修士学位論文

題目

# OCL から JML への変換ツールの<br>対応クラスの拡張と実プロジェクトに対する適用

指導教員

楠本　真二　教授

報告者

宮澤　清介

平成 23 年 2 月 7 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

平成 22 年度 修士学位論文

# OCL から JML への変換ツールの<br>対応クラスの拡張と実プロジェクトに対する適用

宮澤　清介

## 内容梗概

近年 MDA(Model Driven Architecture) 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．それに伴い OCL(Object Constraint Language) から JML(Java Modeling Language) 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) 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 への自動変換については研究が少なく，従来研究ではいずれの方法も 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．しかしそれは提案のみに留まり，実装した上での手法の有用性や，変換の妥当性の確認までは行っていない．

本研究では iterate 演算に対応した OCL から JML への変換手法を実装し，実プロジェクトに適用して評価を行った．

実装や評価の過程で，変換の妥当性が低いと考えられるものを修正し，変換規則として不足しているものを補足，さらに Java 標準ライブラリなどの，プロジェクトに直接含まれないクラスも扱えるように拡張した．

評価では，7 クラス程度の規模のプロジェクトと，約 60 クラス程度の規模のものに適用を行った．結果として，システムに付加すべき制約のうち 90%を網羅し，約 86%が最適な JML を出力することを確認した．また，ツールは実用時間で OCL を JML に変換できることを確認した．

### 主な用語

OCL, JML, Model Driven Architecture, Design by Contract, UML

# 目 次

# 図 目 次

# 表 目 次

# 1 まえがき

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]．しかし文献 [12] は提案のみに留まり，実装した上での手法の有用性や，変換の妥当性の確認までは行っていない．

本研究では文献 [12] で提案した手法を実装し，実プロジェクトに適用して評価を行った．

実装や評価の過程で，変換の妥当性が低いと考えられるものを修正し，変換規則として不足しているものを補足した．具体的には，以下の補足・修正を行った．

- OCL-JML 変換規則の誤りを訂正
- oclIsUndefined() などの変換規則を追加
- iterate の入れ子に対応
- Java 標準ライブラリなどの，プロジェクトに含まれるクラス以外のものに対応

評価では，約 10 クラス程度の規模のプロジェクトと，約 70 クラス程度の規模のものに適用を行った．結果として，システムに付加すべき制約のうち 85%を網羅し，約 82%が適切な JML を出力することを確認した．また，ツールは実用時間で OCL を JML に変換できることを確認した．

以降，2. で背景について述べ，3. で文献 [12] 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 研究背景

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 Design by Contract

Design by Contract(以降，DbC とする) は，オブジェクト指向のソフトウェア設計に関する概念の１つで，クラスとそのクラスを利用する側との間で仕様の取り決めを契約とみなすことにより，ソフトウェアの品質，信頼性，再利用性を向上させることを目指している．契約は，クラスの利用側がそのクラスを利用する際にある条件 (事前条件) を保証すれば，そのクラスはある性質 (事後条件) を満たすことを保証するというものである．事前条件が満たせない場合はクラスを利用する側，事後条件が満たせない場合はクラス側の責任となる．このような責任の分離は開発者ごとの作業の分担を明確にし，ソフトウェアの欠陥の原因を切り分けるのに役立つ．

### 2.2 OCL

OCL 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．

図 1 に OCL による制約の記述例を示す．図 1 の例では，Account クラスのメソッド withdraw に対して OCL により制約を記述している．“ pre ”が事前条件，“ post ”が事後条件を表している．また，“ \result ”によりメソッドの戻り値を表し，変数に“ @pre ”を記述することで，メソッドの実行前の変数の値を参照できる．

すなわち，withdraw メソッドを実行する事前条件として，与えられる引数が 0 よりも大きいことと balance の値よりも小さいことが要求される．また，事後条件として balance の値から引数の値が減算された値が balance に格納され，戻り値として返されることが要求される．

```
context Account::withdraw(val:Integer): Integer
pre : balance - val > 0
pre : val > 0
post: \result = balance@pre - val
post: balance = balance@pre - val
```

図 1: OCL による制約の記述例

```
1  public class Account{
2     int balance;
3     /*@
4         requires balance - val > 0;
5         requires val > 0;
6         ensures \result == \old(balance) - val;
7         ensures balance == \old(balance) - val;
8     @*/
9     public int withdraw(int val){
10       balance -= val;
11       return balance;
12    }
13 }
```

図 2: JML による制約の記述例

### 2.3 JML

JML は，Java のメソッドやオブジェクトに対して，DbC に基づいた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，JML 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．

図 2 に JML による制約の記述例を示す．図 2 の例では，2.2 節と同様に，Account クラスのメソッド withdraw に対して JML により制約を記述している．“ requires ”が事前条件，“ ensures ”が事後条件を表している．また，“ \result ”によりメソッドの戻り値を表し，“ \old() ”は，メソッド実行前の変数の値の参照を表す．

JML には，コード実行時に JML 記述の内容と実行時の値が矛盾しないことをチェックする 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[14] 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．

### 2.4 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．具体的には，JML や Java に直接対応していない $setOfSets{->}flatten()$ などの演算を，式 1 のように，汎用のライブラリを定義することによって解決した．

$$\text{JMLTools.flatten}(setOfSets) \tag{1}$$

また，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．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（2）のような演算が挙げられる．

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

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（2）に対し，式 1 のように対応する Java メソッド $iterate()$ を用意した場合，式 3 のように変換されることが想定できる．

$$\text{JMLTools.iterate}(\text{int } i, \text{int } sum = 0, sum + i, set) \tag{3}$$

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．

著者の研究グループでは，文献 [15] で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 [15] 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 [15] 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 [15] で提案した iterate 演算の変換方法について述べる．次に，文献 [15] 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 [15] 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，式（4）で表される．ここで，$c$ はコレクション型の変数，$e$ はイテレータ変数，$init$ は戻り値変数の宣言と初期化式，$body$ はループ内で実行される式を表す．

$$c{->}\quad \text{iterate}(e;\, init \mid body) \tag{4}$$

作成されるメソッドは図 3 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（5）からメソッドを生成することを考える．

$$\begin{aligned} c{->}\quad & \text{iterate}(e;\, acc\text{: Integer} = 0 \mid \\ & \text{if } e = \text{`}ocl\text{'} \text{ then } acc + 1 \text{ else } \quad acc \text{ endif}) \end{aligned} \tag{5}$$

```
1 private T1 mPrivateUseForJML01(){
2  μ(init);
3  for (T2 e: μ(c1)){
4   res = μ(body) }
5  return res;
6 }
```

図 3: iterate メソッド

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

図 4: Java 変換後の iterate メソッド

$acc$: 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")? $acc + 1$: $acc$) と定義する．図 3 の $res$ は $acc$ に置き換えられる．

すなわち，iterate 演算（5）から生成されるメソッドは，図 4 になる．

### 3.2 iterate 演算の入れ子

3.1 節で述べた文献 [15] 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（6）の，$c2$−>iterate . . . から変換されるメソッドは図 5 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．

$$
\begin{aligned}
c_1 \text{->}\ \text{iterate}(\quad & e1 : \text{String} ; acc1\ \text{Integer}\ = 0 \mid \\
& c_2 \text{->}\ \text{iterate}(e2 : \text{String} ; acc2 \quad : \text{Boolean}\ = false \mid \\
& \quad \text{if } e1 = e2 \text{ then } true \text{ else} \qquad false \text{ endif}))
\end{aligned} \tag{6}
$$

また，文献 [15] 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 5: 入れ子の iterate 演算を既存の方法で変換した結果

修正後の mPrivateUseForJML02() は図 6 のようになる．

### 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 演算を表 1 に示す．これらの表現は OCL の定義書 [5] 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．

```
1 private boolean mPrivateUseForJML02(String e1, int acc1) {
2     boolean acc2 = false;
3     for (String e2 : c2){
4         acc2=(e1.equals(e2) ? true : false) };
5     return acc2;
6 }
```

図 6: 修正後の iterate の入れ子

表 1: Collection-Iterate 対応表

| | | |
|---|---|---|
| $c_1$−>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) |
| $c_1$−>product($c_2$) | = | $c_1$−>iterate( $e_1$; $acc_1$ : Set(Tuple(first: $T$, second: $T_2$) = Set{} \| |
| | | $c_2$−>iterate( $e_2$; $acc_2$ : Set(Tuple(first: $T$, second: $T_2$) = $acc_1$ \| |
| | | $acc_2$ −>including(Tuple{first=$e_1$, second=$e_2$})))) |
| $st_1$−>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) |
| $st_1$−>reject($a_1 \mid a_2$)) | = | $st_1$−>select( $a_1$ \| not $a_2$) |
| $c_1$−>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 |
| $st_1$−>collectNested($a_1 \mid a_2$)) | = | $st_1$−>iterate( $a_1$; $res$ : Bag($a_2.type$)= Bag{} \| |
| | | $res$ −>including($a_2$)) |
| $c_1$−>collect($a_1 \mid a_2$) | = | $c_1$−>collectNested( $a_1 \mid a_2$)−>flatten() |
| $c_1$−>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 $x.value <> y.value$) |
| $st_1$−>sortedBy($a_1 \mid a_2$)) | = | $st_1$−>iterate( $a_1$; $res$ : OrderedSet($T$)= OrderedSet{} \| |
| | | if $res$−>isEmpty() then $res$.append($a_1$) |
| | | else $res$.insertAt($res$−>indexOf( |
| | | $res$−>select($i \mid a_2(i) > a_2(a_1)$ −>first())),$a_1$) |
| | | endif) |

# 4 実装

ここでは実装に関して詳細に述べる．図 7 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．

## 4.1 外部仕様

ツールの外部仕様を，図 8 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．

## 4.2 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，1 章で述べたように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[16] 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)[17] 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．

文献 [10] では UML クラス図描画ツールに Violet[18] 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．

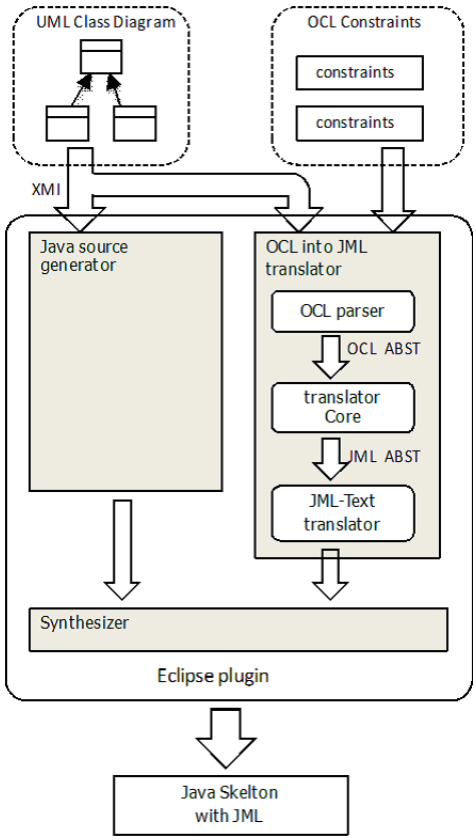


図 7: ツール全体図

### 4.3 OCL Parser の実装

#### 4.3.1 構文解析

構文解析の実装には，ANTLR[19] を利用した．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 [5] 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 [20] 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．修正した OCL の EBNF を付録に掲載する．OCL の文法の規模は，EBNF で約 50 種類の非終端記号と約 100 種類の生成規則で構成された．

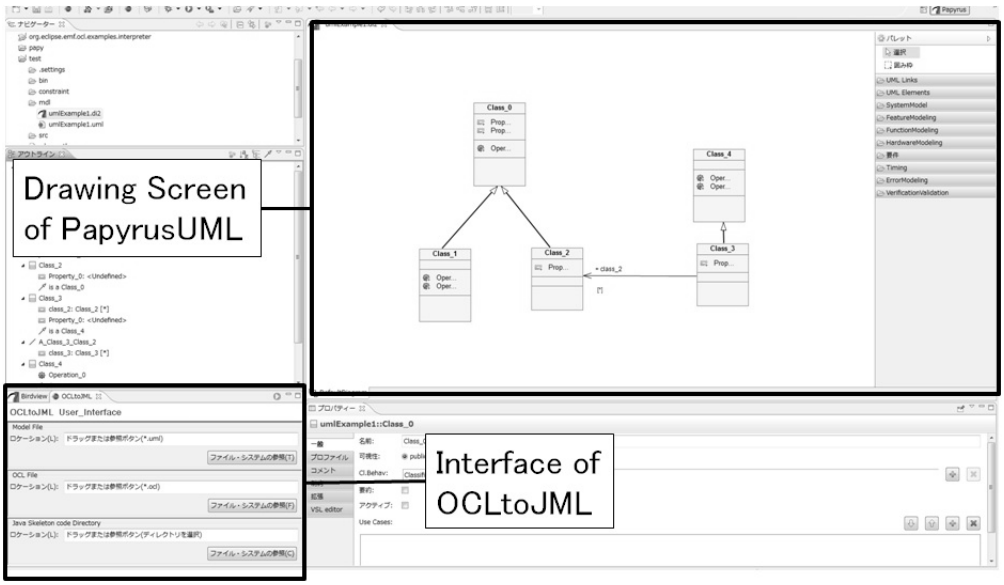


図 8: ツールの概観

#### 4.3.2 意味解析

この節では，4.3.1 節で実装した構文解析を拡張して，意味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 から得られる構文解析器（以下 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 4 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 XML ファイルを入力できるようにした．この XML ファイルは，4.3.1 節で述べた UML CASE ツールから出力されたものを使用する．UML から得られるコンテキスト情報から，以下の 10 種類の情報を使用する．1. クラス名，2. 属性名，3. 属性の型，4. 操作名，5. 操作の引数の名前，6. 操作の引数の型，7. 操作の戻り値の型，8. 関連端名，9. 多重度，10. 継承しているクラス

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．操作の引数の名前と型の情報を用いることで，3.1 節で述べた iterate 演算の変換の問題に対応できる．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，使用しない．

XML 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 構文内に属性や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 を作成することで意味解析を可能にした．

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．すなわち，$byte + short$ 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，コンパイルエラーとした．

### 4.3.3 対応クラスの拡張

4.3 節で述べた 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．この理由により，外部や標準ライブラリのクラスにも対応できるように拡張する必要があった．

拡張方法として，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 に追加するアプローチを取った．OCL Parser で直接対応するアプローチも考えられるが，標準ライブラリに含まれるクラスの数が膨大であり，かつ外部ライブラリのインポートに柔軟に対応することが困難であったため，前述のアプローチを採用した．

ライブラリ中のクラスを UML に変換するために，Java2UML[21] というツールを用いた．これは

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 プラグインである．

本研究では，UMLFromJava の入力を，Java ファイル単位で入力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．

#### 4.3.4 OclVoid 型の扱い

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．正確に OclVoid を扱うためにはツール側で対応するオブジェクトを用意し，差異を適切に吸収する必要がある．この点に関しては，OCL Parser 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 4.6.1 節で述べる．

### 4.4 OCL-JML 変換規則

文献 [15] 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 [9] の記法に従い，OCL 式から JML 式への変換関数の表記を $\mu$ 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 [15] 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．

### 4.5 JML 構文木の構成

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た．

表 2: 数値型の $\mu$ 変換表

$$
\begin{aligned}
\mu(a_1 = a_2) &= \mu(a_1) == \mu(a_2) \\
\mu(a_1 > a_2) &= \mu(a_1) > \mu(a_2) \\
\mu(a_1 < a_2) &= \mu(a_1) < \mu(a_2) \\
\mu(a_1 >= a_2) &= \mu(a_1) >= \mu(a_2) \\
\mu(a_1 <= a_2) &= \mu(a_1) <= \mu(a_2) \\
\mu(a_1 <> a_2) &= \mu(a_1) != \mu(a_2)
\end{aligned}
$$

表 3: Collection 型の $\mu$ 変換表

| | | |
|---|---|---|
| $\mu(c_1 = c_2)$ | = | $\mu(c_1)$.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 $\mu$ 変換表

| | | |
|---|---|---|
| $\mu(c_1$->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)) |

- token: 演算子，feature 名など
- 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

ノードの種類ごとに，JML を文字列化する際の出力順序が異なるため，すべてのノードに共通する操作は抽象クラスとして定義し，異なる部分はサブクラスとして定義することで制御を変えた.（詳細は 4.6.1 節で述べる）.

flag は 4.3.2 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 4.6.1 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 9, 10 でその構成を示す.

表 5: 新たに定義した $\mu$ 変換

| | | |
|---|---|---|
| $a_1$.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 |

### 4.6 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 [22] 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（6）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)) c

#### 4.6.1 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
- methodArguments : 制約を加える対象となるメソッドの引数
- 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 9 のコンテキスト情報から JML 文を挿入するべきクラスとメソッド名を取得し，それぞれ className と methodName に格納する．

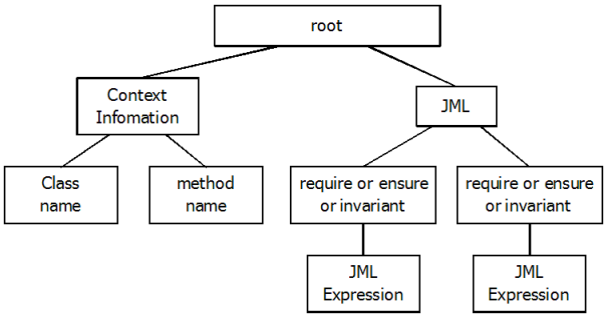


図 9: コンテキスト宣言を含む JML 抽象構文木

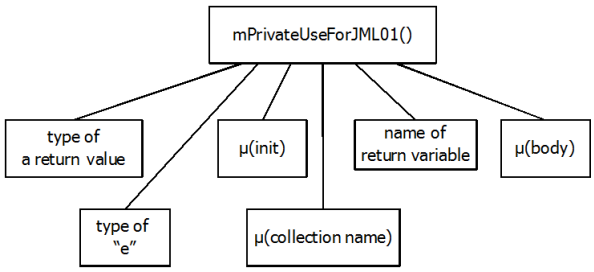


図 10: iterate 演算を含む JML 抽象構文木

iterate 演算を用いた構文は，jmlText 用には mPrivateUseForJML() を出力し，iterateMethod に図 3 に一致するようにテキストを生成する．図 10 で挙げたように，*init* や *body* などのテキスト生成に必要な情報はすべて含まれているので，容易に生成することができる．

4.3.2 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．

# 5 評価実験

ここでは，評価実験について詳細に述べる．本研究では適用したプロジェクトの大きさに応じて，二回の実験を行った．規模の小さいプロジェクトに対して適用した評価実験を実験 1，規模の大きいものに対して適用した評価実験を実験 2 とし，それぞれ詳細に述べる．

## 5.1 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

**動的検査の実行速度** 理想的な JML と比較して，動的検査の実行速度が遅くならない

理想的な JML 記述の定義は，5.2.1 節で後述する．
また，ツールの実用性の評価するために，OCL を JML に変換し終わるまでの時間を計測する．

## 5.2 方法

### 5.2.1 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．

### 5.2.2 網羅率の計測

本稿では網羅率を式 7 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．

$$C_{translatable}/C_{all} \tag{7}$$

### 5.2.3 再現率の計測

再現率を式 8 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．

$$C_{equal}/C_{translatable} \tag{8}$$

#### 5.2.4 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．

#### 5.2.5 JML の実行速度の比較

直接 JML を付加した場合と，ツールを用いて変換した JML では iterate 演算の扱いが異なるため，実行速度を比較する．その手順として，JML 付きの Java ファイルを JML コンパイラによりコンパイルし，出力された実行ファイルを JMLrac で実行する．

本稿の評価では，JML コンパイラに JML4c[23] を，JMLrac に JML4rt[23] を用いる．JML4c は，JML が付加された Java ファイルを入力すると，JML の制約情報が付加された実行ファイルを出力するコンパイラである．我々が調査した限りでは，Java1.5 の拡張 for 文やジェネリクスに対応した唯一の JML コンパイラである．

一方 JML4rt は，JML4c によって出力された実行ファイルを入力すると，プログラムの実行時の値と JML で記述された条件に矛盾が生じないことを検査する，JMLrac である．実行時に JML で記述した条件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．

これらのツールを使用した理由として，以下の三点が挙げられる．

1. 我々の変換アプローチは，iterate 演算を変換したメソッドに必ず拡張 for 文を挿入する
2. 拡張 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

### 5.3 実験対象

#### 5.3.1 実験 1

実験 1 では，図 11 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．図 11 は，我々の研究グループの過去の研究 [24] 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．なお，図 11 は簡単のため，すべてのメソッドと属性は記述していない．

実験 1 の評価では，20 個のメソッドに対して OCL を付加した．その一例として，Storage クラスの checkStockSatisfied() メソッド（図 13）に対して付加した OCL を，図 12 に示す．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())

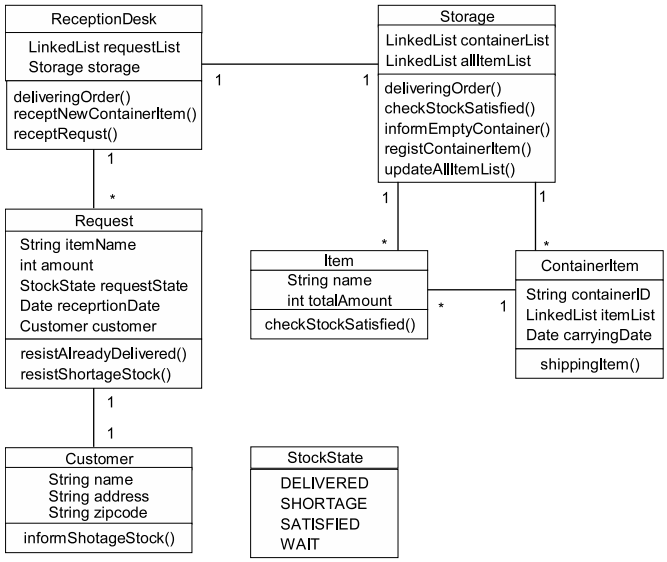


図 11: 在庫管理システムの UML クラス図

```
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 12: 入力 OCL 文

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なければ無ければ false を返す.

挿入する OCL（図 12）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.

### 5.3.2 実験 2

実験 2 では文部科学省先導的 IT スペシャリスト育成推進プログラム (IT Spiral[25]) 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.

```
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 13: checkStockSatisfied() メソッドの実装

表 6: 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 |

OCL，JML を付加したクラスの内訳を表 6 に示す．また，図 14, 15 にシステムのクラス図の一部を示す.

### 5.4 実験環境

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.

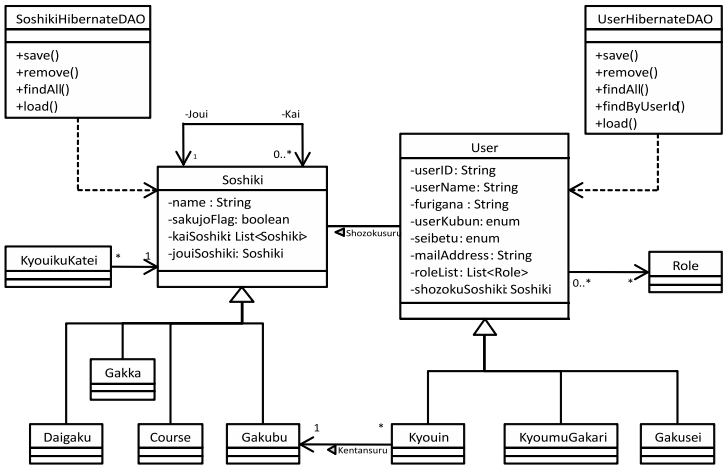


図 14: 教務システムの Entity クラスの一部

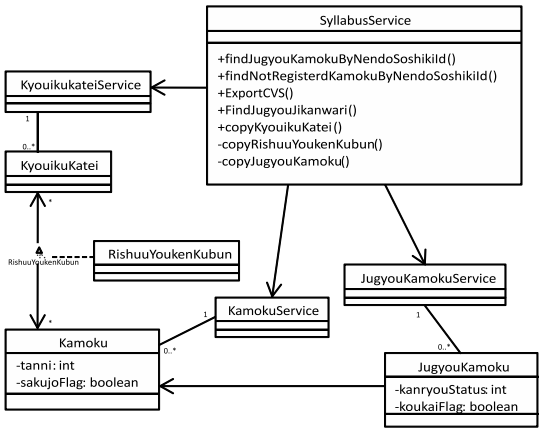


図 15: 教務システムの Service クラスの一部

## 5.5 結果

### 5.5.1 実験 1

### 5.5.2 出力された JML

20 個の OCL を，ツールを利用して JML に変換した．その一例として，5.3.1 節の図 12 を変換した結果を図 16 に示す．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 17 に示す．一方，文献 [24] で使用されている JML 記述を図 18 に示す．

変換の正しさの確認においては，図 16 の 1 行目と図 18 の 2 行目の対応と，図 16 の 2 行目と図 18 の 4,5,6 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，

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

図 16: OCL を JML に変換した結果

```
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 17: mPrivateUseForJML01() の内容

`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．したがって，実験 2 ではこの欠点を改善している．

図 18 の 1 行目は，通常時の動作と例外発生時の動作の制約を別々に記述する場合に記述されるが，我々が調査した限りでは，OCL にそのような文法は存在しない．3 行目の assignable キーワードは，メソッド実行中に状態が変化し得る変数を表す．図 18 では nothing であるので，どの変数も状態が変化してはならないことを示す．1,3 行目ともに，JML4rt では影響がなかったが，ESC/Java2 などといった別のツールでは影響があるため，対応すべき問題だと考えられる．

```
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 18: 手書きの JML 記述

```
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 19: テストケース

#### 5.5.3 JMLrac の実行時間

図 13 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 19 を用いて JMLrac を実行したところ，両方とも約 30ms 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．

#### 5.5.4 OCL-JML 変換の実行時間

図 12 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 11 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．

#### 5.5.5 実験 2

教務システムに付加した全条件数やツールを用いて変換できた条件数の結果を，表 7 に示す．表 7 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 と一致しなかったものは，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 2. 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 20 や図 21 などである．

Service クラスのメソッドは図 20, 21 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や読み込みを行う．両メソッドとも，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，load メソッドにも同様のものを付加したのだが，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 3. の配列について述べる．図 22 にその具体例を示す．

図 22 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 5.6 節で述べる．

表 7: 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 |

```
1  public void save(final User user) throws ServiceException {
2      getDAO(UserDAO.class).save(user);
3  }
```

図 20: UserServiceImpl クラスの save メソッド

```
1  public Role load(final int id) throws ServiceException {
2      return getDAO(RoleDAO.class).load(id);
3  }
```

図 21: RoleServiceImpl クラスの load メソッド

```
1  /*@
2    ensures kyoukasho != null ?
3              ( \forall int i4; i4>=0 && i4<this.kyoukasho.length;
4                        kyoukasho[i4] == null ||
5                        this.kyoukasho[i4].equals(kyoukasho[i4])
6              ) && this.kyoukasho.length == kyoukasho.length
7              : this.kyoukasho == null;
8  @*/
9  public void setKyoukasho(final JugyouShousaiKyoukasho[] kyoukasho) {
10     this.kyoukasho = kyoukasho;
11 }
```

図 22: JML に配列を用いるメソッド

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 23 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．

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 23 を参考に作成した OCL を図 24 に示す.

図 23 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．

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

図 23: 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 24: 教務システムに対する OCL の具体例

### 5.6 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．

- assignable キーワード
- クラスリテラル
- 配列

assignable キーワードは前述の通り，OCL にはない概念である．しかし，OCL の事前条件・事後条件で参照される変数の情報から，十分に推測可能であると考えられる．

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，5.3.2 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 8, 9 に示す．insertAt() などは，不足領域の確保や索引のシフト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．

表 8: Sequence の配列への変換規則 1

| | | |
|---|---|---|
| $\mu(seq_1 -> \text{includes}(a_1))$ | $=$ | $\mu(c_1 -> \text{exists}(\ e \mid e = a_1))$ |
| $\mu(seq_1 -> \text{excludes}(a_1))$ | $=$ | $\mu(\text{not}\ c_1 -> \text{includes}(a_1))$ |
| $\mu(seq_1 -> \text{includesAll}(seq_2))$ | $=$ | $\mu(seq_2 -> \text{forAll}(\ e \mid seq_1 -> \text{includes}(e))$ |
| $\mu(seq_1 -> \text{excludesAll}(seq_2))$ | $=$ | $\mu(seq_2 -> \text{forAll}(\ e \mid seq_1 -> \text{excludes}(e))$ |
| $\mu(seq_1 -> \text{isEmpty}())$ | $=$ | $\mu(seq_1 -> \text{forAll}(\ e \mid e.\text{oclIsUndefined}())$ |
| $\mu(seq_1 -> \text{notEmpty}())$ | $=$ | $\mu(\text{not}\ seq_1 -> \text{isEmpty}())$ |
| $\mu(seq_1 -> \text{size}())$ | $=$ | $seq_1.\text{length}$ |

表 9: Sequence の配列への変換規則 2

$$
\begin{array}{lcl}
\mu(seq_1\text{->sum()}) & = & (\backslash\text{sum int } i;\ i >= 0\ \&\&\ i < seq_1\text{.length; s[i])} \\
\mu(seq_1\text{->count}(a_1)) & = & \mu(seq_1\text{->iterate(}\ e;\ acc : \text{Integer} = 0 \mid \\
 & & \qquad \text{if } e = a_1 \text{ then } acc + 1 \text{ else } acc \text{ endif)} \\
\mu(seq_1\text{->at}(index)) & = & seq_1\text{.}[index] \\
\mu(seq_1\text{->first()}) & = & seq_1\text{.[0]} \\
\mu(seq_1\text{->last()}) & = & seq_1\text{.}[seq_1\text{.length-1]} \\
\mu(seq_1\text{->forAll}(exp)) & = & (\backslash\text{forall int } i;\ i >= 0\ \&\&\ i < seq_1\text{.length; } exp) \\
\mu(seq_1\text{->exists}(exp)) & = & (\backslash\text{exists int } i;\ i >= 0\ \&\&\ i < seq_1\text{.length; } exp)
\end{array}
$$

## 6 あとがき

近年の MDA 関連技術の発展を受けて，本稿では iterate 演算に対応した，OCL から JML への変換手法を実装し，さらに対応クラスを拡張した．対応クラスを拡張した結果，Java 標準ライブラリなどの本来は UML クラス図に描かないクラスの情報を扱うことが容易になった．

実装したツールを実用規模のプロジェクトに適用したところ，理想的な JML と意味が同等な JML を，OCL により約 86%網羅でき，網羅できた JML のうち，理想的な JML と一致するものは，約 82%となる結果を得ることができた．さらに約 600 の OCL 式に対し，約 7 秒で変換が終了することを確認し，ツールが十分に実用的であることを示した．

今後の方針として，MDA の主流である，メタモデルを用いたモデル変換の研究が考えられる．本稿では Java を用いてマッピングを記述したが，OMG の定める標準は QVT(Queries/ Views/Transformations)[26] である．また，ATL(ATLAS Transfomation Language)[27] は Eclipse から開発キットが出ており，多数のユーザを獲得している．

また，JML から OCL に変換するという研究も考えられる．OCL を様々な言語に変換可能であることを仮定すると，JML から OCL に容易に変換が可能であれば，その OCL を元に Java 以外の言語に変換できる可能性が大きくなる．すなわち，Java 言語で書かれたシステムの別の言語への移植が容易になることが予想される．

## 謝辞

本研究を行うにあたり，理解ある個指導を賜り，常に励ましていただきました楠本真二教授に心から感謝申し上げます.

本研究の全過程を通じ，終始熱心かつご丁寧なご指導を頂きました岡野浩三准教授に深く感謝申し上げます.

本研究に多大なるご助言ご指導頂きました肥後芳樹助教に心より感謝申し上げます.

本研究の実験に対して，多大なる協力を頂きました大阪大学基礎工学部情報科学科 4 年生，花田健太郎氏に深く感謝申し上げます.

その他楠本研究室の皆様のご協力に，心より感謝いたします.

## 参考文献


[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. In *UML1999 -Beyond the Standard, Second International Conference*, pp. 473–488, 1999.

[3] W Harrison, C Barton, and M Raghavachari. Mapping uml designs to java. In *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] Object Management 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 *In Proc. of the 2004 ACM symposium on Applied computing*, pp. 1531–1535, 2004.

[10] Moiseev Rodion and Russo 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. In *International Conference on Softw. Eng. Research and Practice*, pp. 403–408, 2008.

[12] 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.

[13] 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.

[14] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 234–245, 2002.

[15] 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.

[16] Papyrus uml. `http://www.papyrusuml.org`.

[17] Eclipse modeling -mdt-. `http://www.eclipse.org/modeling/mdt/`.

[18] A. de Pellegrin. Violet. uml modeling tool, 2007. `http://www.horstmann.com/violet/`.

[19] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Language*. Pragmatic Bookshelf, 2007.

[20] Jos Warmer and Anneke Kleppe. 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, 第 2 版.

[21] Atos Origin TOPCASED Team. How to generate uml from java. `http://gforge.enseeiht.fr/docman/view.php/7/279/TPC_2.4_Java_Reverse_tutorial.pdf`.

[22] 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.

[23] 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.

[24] 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.

[25] It スパイラル. `http://it-spiral.ist.osaka-u.ac.jp/`.

[26] 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/`.

[27] 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.

[28] M Elaasar and L C Briand. An overview of uml consistency management. Technical report, Carleton University, 2008.

[29] C Lange, M R V Chaudron, J Muskens, L J Somers, and H M Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *In Proc. of Workshop on Consistency Problems in UML-based Software Development II*, pp. 26–34, 2003.

[30] JML Specs. Samples of jml specifications. `http://www.eecs.ucf.edu/~leavens/JML/examples.shtml`.

[31] P Chalin, P R James, and G Karabotsos. Jml4: Towards an industrial grade ive for java and next generation research platform for jml. In *Proc. of the International Conference on Verified Software: Theories, Tools, Experiments*, pp. 70–83, 2008.

[32] 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. 電子情報通信学会技術研究報告, Vol. 110, No. 169, pp. 53–58, 2010.

[33] 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と評価, 2010.

[34] 尾鷲方志, 岡野浩三, 楠本真二. 在庫管理プログラムの設計に対する jml 記述と esc/java2 を用いた検証の事例報告. 電子情報通信学会論文誌, Vol. J91D, No. 11, pp. 2719–2720, 2008.

[35] 尾鷲方志, 岡野浩三, 楠本真二. iterate の変換. `http://sdl.ist.osaka-u.ac.jp/~m-owasi/OclJMLTrans.pdf`.

付録

# 修正後の OCL の EBNF 式

表 10: OCL 構文 1

```
packageDeclarationCS      ::=  PACKAGE pathNameCS {contextDeclCS} ENDPACKAGE |
                               {contextDeclCS};
contextDeclCS             ::=  CONTEXT pathNameCS (operationContextDeclCS |
                                    attrOrAssocContextCS |
                                    classifierContextDeclCS);
operationContextDeclCS    ::=  operationCS prePostOrBodyDeclCS;
attrOrAssocContextCS      ::=  DCOLON simpleNameCS [COLON typeCS] initOrDerValueCS;
classifierContextDeclCS   ::=  invOrDefCS;
initOrDerValueCS          ::=  INIT COLON oclExp [initOrDerValueCS] |
                               DERIVE COLON oclExp [initOrDerValueCS];
operationCS               ::=  LPAR [variableDeclarationListCS] RPAR [COLON typeCS];
invOrDefCS                ::=  INV [simpleNameCS] COLON oclExp [invOrDefCS] |
                               DEF [simpleNameCS] COLON defExpressionCS [invOrDefCS];
defExpressionCS           ::=  variableDeclarationCS2 EQUAL oclExp |
                               pathNameCS operationCS EQUAL oclExp;
prePostOrBodyDeclCS       ::=  (PRE | POST | BODY) [simpleNameCS] COLON
                                                    oclExp [prePostOrBodyDeclCS]
simpleNameCS              ::=  SIMPLESTRING | ALPHA;
pathNameCS                ::=  simpleNameCS [ DCOLON pathNameCS ];
collectionLiteralExpCS    ::=  collectionTypeIdentifierCS LBRACE
                                                [collectionLiteralPartsCS] RBRACE;
collectionTypeIdentifierCS ::= COLLECTION | SET | BAG |
                               SEQUENCE | ORDEREDSET;
collectionLiteralPartsCS  ::=  collectionLiteralPartCS [COMMA collectionLiteralPartsCS];
collectionLiteralPartCS   ::=  oclExp [DDOT oclExp];
tupleLiteralExpCS         ::=  TUPLE LBRACE variableDeclarationListCS RBRACE;
```

表 11: OCL 構文 2

| | | |
|---|---|---|
| primitiveLiteralExpCS | ::= | integerLiteralExpCS \| |
| | | realLiteralExpCS \| |
| | | stringLiteralExpCS \| |
| | | booleanLiteralExpCS; |
| integerLiteralExpCS | ::= | INT \| DIGIT; |
| realLiteralExpCS | ::= | REAL; |
| stringLiteralExpCS | ::= | ”’ {PACKAGE \| ENDPACKAGE \| ARROW \| |
| | | COLON \| DCOLON \| SEMICOLON \| |
| | | DOT \| DDOT \| COMMA \| LPAR \| |
| | | RPAR \| LBRACE \| RBRACE \| LBRACKET \| |
| | | RBRACKET \| QUESTION \| BAR \| |
| | | CONTEXT \| INV \| DEF \| INIT DERIVE \| |
| | | LET \| ITERATE \| IN \| ATTR \| |
| | | OPER \| AND \| OR \| NOT \| XOR \| |
| | | IMPLIES \| LT \| GT \| LTEQ \| |
| | | GTEQ \| EQUAL \| NEQUAL \| PLUS \| |
| | | MINUS \| MULT \| DIV \| CARET \| |
| | | DCARET \| AT \| IF \| THEN \| |
| | | ELSE \| ENDIF \| PRE \| POST \| |
| | | BODY \| SET \| BAG \| ORDEREDSET \| |
| | | SEQUENCE \| COLLECTION \| TUPLE \| |
| | | TUPLETYPE \| TRUE \| FALSE \| DIGIT \| |
| | | INT \| REAL \| SIMBOL \| SIMPLESTRING} |
| | | ”’; |
| booleanLiteralExpCS | ::= | TRUE \| FALSE; |
| oclExp | ::= | letExpCS \| implyExpression; |
| implyExpression | ::= | logicalExpression [IMPLIES implyExpression]; |
| logicalExpression | ::= | relationalExpression [logicalOpr logicalExpression] \| |
| | | NOT logicalExpression; |
| logicalOpr | ::= | AND \| OR \| XOR; |
| relationalExpression | ::= | ifExpCS [relationalOpr relationalExpression ]; |
| relationalOpr | ::= | LT \| GT \| LTEQ \| GTEQ \| NEQUAL \| EQUAL; |

表 12: OCL 構文 3

```
ifExpCS                  ::= additionalExpression |
                             IF oclExp THEN oclExp ELSE oclExp ENDIF;
additionalExpression     ::= multipleExpression [ additionalOpr additionalExpression ];
additionalOpr            ::= PLUS | MINUS;
multipleExpression       ::= unaryExpression [ multipleOpr multipleExpression];
multipleOpr              ::= MULT | DIV;
unaryExpression          ::= MINUS dotOrMessageExpression |
                             dotOrMessageExpression;
dotOrMessageExpression   ::= primaryExpression [dotOrMessageOpr];
dotOrMessageOpr          ::= DOT simpleNameCS [isMarkedPreCS]
                                   [LPAR [argumentsCS] RPAR] [dotOrMessageOpr] |
                             DOT simpleNameCS LBRACKET argumentsCS RBRACKET
                                   isMarkedPreCS [dotOrMessageOpr] |
                             arrowExpression |
                             oclMessageExpCS;
arrowExpression          ::= ARROW ITERATE LPAR variableDeclarationCS
                                   [SEMICOLON variableDeclarationCS] BAR
                                   oclExp RPAR [dotOrMessageOpr] |
                             ARROW simpleNameCS LPAR RPAR [dotOrMessageOpr] |
                             ARROW simpleNameCS LPAR variableDeclarationCS
                                   [COMMA variableDeclarationCS] BAR
                                   oclExp RPAR [dotOrMessageOpr] |
                             ARROW simpleNameCS
                                  LPAR argumentsCS RPAR [dotOrMessageOpr];
primaryExpression        ::= literalExpCS |
                             LPAR oclExp RPAR |
                             simpleNameCS (
                                  DCOLON pathNameCS [LPAR [argumentsCS] RPAR] |
                                  LPAR [argumentsCS] RPAR [LBRACKET
                                        argumentsCS RBRACKET ][isMarkedPreCS] |
                                  ϵ );
```

表 13: OCL 構文 4

| | | |
|---|---|---|
| literalExpCS | ::= | collectionLiteralExpCS \| |
| | | tupleLiteralExpCS \| |
| | | primitiveLiteralExpCS; |
| letExpCS | ::= | LET variableDeclarationListCS IN oclExp; |
| oclMessageExpCS | ::= | (DCARET \| CARET) simpleNameCS LPAR |
| | | [oclMessageArgumentsCS] RPAR [dotOrMessageOpr] |
| variableDeclarationCS | ::= | simpleNameCS [COLON typeCS] [EQUAL oclExp]; |
| variableDeclarationCS2 | ::= | simpleNameCS [COLON typeCS]; |
| variableDeclarationListCS | ::= | variableDeclarationCS [COMMA variableDeclarationListCS]; |
| typeCS | ::= | collectionTypeCS \| tupletypeCS \| pathNameCS; |
| collectionTypeCS | ::= | collectionTypeIdentifierCS LPAR typeCS RPAR; |
| tupletypeCS | ::= | TUPLETYPE LPAR [variableDeclarationListCS] RPAR; |
| isMarkedPreCS | ::= | AT PRE; |
| argumentsCS | ::= | oclExp [COMMA argumentsCS]; |
| oclMessageArgumentsCS | ::= | oclMessageArgCS [COMMA oclMessageArgumentsCS]; |
| oclMessageArgCS | ::= | QUESTION [COLON typeCS] \| oclExp; |
| attributeDefinitionCS | ::= | ATTR variableDeclarationListCS; |
| operationDefinitionListCS | ::= | OPER operationListCS; |
| operationListCS | ::= | operationDefinitionCS [COMMA operationListCS]; |
| operationDefinitionCS | ::= | simpleNameCS LPAR [parametersCS] RPAR |
| | | COLON typeCS [EQUAL oclExp]; |
| parametersCS | ::= | variableDeclarationCS [COMMA parametersCS]; |
| SIMBOL | ::= | '&' \| '%' \| '#' \| '!' \| '$' \| '¨'; |
| DIGIT | ::= | '0' \| '1' \| '2' \| '3' \| '4' \| '5' \| '6' \| '7' \| '8' \| '9'; |
| ALPHA | ::= | 'a' \| 'b' \| 'c' \| 'd' \| 'e' \| 'f' \| 'g' \| 'h' \| 'i' \| 'j' \| |
| | | 'k' \| 'l' \| 'm' \| 'n' \| 'o' \| 'p' \| 'q' \| 'r' \| 's' \| 't' \| |
| | | 'u' \| 'v' \| 'w' \| 'x' \| 'y' \| 'z' \| 'A' \| 'B' \| 'C' \| 'D' \| |
| | | 'E' \| 'F' \| 'G' \| 'H' \| 'I' \| 'J' \| 'K' \| 'L' \| 'M' \| 'N' \| |
| | | 'O' \| 'P' \| 'Q' \| 'R' \| 'S' \| 'T' \| 'U' \| 'V' \| 'W' \| 'X' \| |
| | | 'Y' \| 'Z' \| '_'; |

表 14: OCL 構文 5

| | | |
|---|---|---|
| INT | ::= | DIGIT {DIGIT}; |
| REAL | ::= | INT DOT INT; |
| SIMPLESTRING | ::= | ALPHA {ALPHA \| DIGIT}; |
| PACKAGE | ::= | ’package’; |
| ENDPACKAGE | ::= | ’endpackage’; |
| ARROW | ::= | ’−>’; |
| COLON | ::= | ’:’; |
| DCOLON | ::= | ’::’; |
| SEMICOLON | ::= | ’;’; |
| DOT | ::= | ’.’; |
| DDOT | ::= | ’..’; |
| COMMA | ::= | ’,’; |
| LPAR | ::= | ’(’; |
| RPAR | ::= | ’)’; |
| LBRACE | ::= | ’{’; |
| RBRACE | ::= | ’}’; |
| LBRACKET | ::= | ’[’; |
| RBRACKET | ::= | ’]’; |
| QUESTION | ::= | ’?’; |
| BAR | ::= | ’\|’; |
| CONTEXT | ::= | ’context’; |
| INV | ::= | ’inv’; |
| DEF | ::= | ’def’; |
| INIT | ::= | ’init’; |
| DERIVE | ::= | ’derive’; |
| LET | ::= | ’let’; |
| ITERATE | ::= | ’iterate’; |
| IN | ::= | ’in’; |
| ATTR | ::= | ’attr’; |
| OPER | ::= | ’oper’; |
| AND | ::= | ’and’; |
| OR | ::= | ’or’; |

表 15: OCL 構文 6

| | | |
|---|---|---|
| NOT | ::= | ’not’; |
| XOR | ::= | ’xor’; |
| IMPLIES | ::= | ’implies’; |
| LT | ::= | ’<’; |
| GT | ::= | ’>’; |
| LTEQ | ::= | ’<=’; |
| GTEQ | ::= | ’>=’; |
| EQUAL | ::= | ’=’; |
| NEQUAL | ::= | ’<>’; |
| PLUS | ::= | ’+’; |
| MINUS | ::= | ’-’; |
| MULT | ::= | ’*’; |
| DIV | ::= | ’/’; |
| CARET | ::= | ’^’; |
| DCARET | ::= | ’^^’; |
| AT | ::= | ’@’; |
| IF | ::= | ’if’; |
| THEN | ::= | ’then’; |
| ELSE | ::= | ’else’; |
| ENDIF | ::= | ’endif’; |
| PRE | ::= | ’pre’; |
| POST | ::= | ’post’; |
| BODY | ::= | ’body’; |
| SET | ::= | ’Set’; |
| BAG | ::= | ’Bag’; |
| ORDEREDSET | ::= | ’OrderedSet’; |
| SEQUENCE | ::= | ’Sequence’; |
| COLLECTION | ::= | ’Collection’; |
| TUPLE | ::= | ’Tuple’; |
| TUPLETYPE | ::= | ’TupleType’; |
| TRUE | ::= | ’true’; |
| FALSE | ::= | ’false’; |