Vol.2010-SE-170 No.19
2010/11/12

OCL の JML への変換ツールの実装と評価

宮 澤 清 介[†1]　岡 野 浩 三[†1]　楠 本 真 二[†1]

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注目をあびており，それに伴い OCL(Object Constraint Language) から JML(Java Modelling Language) 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に対し，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，その手法に基づいた OCL からの JML への変換ツールの実装方法を示した．本稿では従来研究で対応していなかった演算の変換規則と入れ子になった iterate の変換方法を示し，ツールに実装した．1 メソッドに対して，ツールが出力した JML と手書きのものを用いて JML ランタイムアサーションチェッカの実行時間を比較した結果，両方とも 1ms で実行完了した．また，変換速度を計測したところ，約 20ms で変換できた．

On Implementation and Evaluation of a Translater from OCL into JML

Kiyoyuki Miyazawa ,[†1] Kozo Okano [†1]
and Shinji Kusumoto[†1]

In recent years, MDA techniques have been strong developed, thus translation techniques such as translation from OCL (Object Constraint Language) 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 and implemented a tool. In this report, we present the new translation rules, how to translate nested iterate, as well as implementation of the tool and its evaluation. In this evaluation, we compared execution time of the JML runtime assertion checker using the JML automatically generated and handwritten one. Both of them are 1ms. In addition, execution time of OCL-JML translation is about 20ms.

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[11)]，簡単な実装を行った[12)]．本稿では文献[11),12)] で議論が不十分だった点を実装し，小規模な評価実験を行った．

以降，2. で背景について述べ，3. で文献[11)] で提案した手法と議論が不十分だった点について述べる．そして 4. で実装について述べ，5. で評価について述べ，6. でまとめる．

2. 準　　備

ここ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

†1 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, OsakaUniversity

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 JML ランタイムアサーションチェッカ（以下 JMLrac）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[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．次に iterate 演算の具体例とその対応の難しさについて述べる．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．具体例として，式（1）のような演算が挙げられる．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を意味する．

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

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がある．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．

本研究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ことを目的としている．

3. iterate 演算の変換方法

この章では，まず文献[11] で提案した iterate 演算の変換方法について述べる．次に，文献[11] では議論が不十分だったメソッドの引数の利用と iterate 演算の入れ子への対応について述べる．最後に，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．

文献[11] 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方法を示した．

iterate 演算の一般形は，$c_1{-}{>}\text{iterate}(e; init \mid body))$ で表され，作成されるメソッドは図 1 のようになる．$\mu()$ は引数の式を Java 構文に変換する関数を表し，res と T_1 はそれぞれ，$init$ 内で定義された変数名と型を表す．また，T_2 は e の型を表す．

具体例として，iterate 演算（2）からメソッドを生成することを考える．

$$\begin{aligned} c{-}{>}\ & \text{iterate}(e;\ acc\text{: Integer} = 0 \mid \\ & \quad \text{if } e = \text{`}ocl' \text{ then } acc + 1 \text{ else } acc \text{ endif}) \end{aligned} \tag{2}$$

acc: Integer $= 0$ が $init$ に対応しており，$\mu(init)$ は int acc $= 0$ と定義する．同様に，if $e =$ ‘ocl' then $acc + 1$ else acc endif が $body$ に対応しており，$\mu(body)$ は (e.equals(“ocl”)?

```
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

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

図 2 Java 変換後の iterate メソッド
Fig. 2 iterate method after Java translation

$acc + 1$: acc) と定義する．図 1 の res は acc に置き換えられる．

すなわち，iterate 演算（2）から生成されるメソッドは，図 2 になる．

3.2 iterate 演算の入れ子

3.1 節で述べた文献[11] の方法では，iterate 演算が入れ子になったときに正しく変換できない．例えば iterate 演算（3）の，$c2$−>iterate ∼ から変換されるメソッドは図 3 のように表される．このメソッド内で $e1$ は宣言されていないので，参照できない．

$$\begin{aligned} c_1\text{−>}\ &\text{iterate}(\ e1 : \text{String}\ ;\ acc1\ \text{Integer}\ = 0\ \mid \\ &c_2\text{−>}\ \text{iterate}(e2 : \text{String}\ ;\ acc2\ : \text{Boolean}\ = false\ \mid \\ &\quad \text{if}\ e1 = e2\ \text{then}\ true\ \text{else}\quad false\ \text{endif})) \end{aligned} \tag{3}$$

また，文献[11] の方法ではメソッド内で使用される変数にはフィールド変数のみしか想定しておらず，JML 挿入先のメソッドの引数の情報を利用することができない．この問題を解決するため，JML 挿入先のメソッドの引数と，変換する iterate 演算よりも上層で定義された変数の名前と型を mPrivateUseForJML() メソッドの引数として利用した．

```
private boolean mPrivateUseForJML02() {
  boolean acc2 = false;
  for (String e2 : c2){
    acc2=(e1.equals(e2) ? true : false) };
  return acc2;
}
```

図 3 入れ子の iterate 演算を変換した結果
Fig. 3 result of the translation of nesting iterate operation

3.3 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るため，出力された JML の可読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．

表 1 Collection-Iterate 対応表
Table 1 correspondence table of the collection and iteration

c_1−>exists($a_1 \mid a_2$)	=	c_1−>iterate(a_1; res : Boolean = false | res or a_2)
c_1−>forAll($a_1 \mid a_2$)	=	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($a_1 \mid a_2$)	=	c_1−>collect(a_1 | Tuple {iter=Tuple{a_1}, value=a_2}) −> forAll($x, y \mid (x.iter <> y.iter)$ implies $x.value <> y.value$)
c_1−>any($a_1 \mid a_2$)	=	c_1−>select($a_1 \mid a_2$)−>asSequence()−>first()
c_1−>one($a_1 \mid a_2$)	=	c_1−>select($a_1 \mid a_2$)−>size()= 1
c_1−>collect($a_1 \mid a_2$)	=	c_1−>collectNested($a_1 \mid a_2$)−>flatten()

4. 実　　装

ここでは実装に関して詳細に述べる．図 4 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．

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．それ以外の実装については，文献[12] と同様の内容であるため，省略する．

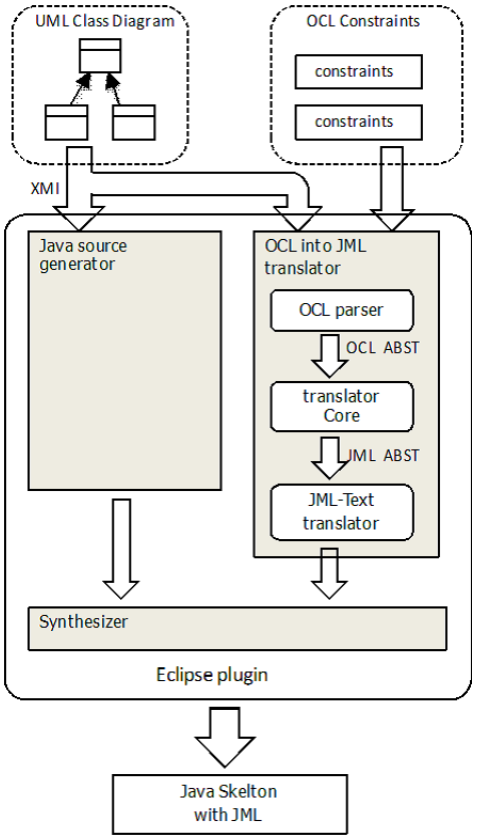

図 4　ツール全体図
Fig. 4　Tool Overview

4.1 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．

文献[1] に掲載されている OCL の EBNF は，重複した文法や左再帰を用いたものが多数含まれており，また四則演算や論理演算などの文法，演算の優先順位の情報が欠落していたため，その EBNF を ANTLR に入力しても，構文解析器は出力されなかった．そこでまず，重複文法を除去し，左再帰除去アルゴリズムに従って左再帰を除去した．さらに文献[14] を参考に，欠落していた演算や優先順位の情報を EBNF に付加することで構文解析器を出力することができた．

4.2 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 Undefined を取り得る演算であることを示すフラグ

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した．この XMI ファイルは，Eclispe UML から出力されたものを使用する．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．

同様に OCL の標準演算も，操作名をキーとして演算の型名を返す Map を作成すること

で型推論を可能にした．

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 型に対応させた．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した．演算による型の評価は，JML のものと統一した．例えば，$byte + short$ の評価型は int，$long * float$ の評価型は float などである．

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した．

4.3 OCL-JML 変換規則

文献[11] で定義された OCL-JML の変換規則の一部を表 2，3，4 で示す．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 で与えている．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．

また，文献[11] では対応が不十分だった変換規則を新たに定義した．その変換規則を表 5 に示す．

4.4 iterate の JML 抽象構文木

iterate の入れ子や挿入先メソッドの引数の参照に対応するため，文献[12] で示した構文木を拡張し，より多くの情報を扱えるようにした．具体的には，挿入するメソッド名をルートノードとし，そのノードは次の 8 つの情報を子ノードリストとして保持する．1. 引数情報（挿入先メソッドの引数リストと上層の iterate で定義された変数），2. 戻り値の型，3. $\mu(init)$，4. イテレータ変数 e の型，5. イテレータ変数 e の変数名，6. コレクション変数の名前，7. 戻り値となる変数名，8. $\mu(body)$

例えば，iterate 演算（3）の，$c2$−>iterate ∼ を JML に変換すると，mPrivateUseForJML02 をルートノードとし，次の 8 つの情報を含んだ JML 抽象構文木が出力される．1. int acc, String e1，2. boolean，3. acc2 = false，4. String，5. e2，6. c2，7. acc2，8. (e1.equals(e2 ? true : false))

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()
$\mu(c_1$−>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))

表 5 新たに定義した μ 変換
Table 5 new μ translation

a_1.oclIsKindOf(a_2)	=	$\mu(a_2)$.class.isAssignableFrom($\mu(a_1)$.getClass())
a_1.oclIsTypeOf(a_2)	=	$\mu(a_1)$.getClass().equals($\mu(a_2)$)
a.oclIsUndefined()	=	$\mu(a)$ == null

5. 評　　価

5.1 方　　法

図 5 で示す在庫管理システムの UML クラス図に対して OCL を付加し，評価を行った．

今回の評価では，Storage クラスの checkStockSatisfied() メソッド（図 6）に対して，図 7 の OCL を付加した．図 5 は，我々の研究グループの過去の研究[15] で作成した UML で

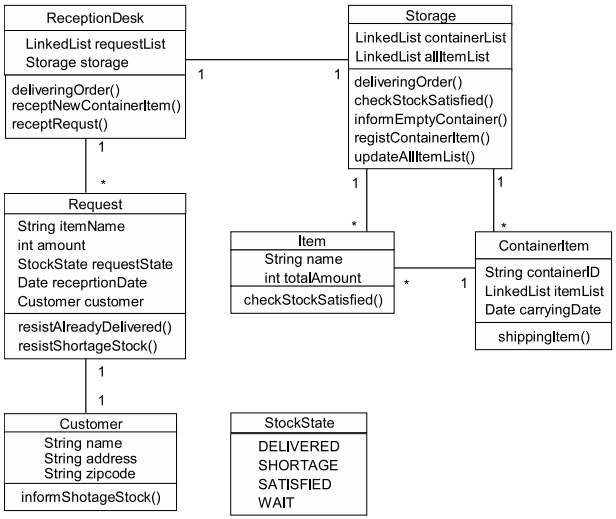

図 5 在庫管理システムの UML クラス図
Fig. 5 UML class diagram of a Stock Management System

あり，JML を用いて設計されている．

本稿の評価では，文献[15] で付加した手書きの JML と，本稿のツールを用いて OCL から自動変換した JML の品質について評価する．品質は以下の観点から評価する．

(1) 手書きの JML 表現に対し，どの程度近づけられるか

(2) JMLrac の実行速度が遅くならないか

本評価で JMLrac を用いる理由は，次の 3 点である．1. iterate を変換したメソッドには必ず for 文が挿入される． 2. for 文のループ回数も考慮してチェックできるツールはこれ以外に存在しない．3. JML 記述の品質の評価方法として，間接的だが客観的な比較を期待できる．

なお，checkStockSatisfied() メソッドは，顧客が注文した商品の発注量は倉庫にある商品の在庫よりも少ないかどうかをチェックするメソッドである．具体的には，倉庫（Storage クラス）が注文（Request クラス）を受けたら，注文に含まれている商品（Item クラス）と同じ名前の商品を，倉庫が持つ商品リストから探す．倉庫に注文と同じ商品名を持つ商品があれば，商品の発注量 (r.getAmount()) と在庫量 (item.getTotalAmount()) を比較し，発注量の方が少なければ true，多ければ false を返す．また，倉庫に注文と同じ商品が存在し

```
public boolean checkStockSatisfied(Request r){
  Iterator i = allItemList.iterator();
  while(i.hasNext()){
    Item item = (Item)i.next();
    if(r.getItemName().equals(item.getName())
          && r.getAmount() <= item.getTotalAmount()){
      return true;
    }
  }
  return false;
}
```

図 6 checkStockSatisfied() メソッドの実装
Fig. 6 implementation of the checkStockSatisfied() method

```
context Storage::checkStockSatisfied(r : Request)
pre : not r.oclIsUndefined()
post: result = allItemList->exists(i : Item | r.getItemName() = i.getName()
                                  and r.getAmount() <= i.getTotalAmount())
```

図 7 入力 OCL 文
Fig. 7 input OCL expression

なければ無ければ false を返す．

挿入する OCL（図 7）は，checkStockSatisfied() メソッドが実行される前提として，注文が null でないことをチェックし，メソッドが終了したときには，「倉庫内に注文と同じ商品がありかつその商品の発注量が倉庫内の在庫よりも少ないという条件を満たす商品が存在するか否か」が，リターンされることを示している．

5.2 実験環境

本稿の評価では，JMLrac に JML4c を用いる．JML4c は，本稿の iterate メソッドで利用している Java1.5 の拡張 for 文やジェネリクスに対応した JMLrac であり，JML を付加した Java ソースコードを入力すると，JML と実装の違いをチェックする機能が付加された実行ファイルを出力する．この実行ファイルの実行時に，JML で記述した仕様と異なる動

作をした場合，違反した行数やそのときの変数の値などが出力される．

実験環境として，CPU は Intel(R) Core(TM)2 Duo E7300 2.66GHz 2.67GHz，メモリは 4.00GB，OS は Windows Vista 64bit を用いた．

5.3 結　　果

5.3.1 出力された JML

図 7 をツールを利用して JML に変換した結果，図 8 の JML 文が出力された．表 1 で示したように，exists 演算は iterate 演算に置換して変換するため，mPrivateUseForJML01() のようにメソッドとして出力されている．mPrivateUseForJML01() の内容は，図 9 に示す．一方，文献[15] で使用されている JML 記述を図 10 に示す．

本ツールには，図 10 の 1 行目と 3 行目の文を出力する機能はないが，JMLrac を利用するにあたって，これらの文は不要である．また，変換の正しさの確認においては，図 8 の 1 行目と図 10 の 2 行目の対応と，図 8 の 2 行目と図 10 の 3,4,5 行目の対応をそれぞれ見比べると，容易に意味の正しさが理解できる．OCL の構文，`not r.oclIsUndefined()` を忠実に JML に変換すると，`!(r == null)` になってしまうが，`r != null` という記述の方が一般的で可読性が高いと思われる．ユーザにとって，変換の忠実性を多少犠牲にしてでも可読性を向上させたいケースがあると考えられるので，変換のオプション機能で優先順位を設定できるような機能を設けることも考えている．

```
requires !(r == null);
ensures \result == mPrivateUseForJML1(r);
```

図 8　OCL を JML に変換した結果
Fig. 8　the result of OCL2JML translation

```
private boolean mPrivateUseForJML1(Request r){
  res = false;
  for(Item i : allItemList){
    res = res || r.getItemName().equals(i.getName())
          && r.getAmount() <= i.getTotalAmount();
  }
  return res;
}
```

図 9　mPrivateUseForJML01() の内容
Fig. 9　the content of the mPrivateUseForJML01()

```
public behavior
  requires r != null;
  assignable \nothing;
  ensures \result == (\exists Item i;allItemList.contains(i);
                      r.getItemName().equals(i.getName()) &&
                      r.getAmount() <= i.getTotalAmount());
```

図 10　手書きの JML 記述
Fig. 10　handwritten JML expression

5.3.2 JMLrac の実行時間

図 6 のメソッドに対し，手書きの JML と本ツールで生成した JML をそれぞれ付加して，図 11 を用いて JMLrac を実行したところ，両方とも 1ms でチェックが完了した．この結果から，JML のネイティブな演算と，本ツールで生成したメソッドの実行時間に大差がないことが推測できる．

5.3.3 OCL-JML 変換の実行時間

図 7 の OCL 文を構文解析する時間は約 20ms，OCL 抽象構文木を JML 抽象構文木に変換する時間は約 1ms だった．図 5 の在庫管理システムの属性と操作すべてに OCL を付加しても，約 600ms 程度で変換できる計算である．属性と操作を合計して 100 個持つような大規模なシステムでも，約 2 秒で変換できる計算であるため，十分に実用的であると推測できる．今後，より大規模な評価実験を行い，この仮説の正しさを検証したい．

6. 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，未対応であった iterate の入れ子などに対応し，型推論や OCL-JML 変換表など具体的な実装方法を示して，簡単な評価実験を行った．

今後の課題として，より大規模な評価実験を行うことを考えている．今回の実験では一つ

```
public static void main(String args[]){
  Item item1 = new Item(”Item1”, 50); Item item2 = new Item(”Item2”, 100);
  Item item3 = new Item(”Item3”, 30); Item item4 = new Item(”Item4”, 200);

  LinkedList<Item> list = new LinkedList<Item>();
  list.add(item1); list.add(item2); list.add(item3); list.add(item4);

  Storage st = new Storage(); st.setAllItemList(list);

  Customer c = new Customer(); Request r = new Request(”item4”, 400, c);

  long start,stop;
  start = System.currentTimeMillis();
  st.checkStockSatisfied(r);
  stop = System.currentTimeMillis();

  System.out.println(”Running Time is :” + (stop - start) + ”ms”);
}
```

図 11 テストケース
Fig. 11 test case

のメソッドに OCL を記述して，変換速度を計測し，JMLrac の実行速度の比較を行った．手書きの JML とツールで出力した JML の品質に大きな差が観測されなかったが，この結果は実験規模が小さいことが影響の一つとして考えられる．大規模な評価実験では，すべてのメソッドに対して OCL を付加し，手書きの場合と比較した品質を調査したい．また，サポートできていない演算などの対応や GUI の構築，出力される JML の可読性向上などに取り組みたい．

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．

参 考 文 献

1) Group, O.M.: OCL 2.0 Specification (2006). http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.
2) Leavens, G., Baker, A. and Ruby, C.: JML: A Notation for Detailed Design, *Behavioral Specifications of Businesses and Systems*, pp.175–188 (1999).
3) Meyer, B.: *Eiffel: the language*, Prentice-Hall, Inc., Upper Saddle River, NJ (1992).
4) Kleppe, A., Warmer, J. and Bast, W.: *MDA explained: the model driven architecture: practice and promise*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2003).
5) Engels, G., Hücking, R., Sauer, S. and Wagner, A.: UML collaboration diagrams and their transformation to Java, *UML1999 -Beyond the Standard, Second International Conference*, pp.473–488 (1999).
6) Harrison, W., Barton, C. and Raghavachari, M.: 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) Hamie, A.: 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) Rodion, M. and Alessandra, R.: Implementing an OCL to JML translation tool, 電子情報通信学会技術研究報告，Vol.106, pp.13–17 (2006).
10) Avila, C., Flores, Jr., G. and Cheon, Y.: 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，コンピュータ ソフトウェア， Vol.27, No.2, pp.106–111 (2010).
12) 宮澤清介，岡野浩三，楠本真二：OCL の JML への変換ツールの実装について，*IEICE technical report*, Vol.110, No.169, pp.53–58 (2010).
13) Cheon, Y. and Leavens, G.: A simple and practical approach to unit testing: The JML and JUnit way, *ECOOP 2002 Object-Oriented Programming*, pp.1789–1901 (2006).
14) Warmer, J. and Kleppe, A.: UML/MDA のためのオブジェクト制約言語 OCL，エスアイビーアクセス，2 edition.
15) 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).