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
SS2010-25 (2010-8)

OCLのJMLへの変換ツールの実装について

宮澤　清介†　　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

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あびており，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．従来研究では OCL からの JML への変換についてコレクションに関するいくつかの重要な機能，とりわけ，iterate に非対応であった．研究グループではこの問題を，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方法で対応した．本稿ではその手法に基づき，OCL からの JML への変換ツールの実装について詳細に述べる．

キーワード　モデル変換，OCL，JML

On Implementation of a Translater from OCL into JML

Kiyoyuki MIYAZAWA†, 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

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 an implementation of a tool of the translation based on the proposed method.

Key words　model translation, OCL, JML

1. はじめに

OCL(Object Constraint Language) [1] 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 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 (Java Modeling Language) [2] が提案されている．JML，OCL ともに DbC (Design by Contract) [3] 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．

近年 MDA(Model Driven Architecture) [4] 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注目をあびている．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 [5], [6]，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 [7]．一方 OCL から JML への変換については Hamie が文献 [8] 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 [9] 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 [10] 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 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 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，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 [11]．本稿では実装について詳しく述べる．

以降，2 章で背景について述べ，3 章で文献 [11] で提案した手法について述べ，4 章で実装について述べ，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 JML-LUnit [13]，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 [5] や Harrison らの文献 [6] 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 [8] 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 [8] を基に，文献 [9] 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 [10] において，文献 [8] 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．また，iterate 演算の引数として与えられる OCL 式について，具体例として，式 (1) のような演算がある．これはコレクション *students* から *score* が 70 以上の要素を抽出したコレクションを返す演算である.

$$students\text{->}\ select(score > 70) \qquad (1)$$

JML や Java において $score > 70$ 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もある．例えば，式 (1) に対し，対応する Java メソッド *select*(*exp*) を用意した場合，OCL 式で与えられる引数 $score > 70$ が *exp* として与えられることになり，select の評価時の 1 回しか評価されず，以降、ループのたびに動的に繰り返し評価されない.

文献 [11] では個々のループ演算に対応する 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．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 [11] 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，文献 [11] 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.

本稿では変換に際し型情報を用いた，より厳密な変換プロセスを提案し，そのプロセスを用いて UML や OCL の入力部分から，JML を付加した Java コードを出力する部分までをツールとして実装することにより，利用者に利便性も与える.

3. iterate 演算の変換方法

この章では文献 [11] で提案した iterate 演算の変換方法について述べる．この変換方法は文献 [8], [9] の構文変換で未対応である，iterate 演算のについて対応している．そして，Collection の一部の演算を，iterate 演算を用いた式で表現することで，OCL-JML 変換の妥当性を向上させる方法について述べる.

3.1 iterate 演算の変換

iterate 演算の変換は次の理由により，単純な構文変換では対応できない.

- iterate 演算の引数はほぼ任意の OCL 式であり，単純な構文変換をするためには，変換先の言語 L において，L の任意式の動的な評価機構が必要となる.
- JML や Java1.6 はそのような機構を直接的にはサポートしていない.

本研究では，iterate 演算に引数として与えられる OCL 式を評価するメソッドを変換時に作成し，JML 式でそのメソッドの戻り値を参照することで，間接的にこの問題に対応する．具体的には，$\mu(c_1$ ->iterate(e; *init* | *body*)) に対し，初期化式 *init* で T_1 型の変数 *res* が初期化され，Collection のすべての要素 e に対して *body* を実行した評価結果が *res* に入る．この演算に対し，図 1 のメソッドを生成する．ここで，$\mu()$ は引数の式を Java 構文に変換する関数を表す.

変換例を以下に示す．以下の OCL 式 からメソッドを生成することを考える.

$$c\text{->}\ \ \text{iterate}(e;\ acc\text{: Integer} = 0\ |$$
$$\text{if } e = \text{`}ocl\text{'} \text{ then } acc + 1 \text{ else } acc \text{ endif}) \qquad (2)$$

図 1 の init と body に OCL 式 (2) の対応する部分を当ては

```
private T1 mPrivateUseForJML01() {
  μ(init);
  for (T2 e: μ(c1)){
    res = μ(body)}
return res;
}
```

図 1　iterate メソッド
Fig. 1　iterate method

めると，図 2 のようになる．

このメソッドに，μ 変換を適用することで，図 3 のメソッドが生成される．

iterate 演算を含む JML 式は，例として図 4 のように与えられる．

3.2 Collection 演算の変換

この節では，Collection 演算の一部を iterate 演算を用いた式に変換することについて述べる．iterate 演算を用いて表現できる Collection 演算の一部を表 1 に示す．これらの表現は OCL の定義書 [1] で定義されているため，OCL-JML 変換の妥当性を OCL 定義書に委ねることができるという点が利点である．一方欠点としては，iterate に対応するためのメソッドが多く挿入され，JML はそのメソッドをそれぞれ参照するため，Java 変換された JML 式の読解性が低下することが挙げられる．この問題の解決としては変換ツール実装時に，Collection 演算を iterate を用いて表現する場合としない場合の両方の変換に対応できるようにすることが考えられる．

4. 実　　装

本章では実装に関して詳細に述べる．図 5 にツールの概要を示す．このツールは最終的に Eclipse のプラグインとして開発することを考えている．

4.1 UML から Java への変換

UML クラス図から Java ソースコードに変換する方法に関しては，Eclipse UML Studio Edition を利用する．このツールを選定した理由として，属性に対する型の割り当てや関連に

```
private int mPrivateUseForJML01() {
  μ(acc : Integer = 0);
  for (String e : c){
    acc = μ( if e = 'ocl' then acc + 1 else acc endif)
  }
  return acc;
}
```

図 2　init と body に式を割り当て

Fig. 2　assignment of the OCL expression into init and body

```
private int mPrivateUseForJML01() {
  int acc = 0;
  for (String e : c){
    acc=(e.equals("ocl")? acc + 1: acc) };
  return acc;
}
```

図 3　Java 変換後の iterate メソッド

Fig. 3　iterate method after Java translation

```
mPrivateUseForJML01()> 0
```

図 4　iterate メソッドの JML からの呼び出し

Fig. 4　the call of iterate method from JML expression

表 1　Collection-Iterate 対応表

Table 1　correspondence table of the collection and iteration

c_1->exists(a_1 | a_2)	=	c_1->iterate(a_1; res : Boolean = false | res or a_2)
c_1->forAll(a_1 | a_2)	=	c_1->iterate(a_1; res : Boolean = true | res and a_2)
c_1->count(a_1)	=	c_1->iterate(e; acc : Integer = 0 | if $e = a_1$ then $acc + 1$ else acc endif)
c_1->isUnique(a_1 | a_2)	=	c_1->collect(a_1 | Tuple {iter=Tuple{a_1}, value=a_2}) -> forAll(x, y | ($x.iter$ <> $y.iter$) implies $x.value$ <> $y.value$)
c_1->any(a_1 | a_2)	=	c_1->select(a_1 | a_2)->asSequence()->first()
c_1->one(a_1 | a_2)	=	c_1->select(a_1 | a_2)->size()= 1
c_1->collect(a_1 | a_2)	=	c_1->collectNested(a_1 | a_2)->flatten()

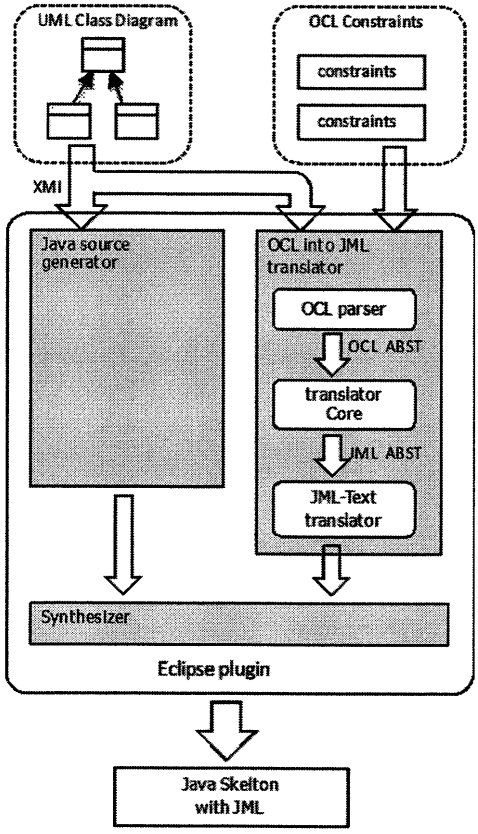

図 5　ツール全体図

Fig. 5　Tool Overview

関する設定などを詳細に記述することができる点や，出力される XMI ファイルの構造がコンテキスト情報の抽出に適している点などが挙げられる．

UML から得られるコンテキスト情報から，以下の情報を利用する．

- クラス名
- 属性名
- 属性の型
- 操作名
- 操作の引数の型
- 操作の戻り値の型

- 関連端名
- 多重度
- 継承しているクラス

4.2.2節で詳細に述べるが，OCLからJMLへ変換するための情報として，変数や演算結果の型の情報が必要になる．OCLの構文では属性や操作の参照は非常に頻繁に起こるため，それらの名前と型をコンテキスト情報として必要とするのは自明である．変数や，操作の戻り値の型はユーザが定義した型である場合があり得るので，クラス名の情報も必要になる．生成されたJMLをより正確なものにするため，操作の引数の型も与えておくことで，OCL構文解析の段階で型不整合は検出しておきたい．また，関連しているクラスの情報を用いた演算も頻繁に利用される．そのためには関連端名が必要である．誘導の型は，ユーザ定義型もしくはユーザ定義型のコレクションのいずれかである．その型は多重度によって決まるため，多重度の情報が必要となる．関連クラスは，あるクラスのサブクラスである可能性があり，スーパークラスの属性や操作を利用するためには，継承クラス情報が必要になる．以上により，UMLから利用するコンテキスト情報の必要性を述べた．UMLには他に可視化の情報も含まれるが，操作の多用でOCLを複雑にさせないため，利用はしない．

文献[9]ではUMLクラス図描画ツールにVioletが用いられているが，VioletはUMLの作図を簡単な操作で素早く実現することに特化しており，属性の型を定義するなどの詳細な情報を持たせることができない．このため，出力されるXMIファイルも厳密に定義されたものではないので，属性や関連などの情報取得が困難だと考え，本研究では新たにツールの選定を実施した．

4.2 OCLからJMLへの変換

4.2.1 OCL構文解析器の実装

OCL構文解析器の実装には，ANTLRを利用する．ANTLRはLL(k)構文解析を用いた構文解析器自動生成ツールである．構文解析器の出力言語としてJavaをサポートしており，構文解析器と同時に字句解析器も出力することから，ツール開発の速度を向上させることができる．

4.2.2 型 推 論

この節では，構文解析器に型情報を付加することについて述べる．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，構文木のノードが変数や演算の型を保持しなければならない．

ANTLRパーサが出力するOCL抽象構文木のノードは型情報を持たないので，ノードクラスを拡張することで対応した．拡張したノードは，以下の3つの情報を持つこととする．

- token: 演算子，feature名，終端token値など
- type: 該当ノードの部分木に対する型
- children: 子ノードリスト
- flag: 部分木が関連もしくは，コレクション型を戻り値とする演算であることを表すフラグ

また，UMLのコンテキスト情報を得るために，ANTLRパーサを拡張してUML情報が記述されたXMIファイルを入力できるようにする．XMIファイルから，属性・操作名をキーとしてその型名を返すMapを作成し，OCL構文内に属性・操作名が現れたとき，Mapから型を取得する．

Collection型の演算にはcount(),size(),isEmpty()など，様々な演算がOCLの標準で与えられているが，この型推論も，ユーザ定義の操作名から型情報を得る方法と同様に，演算名をキーとしてその型名を返すMapで解決することを考えている．

また，UMLやJMLでは数値を表す型としてbyte, short, char, int, long, float, doubleといった型が定義されているが，OCLではInteger型とReal型の2種類のみしか定義されていない．Integer, Realの両型とも，上限は定義されていないため，本稿ではInteger型はint型に，Real型はdouble型に対応させることとする．設計の自由度に制限を与えないため，OCLでは対応していないlongやfloatなどの型推論も，評価可能にする．演算による型の評価は，JMLのものと統一する．例えば，$byte + short$の評価型はint，$long * float$の評価型はfloatなどである．coまた，ここでOclVoid型の扱いについて述べる．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として扱われるが，nullとの違う点として，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．この点に関しては，OCLの構文解析器に動的な評価機構を持たせられないため，JMLに出力する際に工夫を施す必要がある．そのために，OCL抽象構文木のノードにflagを持たせる．詳細は4.2.5節で述べる．

ユーザ定義型の型推論に関しては，OCLの定義書に従い，'=','<>', oclIsUndefined(), oclIsKindOf(), oclIsTypeOf(), oclIsNew(), oslAsType(), oclInState()に対応する．

OCL定義書で定義されているOCL構文からは型的に不整合な表現式が導出され得るが，そのような型不整合な式はJavaやJMLで直接扱えないため，例外を出力し，コンパイルエラーになるようにする．

4.2.3 Collection演算のiterate演算への変換

3.2節で述べたように，一部のCollection演算をiterate演算に変換する方法について述べる．対応方法としては，各演算をiterate演算で表現した構文木をテンプレートとして与えておき，ノードのリンクを変更するなどの方法を考えている．具体例として，c ->count('ocl')をiterate演算を用いた構文木に変換することを考える．count演算をiterate演算を用いた式で表すと，c_1->iterate(e; acc : Integer = 0 | if $e = a_1$ then $acc + 1$ else acc endif) で表される．構文木テンプレートは図6のように与えられる．

このテンプレートに対し，eに当てはまる型名とa_1に当てはまるOCL抽象構文木を入力することで，テンプレート内の

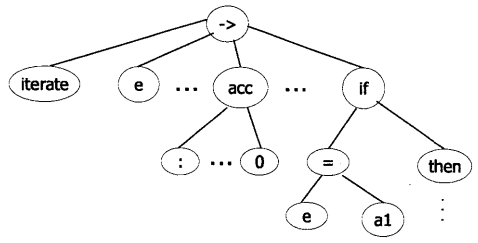

図 6　count 演算の構文木テンプレート
Fig. 6　The tree template of count operation

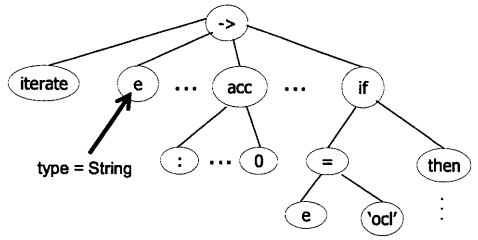

図 7　count 演算を iterate 変換した結果
Fig. 7　The result of translation from count operation to iterate operation

ノードを変更することを考えている．この場合，型名に String，OCL 構文木に *'ocl'* の構文木を与える．テンプレートに情報を与えて出力される OCL 抽象構文木は図 7 で表される．

4.2.4　OCL-JML 変換アルゴリズム

以下の変換規則に従って，OCL 抽象構文木を葉に近いノードから順に，JML 抽象構文木へと変換する．文献 [8] の記法に従い，OCL 式から JML 式への変換関数の表記を μ で与える．スペースの都合上，変換規則の一部を掲載する（表 2，3，4）．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．

ここで，変換する JML の構文木の構成について述べる．構文木の各ノードには以下の情報を持たせることを考えている．

- token: 演算子，feature 名など
- feature: ノードの種類
- children: 子ノードリスト

表 2　数値型の μ 変換表
Table 2　μ translation table of the numeric type

$\mu(a_1 = a_2)$	=	$\mu(a_1) == \mu(a_2)$
$\mu(a_1 > a_2)$	=	$\mu(a_1) > \mu(a_2)$
$\mu(a_1 < a_2)$	=	$\mu(a_1) < \mu(a_2)$
$\mu(a_1 >= a_2)$	=	$\mu(a_1) >= \mu(a_2)$
$\mu(a_1 <= a_2)$	=	$\mu(a_1) <= \mu(a_2)$
$\mu(a_1 <> a_2)$	=	$\mu(a_1)! = \mu(a_2)$

表 3　Collection 型の μ 変換表
Table 3　μ translation table of the collection type

$\mu(c_1 = c_2)$	=	$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 > c_2)$	=	$\mu(c_1)$.containsAll($\mu(c_2)$)&&!$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 < c_2)$	=	$\mu(c_2)$.containsAll($\mu(c_1)$)&&!$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 >= c_2)$	=	$\mu(c_1)$.containsAll($\mu(c_2)$)
$\mu(c_1 <= c_2)$	=	$\mu(c_2)$.containsAll($\mu(c_1)$)
$\mu(c_1 <> c_2)$	=	!$\mu(c_1)$.equals($\mu(c_2)$)

表 4　Collection 演算の μ 変換表
Table 4　μ translation table of the operation of the collection type

$\mu(c_1$->size())	=	$\mu(c_1)$.size()
$\mu(c_1$->includes(a_1))	=	$\mu(c_1)$.has($\mu(a_1)$)
$\mu(c_1$->isEmpty())	=	$\mu(c_1)$.isEmpty()
$\mu(c_1$->notEmpty())	=	!$\mu(c_1)$.isEmpty()
$\mu(c_1$->excludes(a_1))	=	$\mu(c_1$->count(a_1) = 0)
$\mu(c_1$->count(a_1))	=	$\mu(c_1$->iterate(e; acc : Integer = 0 | if $e = a_1$ then $acc + 1$ else acc endif))

- flag: 部分木が関連もしくは，コレクション型を戻り値とする演算であることを表すフラグ

feature 情報を用いて，JML を文字列化する際に文字列を出力する順序を制御する（詳細は 4.2.5 節で述べる）．

flag は 4.2.2 節で述べた，undefined を用いた論理演算の差異を吸収するために用いる（詳細は 4.2.5 節で述べる）．

具体的な JML 抽象構文木の例として，コンテキスト宣言を含む構文木，iterate 演算を含む構文木に対して，それぞれ図 8，9 でその構成を示す．

4.2.5　JML-Text トランスレータ

この節では，OCL-JML 変換によって得られた JML 抽象構文木を入力とし，構文木をテキストとして出力する方法について述べる．まず，JML トランスレータクラスは以下のデータ構造を持つクラスのリストを，フィールドとして持つこととする．

- className：制約を加える対象となるクラス名
- methodName：制約を加える対象となるメソッド名
- jmlText：挿入する JML 式のテキスト
- iterateMethod：文字列化した iterate メソッドのリスト

まず図 8 のコンテキスト情報から JML 文を挿入するべきクラ

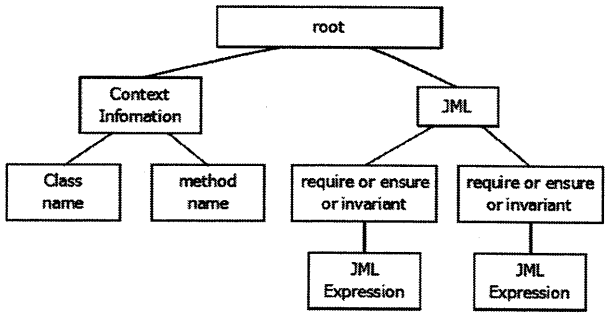

図 8　コンテキスト宣言を含む JML 抽象構文木
Fig. 8　JML ABST Tree containing context declaration

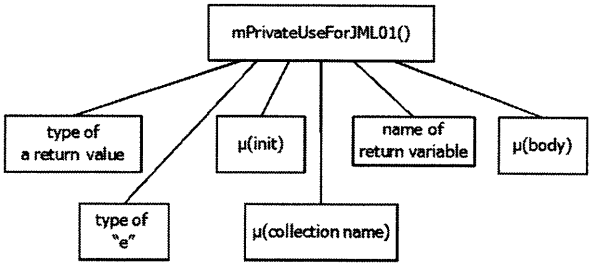

図 9　iterate 演算を含む JML 抽象構文木
Fig. 9　JML ABST Tree containing iterate operation

```
"public" + type of a return value
                + "mPrivateUseForJML01(){ \n"
 + "\t" + init + ";\n"
 + "\t for(" + type of e + " e :"
                + collection name "){\n"
 + "\t\t" + name of return variable "="
                              + body + "}\n"
 + "\t return" + name of return variable + ";}"
```

図 10　iterateMethod に格納するテキスト
Fig. 10　generation text of iterate method

スとメソッド名を取得し，それぞれ className と methodName に格納する．次に，JML の部分木を以下の方法に従い文字列化し，jmlText に格納する．

iterate 演算を用いた構文は，jmlText 用には mPrivateUseForJML() を出力し，iterateMethod に図 10 のように生成したテキストを追加する．

4.2.2 節で述べた Undefined を用いた論理演算への対応であるが，flag を用いて解決する．例えば，焦点を当てているノードの種類が 'or' であり，その右の子の flag が True であったとき，or 演算の式が True or Undefined である可能性がある．この条件を満たす場合，JML-Text トランスレータは論理式の右辺として以下のコードを出力する．

```
(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．

4.3　Java と JML の統合

この節では，Java と JML の統合を行う（図 5 の Synthesizer に相当する）モジュールの実装について述べる．

まず，className を用いて JML を挿入する Java ファイルを決定する．methodName が null(挿入する制約式が不変条件) の場合，クラス宣言の 1 行下に jmlText を挿入する．methodName が null でない（挿入する制約式が事前条件または事後条件）場合，methodName を参照して JML を挿入するメソッドを特定し，メソッド宣言文の 1 行上に jmlText を挿入する．

最後に，Java ファイルの一番後ろに iterateMethod の要素をすべて挿入する．

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] O. M. Group: "Ocl 2.0 specification" (2006). http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.
[2] G. Leavens, A. Baker and C. Ruby: "Jml: A notation for detailed design", Behavioral Specifications of Businesses and Systems, pp. 175–188 (1999).
[3] B. Meyer: "Eiffel: the language", Prentice-Hall, Inc., Upper Saddle River, NJ (1992).
[4] 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).
[5] 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).
[6] 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).
[7] Eclipse Foundation: "Eclipse modeling framework". http://www.eclipse.org/modeling/emf/.
[8] 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).
[9] 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).
[10] 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).
[11] 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).
[12] J. Specs: "Samples of jml specifications". http://www.eecs.ucf.edu/~leavens/JML/examples.shtml.
[13] 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).
[14] A. de Pellegrin: "Violet. uml modeling tool" (2007). http://www.horstmann.com/violet/.