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

信学技報
TECHNICAL REPORT OF IEICE.

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

宮澤　清介†　　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: †{k-miyazw,okano,kusumoto}@ist.osaka-u.ac.jp, ††k-hanada@ics.es.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，従来研究で未対応であった iterate 演算を，生成される Java スケルトンに対応するメソッドを記述するという手法を用いてツールに実装している．本稿では，ツールの Eclipse プラグイン化ならびに対応クラスを拡張し，プロジェクト外のライブラリを含んだ制約を記述可能にした．ツールをより大規模な実プロジェクトに対して適用したところ，メソッドに記述されるべき JML のうち約 90%を変換により出力できた．
キーワード　モデル変換，OCL，JML, UML, Java

Class Enhancement of our OCL to JML translation tool and Its Application to a Curriculum Management System

Kiyoyuki MIYAZAWA†, 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: †{k-miyazw,okano,kusumoto}@ist.osaka-u.ac.jp, ††k-hanada@ics.es.osaka-u.ac.jp

Abstract　OCL (Object Constraint Language) is an annotation language for UML, which can describe specification more precisely. In recent years, MDA 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. Past researches on translation from OCL to JML often pays little attention to collection features, especially iteration. Our research group has proposed a method to overcome such the problem by using Java method templates. In this report, we present a tool enhancement such as plug-in onto Eclipse and adaptation to external libraries. We have also performed an experiment where the tools is applied to real-examples. We found that 90% of necessary constraints are well translated.
Key words　model translation, OCL, JML, UML, Java

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 [4]．それに伴い OCL(Object Constraint Language) [5] から JML(Java Modeling Language) [6] 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) [7] 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) [8] 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．

OCL から JML への変換については Hamie が文献 [9] において構文変換技法に基づいた OCL から JML への変換法を提案しており，Rodion と Alessandra らが文献 [10] において，Hamie の研究の拡張とツールの実装を示している．また，Avila らが文献 [11] にて型の扱いなどについて改善を示しているものの，いずれの方法も Collection の対応が不十分であり，iterate 演算の対応が一部の演算に対応しているのみである．しかし，iterate 演算は広く用いられる演算であるため，対応すべき問題だと考えられる．

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た．また，各 iterate 演算に対し，対応する Java メソッドを自動生成し，Java コードに挿入する手法を提案した [12]．そして，文献 [13] において実装を示し，文献 [14] において簡単な評価を示した．

文献 [14] の評価実験は，7 クラス程度の小規模なプロジェクトを用いて簡易的に行ったため，正確な実用性までは示していない．そこで本稿では，200 クラス程度の規模のプロジェクトに対し評価実験を行い，ツールの実用性を確認した．具体的には，システムに付加すべき制約のうち，約 90%を変換により出力でき，ツールは約 600 の制約を約 7 秒で JML に変換した．

また，ツールを Eclipse プラグインとして実装し，対応クラスを拡張することで，Java 標準ライブラリなどのプロジェクト外のクラスを用いた制約も変換可能にした．

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

本章では研究の背景となる諸技術と関連研究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 [15]，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 関 連 研 究

UML から JML への変換については，Engels らの文献 [2] や Harrison らの文献 [3] 等において言及されているが，変換する上で，UML 上での仕様の厳密な定義を行う OCL に関する言及が不十分である．Hamie は文献 [9] において構文変換技法に基づいた OCL から JML への変換法を提案している．Rodion と Alessandra らは文献 [9] を基に，文献 [10] において未対応であった Tuple 型や Collection 型の演算の一部に関する変換法を提案し，ツールの実装を示している．Avila らは文献 [11] において，文献 [9] においてマッピングされた OCL と JML の Collection 型の差異を吸収し，より完全な変換を行うライブラリを提案し，変換後の可読性について言及している．しかしながら，いずれの方法も Collection ループ演算の中で最も基本的な演算である iterate 演算への対応が不十分である．iterate 演算は，引数で与えられた式を Collection のすべての要素に対して繰り返し実行するという演算である．iterate 演算の具体例として，式（1）のような演算が挙げられる．

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

これは Set に含まれるすべての要素を加算した値を返す演算を定義している．ここで，第一引数（$i : Integer$）はイテレータ変数の定義，第二引数（$sum : Integer = 0$）は戻り値として使用する変数の定義と初期化，第三引数（$sum + i$）はループ内で繰り返し実行される式を表す．

JML や Java において $sum + i$ といった式の動的な評価機構が用意されておらず直接対応することができないという問題点がある．例えば，式（1）に対し，汎用のクラス JMLTools を定義し，対応する Java メソッド *iterate*() を用意した場合，式（2）のように変換されることが想定できる．

$$\mathrm{JMLTools.iterate}(\mathrm{int}\ i, \mathrm{int}\ sum=0, sum+i, set) \quad (2)$$

このとき，$sum + i$ はメソッド呼び出し時の一度しか評価されず，以降、ループのたびに動的に繰り返し評価されない．

文献 [16] では個々の iterate 演算に対応する Java メソッドを用意することでこの問題を解決することを提案した．具体的には，式（1）に対応するメソッドは，図 1 となる．

iterate 演算はデータベースをモデル化する際など，広く用いられる演算であるため，文献 [16] はこの演算の変換に対応するアルゴリズムを示したという点で有用である．しかし，文

```
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

献 [16] では具体的な実装方法までは示しておらず，生成される JML が実用的なものであるかといった評価がなされていない．

本稿では [16] の対応クラスを拡張し，Eclipse プラグインとしての実装を行った．そして，実プロジェクトに対して評価実験を行った．

3. 実　　装

ここでは実装に関して詳細に述べる．図 2 にツールの概要を示す．このツールは Eclipse のプラグインとして開発した．

本ツールは OCL の構文解析時に，詳細な型情報を付加することにより，JML への変換を容易にした．ここでは OCL 構文解析器の実装と，それに型情報を付加することについて詳細に述べる．また，OCL から JML への変換規則の一部を掲載する．最後に，iterate の入れ子などに対応するために，どのように iterate 情報を扱う JML の構文木を拡張したかについて述べる．

3.1 外 部 仕 様

ツールの外部仕様を，図 3 に示す．

まずユーザは，右上のコンポーネントを用いて UML を描画する．OCL はテキストエディタを用いて記述する．そして，左下のコンポーネントを用いて UML，OCL，JML 挿入先 Java プロジェクトのロケーションを指定し，実行ボタンをクリックすることで，OCL を JML に変換することができる．ロケーション指定はファイルシステムからの選択と，ドラッグアンドドロップの両方に対応し，ユーザの利便性に配慮した．

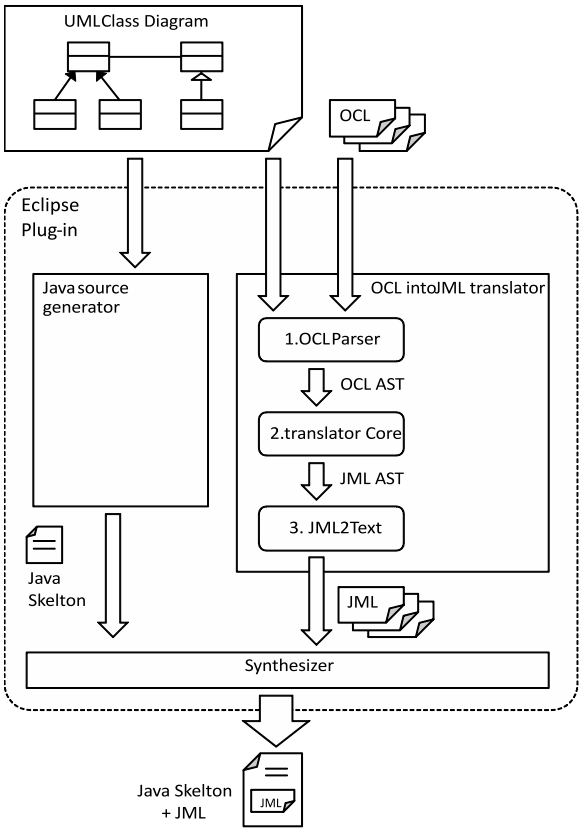

図 2　ツール全体図

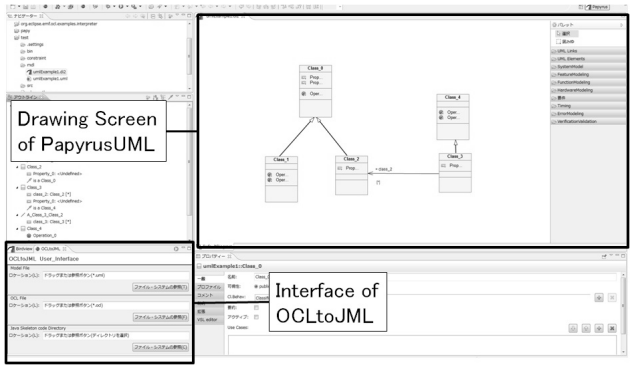

図 3　ツールの概観

3.2 UML から Java への変換

UML クラス図から Java ソースコードへの変換に関しては既存研究で多くの方法が提案されており，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．よって本研究では既存のリソースを活用した．

本研究では，UML の描画に Papyrus UML [17] を，UML から Java ソースコードへの変換に Acceleo を用いた．Papyrus UML は Eclipse Foundation が進める Model Development Tools(MDT) [18] プロジェクトのうちのひとつで開発されている無償の UML CASE ツールである．Eclipse プラグインとして利用することができ，属性に対する型の割り当てや関連に関する設定などを詳細に記述することができる．また，MDT プロジェクトは標準規格を推進するためのプロジェクトであるため，Papyrus UML の入出力で用いる UML ファイルは標準規格に従っている．以上の理由により，Papyrus UML が本研究で利用するのに一番適していた．

しかし，Papyrus UML には Java コードの生成機能が存在しなかったため，コード生成には Acceleo を用いた．Acceleo も Eclipse プラグインとして提供されているため，ユーザは Eclipse 上で UML の描画とコード生成を行うことができる．

3.3 対応クラスの拡張

OCL Parser は，OCL で標準に定義されている型と，ユーザが UML クラス図に定義した型の 2 種類のみを用いて，構文解析と意味解析を行うように設計している．しかし，実用規模のプロジェクトにおいて，Java 標準ライブラリやサードパーティが開発した外部パッケージを用いずに設計を行うことは困難であり，また事前条件や事後条件を記述することも同様である．

本稿では利用するライブラリのクラスを，ユーザが定義しているプロジェクトの UML に追加することで，対応クラスを拡張した．具体的には，UMLFromJava という，Java プロジェクトまたはパッケージフォルダを入力とし，そのディレクトリ以下に含まれる Java ファイルを UML クラス図に変換する Eclipse プラグインを，Java ファイル単位で選択できるように拡張した．また，OCL の意味解析時にクラスの継承関係を用いるため，入力した Java ファイルの継承関係を取得し，継承関係にあるクラスも入力に加えるようにユーザに促すような機能を追加した．

これらの対応クラスの拡張により，ツールを実プロジェクト

に適用したときに，メソッドに記述されるべき JML を数多く出力することができた．

4. 評価実験

ここでは，評価実験について詳細に述べる．文献 [13] では，7 クラス程度の小規模なプロジェクトに対して簡単な評価実験を行ったものの，対象プロジェクトの小ささから，正確な実用性までは判断できなかった．

本稿では，約 200 クラス程度の規模のプロジェクトのうち，システムのコア部分の約 60 クラスを抽出し，実験を行った．

4.1 計測内容

ツールで生成した JML の品質を評価する指標として，以下の三点を計測する．

網羅率　理想的な JML 記述が定義した制約を，どの程度網羅できるか

再現率　理想的な JML 記述にどの程度一致するか

変換時間　すべての OCL を JML に変換し終わるまでの時間

理想的な JML 記述の定義は，4.2.1 節で後述する．

4.2 方法

4.2.1 テストデータの準備

実験を行うためには，OCL 記述と理想的な JML 記述が必要である．ここで理想的な JML 記述を，“プログラマがソースコードや Javadoc コメントなどを参考に記述した JML ”と定義する．OCL は，理想的な JML を参考に記述する．

したがって，テストデータの準備手順としては，まず JML を作成した後に OCL を記述する．

4.2.2 網羅率の計測

本稿では網羅率を式 3 と定義する．ここで，すべての事前・事後条件数を C_{all} とし，ツールで変換できた条件数を $C_{translatable}$ と定義する．

$$(C_{translatable}/C_{all}) \times 100(\%) \tag{3}$$

4.2.3 再現率の計測

再現率を式 4 と定義する．ここで，ツールで変換した JML のうち，理想的な JML と一致した数を C_{equal} と定義する．

$$(C_{equal}/C_{translatable}) \times 100(\%) \tag{4}$$

4.2.4 OCL から JML への変換時間の計測

変換時間は，OCL2JML に対して変換対象の UML と OCL を入力した瞬間から，JML の抽象構文木が出力されるまでの時間であると定義する．Java 標準ライブラリの System.currentTimeMillis() を利用すれば現在時刻が取得できるため，出力時刻と入力時刻の差分を計算することで計測を行う．

4.3 実験対象

実験対象として，IT Spiral で作成された実プロジェクトの教材データを用いた．具体的には，ある大学の教務システム開発における，実装フェーズで作成された約 200 のクラス図のうち，約 60 のクラスの中の約 400 のメソッドに対して，OCL と JML を記述した．制約を付加するクラスは，システムのコアとなる操作を実装しているものを中心に選択した．

OCL，JML を付加したクラスの内訳を表 1 に示す．また，図 4, 5 にシステムのクラス図の一部を示す．

4.4 結果

教務システムに付加した全条件数や正しく変換できた数の結果を，表 2 に示す．表 2 から，網羅率は 89.9%，再現率は 86.0% であり，OCL を用いて表現できた条件は，すべてツールを用いて変換できたことがわかる．

ここで，ツールから出力された JML のうち，理想的な JML

表 1　制約を付加したクラスとメソッド数の内訳

クラスの種類	クラス数	メソッド数
ServiceImpl	13	74
DAO	11	64
Util	6	29
DTO	2	25
Entity	30	216
計	62	408

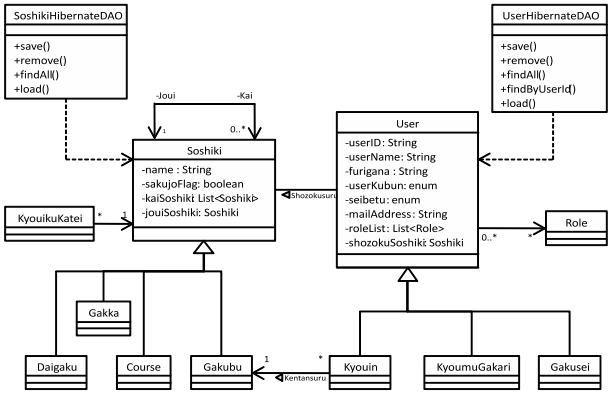

図 4　教務システムの Entity クラスの一部

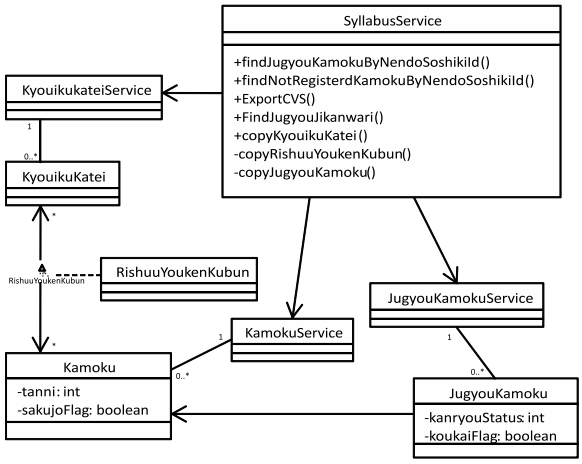

図 5　教務システムの Service クラスの一部

表 2　教務システムに対するツールの適用結果

全事前・事後条件数 (C_{all})	602
OCL で表現できた条件数	541
ツールで JML に変換できた条件数 ($C_{translatable}$)	541
変換後の JML が理想的な JML と一致した数 (C_{equal})	465
メソッド挿入による意味的な一致	57

```
public void save(final User user)
              throws ServiceException {
    getDAO(UserDAO.class).save(user);
}
```

図 6　UserServiceImpl クラスの save メソッド

と一致しなかったものは，2.3 節で述べた iterate 演算をメソッドとして出力した条件である．

OCL で表現できなかった条件式として，以下が挙げられる．これらは，変換できなかった JML のうち，ほぼ半分程度をそれぞれ占める．

（1） クラスリテラルを含む JML

（2） 配列表現を含む JML

変換できなかった式について，具体例を挙げる．

上記 1. のクラスリテラルを含む JML について述べる．そのような式は，主に Service クラスにおいて多く見られた．例えば，図 6 などである．

Service クラスのメソッドは図 6 のように，getDAO() の引数に適切な Class オブジェクトを入力することで，適切なデータアクセスオブジェクト (DAO) を取得し，そのオブジェクトを操作することでデータの保存を行う．このメソッドには事前条件として DAO が存在することを確認しなければならない．したがって，save メソッドには，`requires this.getDAO(UserDAO.class) != null;` といった JML を付加したが，UserDAO.class は Java 特有の言語仕様であり，OCL には対応する文法がないため変換できなかった．

上記 2. の配列について述べる．図 7 にその具体例を示す．

図 7 は setter であり，配列の各々の要素に対して，引数の値と this.kyoukasho の値が一致することを制約として記述している．しかし配列に対応する OCL の型は，本稿及び関連研究において，List として変換されるように定義されており，配列に変換されることは想定されていない．詳細は 4.5 節で述べる．

次に，教務システムに付加した JML の具体例について述べる．図 8 は教育課程に関する DAO クラス中のメソッドの 1 つである．

```
/*@
 ensures kyoukasho != null ?
   ( \forall int i4;
           i4>=0 && i4<this.kyoukasho.length;
           kyoukasho[i4] == null ||
             this.kyoukasho[i4].equals(kyoukasho[i4])
   ) && this.kyoukasho.length == kyoukasho.length
   : this.kyoukasho == null;
@*/
public void setKyoukasho
       (final JugyouShousaiKyoukasho[] kyoukasho) {
    this.kyoukasho = kyoukasho;
}
```

図 7　JML に配列を用いるメソッド

このメソッドはオブジェクトをデータベースから削除メソッドである．1 ～ 5 行目で，メソッドに対する JML を定義しており，2 行目は事前条件として引数が null でないことを定義し，3 行目で事前条件としてセッションが確立されていることを確認し，4 行目で事後条件としてデータが削除されたことを確認している．

図 8 を参考に作成した OCL を図 9 に示す．

図 8 の 4 行目の JML は，本稿での対応クラスの拡張により，うまく変換できた一例である．

4.5 考　　察

本ツールでは，60 クラスに OCL を適用した結果，約 90%の JML 式を再現できた．しかし，以下のような一部の JML 式は OCL では再現できなかった．ここでは，再現できなかった表現について考察する．

- クラスリテラル
- 配列

クラスリテラルは，Java 特有の言語仕様であるため，OCL には同等の概念は存在しない．代替案として，null でないクラス A の変数 aObj と Class クラスの変数 cObj の 2 つを UML に宣言し，cObj の不変条件を cObj = aObj.getClass() と設定することで，A.class の代わりに cObj を用いられるが，自動生成されたコードの複雑さが増す．

そもそも，仕様記述にクラスリテラルを使わなければならないような設計は，保守性や拡張性の観点から避けるべきである．本来ならば，抽象クラスやインターフェイスを用いてポリモルフィズムを活用し，クラス間の結合度を下げるべきである．

仕様記述を書くことは，品質の低い設計を明らかにし，より保守性，拡張性の高い設計へと設計し直す機会を与えるための重要なプロセスであると考える．

配列に関しては，OCL の文法上で配列が直接考慮はされていないが，配列に対応する型として Sequence が与えられている．Sequence はインデックス番号を保持する重複要素を許可

```
/*@
    requires soshiki != null;
    requires this.getSession() != null;
    ensures !(this.getSession().contains(soshiki));
@*/
public final void remove(final Soshiki soshiki) {
    super.remove(soshiki);
}
```

図 8　教務システムに対する JML の具体例

```
context SoshikiHibernateDAO::remove
                        (soshiki : Soshiki)
pre : not soshiki.oclIsUndefined()
pre : not self.getSession().oclIsUndefined()
post: not (self.getSession().contains(soshiki))
```

図 9　教務システムに対する OCL の具体例

表 3　配列の変換表 1

$\mu(seq_1$->includes$(a_1))$	=	$\mu(c_1$->exists($e \mid e = a_1))$
$\mu(seq_1$->excludes$(a_1))$	=	μ(not c_1->includes$(a_1))$
$\mu(seq_1$->includesAll$(seq_2))$	=	$\mu(seq_2$->forAll($e \mid seq_1$->includes$(e))$
$\mu(seq_1$->excludesAll$(seq_2))$	=	$\mu(seq_2$->forAll($e \mid seq_1$->excludes$(e))$
$\mu(seq_1$->isEmpty())	=	$\mu(seq_1$->forAll($e \mid e$.oclIsUndefined())
$\mu(seq_1$->notEmpty())	=	μ(not seq_1->isEmpty())
$\mu(seq_1$->size())	=	seq_1.length
$\mu(seq_1$->sum())	=	(\sum int i; $i >= 0$ && $i < seq_1$.length; s[i])
$\mu(seq_1$->count$(a_1))$	=	$\mu(seq_1$->iterate(e; acc : Integer = 0 | if $e = a_1$ then $acc + 1$ else acc endif))
$\mu(seq_1$->at$(index))$	=	seq_1.[$index$]
$\mu(seq_1$->first())	=	seq_1.[0]
$\mu(seq_1$->last())	=	seq_1.[seq_1.length-1]
$\mu(seq_1$->forAll$(exp))$	=	(\forall int i; $i >= 0$ && $i < seq_1$.length; exp)
$\mu(seq_1$->exists$(exp))$	=	(\exists int i; $i >= 0$ && $i < seq_1$.length; exp)

したデータの集合を表す.

Sequence 型は指定したインデックスに要素を挿入する insertAt() や，リストの最初にデータを追加する prepend() などの操作を持ち，その性質が可変長の List に近いことから，関連研究においても List として JML に変換することが考えられてきた．また，OMG が配列を可変長 List の概念に近い Sequence に抽象化した事実を考慮すると，Sequence で定義された型は，Java においては List で実装すべきであると考えられる．したがって，本ツールの実験として，4.3 節のプロジェクトを用いたことは一部不適切であった.

しかし，動作速度への要求や，多次元配列の実装などを考慮すると，開発者が Sequence を配列として実装することのニーズは少なくないと考えられるため，今後配列を考慮に入れた変換方法は必要である．Sequence から配列への変換規則のうち，容易に変換可能であるものを表 3 に示す．insertAt() などは，不足領域の確保や番号ずらしなど複雑な操作が要求されるため，専用のライブラリを定義する必要があると考えられる.

UML の解析時に，1 対多の関連をすべてリストアップし，配列として実装したい変数をユーザにチェックさせることで，Sequence の変換を分岐させることは十分に可能である．また，OCL ではコレクションのネストは本来推奨されていないが，collectNested() などの演算がサポートされており，多次元配列への対応もユーザの要求に任せられる.

5. あとがき

本稿では著者の所属する研究グループが提案する，OCL 記述を JML 記述へと変換する手法を紹介し，とりわけ iterate 演算の変換に対する手法を紹介することで，従来手法で提案されていたクラスより広いクラスに対して適用できることを示した．本研究ではその手法に対し，型推論や OCL-JML 変換表など具体的な実装方法を示した.

今後の課題としては，本ツールを完成させて評価実験を行うことと，MDA へ適用することの 2 点を考えている．ツールの評価に関しては，実用的な制約記述に対し実時間で変換できるか，与えた OCL 式に対して JML のランタイムアサーションチェッカが意図した結果を返すかどうかなどの評価を行うことを考えている．一方，MDA への適用では MOF と QVT を用いたモデル変換手法への適用も考えている．これは UML/OCL のメタモデルと，Java/JML のメタモデルのマッピングを定義することでモデル変換を実現する手法である.

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.

文　献

[1] A. 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] Eclipse Foundation: “Eclipse modeling framework”. http://www.eclipse.org/modeling/emf/.

[5] O. M. Group: “Ocl 2.0 specification” (2006). http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[6] G. Leavens, A. Baker and C. Ruby: “Jml: A notation for detailed design”, Behavioral Specifications of Businesses and Systems, pp. 175–188 (1999).

[7] “Object management group”. http://www.omg.org/.

[8] B. Meyer: “Eiffel: the language”, Prentice-Hall, Inc., Upper Saddle River, NJ (1992).

[9] 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).

[10] M. Rodion and R. Alessandra: “Implementing an ocl to jml translation tool”, 電子情報通信学会技術研究報告, 第 106 巻, pp. 13–17 (2006).

[11] C. Avila, G. Flores, Jr. and Y. Cheon: “A library-based approach to translating ocl constraints to jml assersions for runtime checking”, International Conference on Softw. Eng. Research and Practice, pp. 403–408 (2008).

[12] 尾鷲, 岡野, 楠本 : “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), 近代科学社, pp. 191–198 (2009).

[13] 宮澤, 岡野, 楠本 : “OCL の JML への変換ツールの実装”, 電子情報通信学会技術研究報告, **110**, 169, pp. 53–58 (2010).

[14] 宮澤, 岡野, 楠本 : “Ocl の jml への変換ツールの実装と評価” (2010).

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

[16] 尾鷲, 岡野, 楠本 : “メソッドの自動生成を用いた ocl の jml への変換”, コンピュータ ソフトウェア, **27**, 2, pp. 106–111 (2010).

[17] “Papyrus uml”. http://www.papyrusuml.org.

[18] “Eclipse modeling -mdt-”. http://www.eclipse.org/modeling/mdt/.

[19] A. Sarcar and Y. Cheon: “A new Eclipse-based JML compiler built using AST merging”, Department of Computer Science, The University of Texas at El Paso, Tech. Rep, pp. 10–08 (2010).