	$\mu(b_2)$ implies $\mu(b_1)$
$\mu(b_1\&\&b_2)$	=	$\mu(b_1)$ and $\mu(b_2)$
$\mu(b_1\|\|b_2)$	=	$\mu(b_1)$ or $\mu(b_2)$
$\mu(b_1\|b_2)$	=	$\mu(b_1)$ or $\mu(b_2)$
$\mu(b_1 \hat{} b_2)$	=	$\mu(b_1$ xor μ(b)
$\mu(b_1\& b_2)$	=	$\mu(b_1)$ and $\mu(b_2)$
μ(\result)	=	result
$\mu(\backslash old(a_1))$	=	$\mu(a_1)$@pre
$\mu(\backslash not\_modified(a_1))$	=	$\mu(a_1) = \mu(a_1)$@pre
$\mu(\backslash fresh(a_1))$	=	$\mu(a_1)$.oclIsNew()

しかし，OCL では文字列型と文字列型なら 文字列型.*concat*(文字列型) のように表現しなければならないため，型情報を考慮して変換をしなければならない．

ループ演算に関しては，OCL の exist や forall 演算は Collection 型の演算として存在しているが，JML の exist や forall は OCL と異なり，Java における for 文のような記述を行うこともできる．そのため，JML における全てのループ演算を OCL で対応することはできない．そこで今回は，Collection に対して演算している場合のみ JML から OCL へ変換し，それ以外の場合はエラーを出力する形で対応した．

3.4 型 推 論

二つのオブジェクトが同値であることを評価する演算に関して，OCL では任意の型において ‘=’ が用いられるが，JML では基本データ型の比較には ‘==’ が用いられ，参照型には *equals*() メソッドが用いられる．変換を正しく実行するためには，変数の型などを正しく判別する必要がある．

ここでは，変換を行う前にまずクラス名・メソッド名・変数名などをキーとして型の情報を返す Map を作成し，構文内で変数などが現れた際に，作成した Map から型を取得することで対応している．本来はエディタ上のモデルチェックの段階で判別すべきと考えられるが，現在は変換を行うまでは型の情報についての正しい判別行うことができない状態である．

4. 評 価 実 験

この章では，評価実験について述べる．

4.1 実 験 概 要

本研究では作成したツールを小規模なプロジェクトに対して適用した．実験 1 は実験対象プログラムに記述された JML を作成したツールで変換して, どの程度意味的に正しく変換が行えるかどうかを検証する．実験 2 では RTE 適用可能性を確かめるために作成したツールで JML から OCL へ変換した結果を, 既存の変換ツールで再び JML へと変換し, どの程度元の表現を再現できるか検証する．

4.2 計 測 内 容

作成したツールで変換した結果を評価する指標として以下の

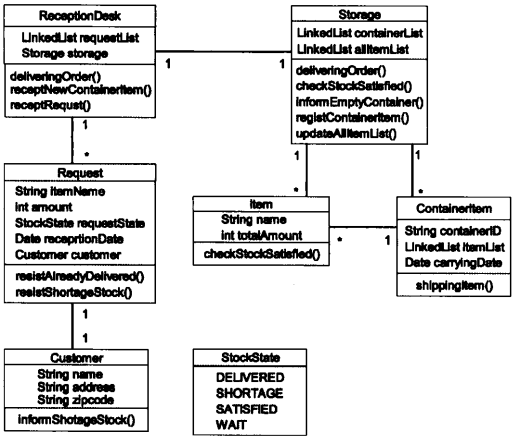

図 5　在庫管理システムの UML クラス図

2 点を計測する.

変換率 JML から OCL へ，意味的に正しく変換できた数.

逆変換率 JML から OCL へ変換し，再び JML に変換したときに意味的に同じ表現に戻った数.

変換率については，入力として与えた JML のうち，OCL へ意味的に正しく変換できた割合とする．逆変換率については，入力として与えた JML と 2 段階の変換を経て再び JML に変換された結果を比較して，意味的に同じ表現に戻る割合とする.

4.3 実験結果

4.3.1 実　験　1

実験 1 では，図 5 で示す在庫管理システムを用いて評価を行った.

図 5 は，我々の研究グループの過去の研究[19] で作成した UML であり，既にすべてのメソッドに JML が記述されている．在庫管理プログラムに記述されている事前条件, 事後条件, 不変条件の数は全部で 130 で, それらについて変換を適用した．意味的に正しく変換できた条件数は 102 で変換率は 78.4%となった．図 6，図 7 に変換できなかった場合の一部を具体例として示す.

```
/*@
ensures \result.matches("containerID." + containerID
          + "CarryingDate | " + carryingDate + "\n{1}")
@*/
String toString(){
}
/*@
ensures (\forall Request r; requestList.contains(r);
         r.getAmount() > 0);
ensures (\forall Request r; requestList.contains(r)
         && r.getAmount() != \old(r.getAmount());
         r.getRequestState() == StockState.SHORTAGE);
@*/
List deliveringOrder(){
}
```

図 6　JML から OCL へ変換できなかった例 (入力)

正しく変換できなかった場合は forall などのループ演算で変数が複数宣言されていたり条件が複雑な場合,\type 演算,\typeof

```
context ContainerItem::toString()::String
post : result.matches('ContainerID.'
       [type error][type error][type error][type error])

context ReceptionDesk::deliveringOrder()::List
post : requestList->forAll(r:Request|r.getAmount() > 0)
post : requestList and r=(r)@pre and ->forAll(
       r:getRequestState() = StockState.SHORTAGE)
```

図 7　JML から OCL へ変換できなかった例 (出力結果)

演算,String 型と数値型の+演算などがあげられる.

4.3.2 実　験　2

実験 1 において，JML から OCL への変換で意味的に正しく変換できた式の数は 103 個である．それらの 103 個の式を，試作型の OCL から JML への変換ツールに入力として与えたところ，全て変換対象として正しく認識されることが確認された．JML から OCL への変換で正しく変換できなかったものを含めると 76%となるが，正しく OCL に変換できた式に関して言えば，OCL から JML への変換可能率は 100%となった．このことから JML から OCL への変換に関しては，問題なく変換されていることがわかる．ただし，現在ツールの変換ルールの部分が実装途中の段階であることから，OCL から JML への変換結果にバグがある部分もいくつか見つかった．結果としては 103 の入力した OCL 式のうち，87 の式は正しく変換され，変換率は 84.5%となり，16 の式は変換結果にバグが見つかった．図 8, 図 9 に変換できなかった場合の一部を具体例として示す.

```
pre : o.oclIsTypeOf(Request)
post : result = (receptionDate.getTime()-
       (o.oclAsType(Request)).getReceptionDate())
       .oclAsType(Integer) or result = 0
op compareTo(o : Object)
```

図 8　OCL から JML へ変換できなかった例 (入力)

```
/*@
requires o.getClass().equals(Request);
ensures (\result == (receiptionDate.getTime()-
         ((o.oclAsType(Request)).getReceiptionDate()))
         .oclAsType(Integer)) || (\result == 0);
@*/
public void CompareTo(Object o){
}
```

図 9　OCL から JML へ変換できなかった例 (出力結果)

oclAsType メソッドは構文定義には記述されていたが，変換ルールには実装されていないという実装漏れが見つかった．うまく変換されなかった 16 の式は全て修正方法は考察できているので，今後修正していきたいと考えている.

4.4 考　　察

実験 1 については，実験結果の変換率を見てみると変換率は 78.4%となっている．今回は試作型として主に基本演算の変換を実現したので全てを変換することはできなかった．しかし 78.4%という変換率は JML 記述の多くを基本演算が占めている

ことを表していて, 変換の有効性を確かめることができた.

変換できなかったものの一部についての考察を行なっていく. まず JML の \type と \typeof といった引数として与えた変数の型名やリストなどの要素の型名を返す演算については変換出来なかった. これは OCL では型名を求める演算は存在していないためである. この変換に関しては, JML の解析の時点で引数の型の情報を求めておき, OCL には直に型名を出力するという対応が考えられる.

実験 2 については, 実験結果の逆変換率は JML から OCL で意味的に正しく変換できたものについては 84.4%となった. 変換結果には正しくない表現になっているものが存在したが, これは逆変換に使用したツールが試作段階で, デバッグが完了していなかったり実装の漏れが存在するためである. 入力として与えた OCL は変換可能と認識されたので品質に問題はないが, 変換部分が不完全なのでこのような結果になったと考えられる. これらより JML から OCL への変換は実際に他のツールなどに適用できる形に変換できていて, 生成された OCL の品質に問題がないことが確認できた. 変換できなかったものについては, 変換は 1 対 1 で対応しているので実装するときのミスであり, 今後追加実装を行うことで正しい変換を実現できると考えている.

5. あとがき

本稿では仕様記述レベルでの RTE による開発を支援すること, 様々な制約記述言語間での相互変換の 2 点を目標とした, OCL から JML への変換及び, その逆変換である JML から OCL への変換についての具体的な実装方法, 及び試作型として実装したツールについての簡単な評価を行った.

今後の課題としては, まずツールを完成させることである. 現在はまだ試作段階であり, 対応し切れていない部分も存在しているので, それらの点を実装に移していく. 例えば OCL から JML に関して言えば, Undefined の扱いなどが現状不十分であるので, 今後正確に対応していくことなども課題として考えている.

ツールの実装が完成後には, 追加実験を行うことを考えている. ツールの評価に関しては, まず OCL から JML への変換と JML から OCL への変換のそれぞれについて評価を更に詳細に行おうと考えている.

OCL から JML への変換ツールに関しては, 現在は変換成功数などについての簡単な評価のみ行った状態である. 今後は esc/java2 や jml4c などの JML 用の検査ツールに対して生成された JML を適用した場合の結果と, 人手によって記述された JML を適用した場合の結果を比較・検証することを考えている.

JML から OCL への変換ツールに関しては, Octopus のような OCL 式検証ツールを利用して, 生成された OCL を検証, 人手によって記述された OCL 式と比較したコードの可読性の評価などを考えている.

また, OCL-JML 間の相互変換をツールを利用して繰り返し適用し続けることができるか, 相互変換を繰り返しても意味的に正しさを保てるか, などの評価を作成した OCL から JML 及び JML から OCL への変換ツールを用いて行うことを考えている.

謝辞 本研究の一部は科学研究費補助金基盤 C (21500036) と文部科学省「次世代 IT 基盤構築のための研究開発」(研究開発領域名：ソフトウェア構築状況の可視化技術の普及) の助成による. また, Xtext について教えていただいた公立はこだて未来大学の田中明氏に感謝する.

文　献

[1] A.G. Kleppe, J. Warmer, and W. Bast, MDA explained: the model driven architecture: practice and promise, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[2] G. Engels, R. Hücking, S. Sauer, and A. Wagner, "Uml collaboration diagrams and their transformation to java," UML1999 -Beyond the Standard, Second International Conference, pp.473–488, 1999.

[3] W. Harrison, C. Barton, and M. Raghavachari, "Mapping uml designs to java," Proc. of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp.178–187, 2000.

[4] Object Management Group, "Ocl 2.0 specification," 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[5] G. Leavens, A. Baker, and C. Ruby, "Jml: A notation for detailed design," Behavioral Specifications of Businesses and Systems, pp.175–188, 1999.

[6] B. Meyer, Eiffel: the language, Prentice-Hall, Inc., Upper Saddle River, NJ, 1992.

[7] 尾鷲方志, 岡野浩三, 楠本真二, "メソッドの自動生成を用いた ocl の jml への変換ツールの設計," ソフトウェア工学の基礎 XVI, 日本ソフトウェア科学会 (FOSE 2009), pp.191–198, 近代科学社, 2009.

[8] 宮澤清介, 花田健太郎, 岡野浩三, 楠本真二, "Ocl から jml への変換ツールにおける対応クラスの拡張と教務システムに対する適用実験," 信学技報, vol.110, no.458, pp.115–120, 2011.

[9] Eclipse Foundation, "Xtext - Language Development Framework". http://www.eclipse.org/Xtext/.

[10] N. Medvidovic, A. Egyed, and D.S. Rosenblum, "Round-trip software engineering using uml: From architecture to design and back," 1999.

[11] S. Sendall and J. Kuster, "Taming model round-trip engineering," In Proceedings of Workshop 'Best Practices for Model-Driven Software Development, pp.1–13, 2004.

[12] Y. Cheon and G.1 Leavens, "A simple and practical approach to unit testing: The jml and junit way," ECOOP 2002 Object-Oriented Programming, pp.1789–1901, 2006.

[13] Object Management Group, "Documents associated with meta object facility (mof) 2.0 query/view/transformation, v1.1," 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

[14] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," Science of Computer Programming, vol.72, no.1-2, pp.31–39, 2008.

[15] "Uml2java". "http://uml2java.sourceforge.net/".

[16] A. Hamie, "Translating the object constraint language into the java modelling language," In Proc. of the 2004 ACM symposium on Applied computing, pp.1531–1535, 2004.

[17] M. Rodion and R. Alessandra, "Implementing an ocl to jml translation tool," 信学技報, vol.106, no.426, pp.13–17, 2006.

[18] 林千博, 名倉正剛, 高田眞吾, "ajax アプリケーションを対象としたラウンドトリップエンジニアリング支援手法," ソフトウェア工学の基礎 XVI, 日本ソフトウェア科学会 (FOSE2011), pp.51–60, 近代科学社, 2011.

[19] 尾鷲方志, 岡野浩三, 楠本真二, "JML を用いた在庫管理プログラムの設計と ESC/Java2 を用いた検証," 電子情報通信学会技術報告, vol.107, no.176, pp.37–42, 2007.