特別研究報告

題目

Java Modeling Language から Object Constraint Language へのモデル変換技法に基づいた変換手法の提案

指導教員
楠本 真二 教授

報告者
榛葉 浩章

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

Java Modeling Language から Object Constraint Language へのモデル変換技法に基づいた変換手法の提案

榛葉 浩章

内容梗概

RTE (Round-trip Engineering) は設計段階と実装段階を往復しながらソフトウェア開発を行う手法である．RTE を実現するためには設計段階のモデルと実装段階のソースコードの双方向の変換が必要である．

著者の研究グループでは OCL (Object Constraint Language) と JML (Java Modeling Language) の間の変換について研究しており，既存研究では OCL から JML への変換を構文変換によって実現している．しかし，MDA (Model Driven Architecture) ではモデル変換的な変換手法が主流である．既存研究では，構文定義と変換機構が分離されていないため再利用性が低いことが課題として挙げられる．また制約記述レベルでの RTE を実現するためには JML から OCL への変換が必要である．

本研究では制約記述レベルでの RTE 実現の支援を目標として，モデル変換技法に基づいた JML から OCL への変換手法を提案する．今回は試作型として基本演算と，OCL の collection 演算に対応するループ演算の一部の変換を実装した．変換は Eclipse のプラグイン Xtext を利用して実現した．具体的には JML の構文モデルの定義，JML から OCL への変換ルールの定義を行った．Xtext では定義した構文モデル専用のエディタを生成することが可能で，ユーザは生成されたエディタを用いることで入力時点でのソースコードの構文チェックを行うことが可能になる．また、実装した変換ツールの性能評価を行うために、在庫管理プログラムに対して実装ツールの適用実験を行った。在庫管理プログラムの各メソッドには，過去の研究で JML が付加されている。結果は人手で記述された JML のうち 78.4 %を OCL へ変換できた．また JML は基本演算による記述が大部分を占めていることなどからも，変換の有効性を確認できた．

主な用語

OCL，JML，RTE，Xtext，モデル変換

目 次

1 まえがき **1**

2 準備 **2**

2.1 RTE 2
2.2 Design by Contract 2
2.3 UML 2
2.4 OCL 3
2.5 JML 4
2.6 モデル変換 5
2.7 Xtext 5
2.8 関連研究 6

3 JML から OCL への変換手法 **8**

3.1 変換の対象とする JML 構文規則 8
3.2 変換における仮定 11
3.3 変換方法 11
3.3.1 基本演算 12
3.3.2 特殊な演算 14

4 実装 **15**

4.1 実装方針 15
4.2 JML の構文モデルの記述 16
4.3 JML から OCL への変換ルールの記述 16

5 評価実験 **18**

5.1 実験概要 18
5.2 実験対象 18
5.3 計測内容 18
5.4 実験結果 19
5.4.1 実験 1 19
5.4.2 実験 2 20
5.5 考察 20
5.5.1 実験 1 20
5.5.2 実験 2 24

6　あとがき **25**

謝辞 **26**

参考文献 **27**

表 目 次

1 基本演算に対する JML 構文 (boolean 型) 8
2 基本演算に対する JML 構文 (全ての型) 9
3 基本演算に対する JML 構文 (全ての型) 9
4 特殊演算に対する JML 構文 (全ての型) 10
5 μ 変換：基本演算 (boolean 型) 12
6 μ 変換：基本演算 (short，int，long，float，double，boolean) 13
7 μ 変換：基本演算 (String，ユーザ定義型) 13
8 μ 変換：基本演算 (short，int，long，float，double) 13
9 μ 変換：基本演算 (boolean) 13
10 μ 変換：基本演算 (String，ユーザ定義型) 13
11 μ 変換：基本演算 (keyword) 14
12 μ 変換：特殊演算 (全ての型) 15
13 μ 変換：特殊演算 (collection 型) 15
14 基本演算，\result，\old が占める割合 23

図 目 次

1 OMG のモデル 4 層 3
2 OCL による制約の記述例 4
3 JML による制約の記述例 5
4 Xtext による変換の流れ 7
5 変換可能なコード例 14
6 変換不可能なコード例 15
7 JML の構文モデルの記述例 16
8 変換例：入力として与えた JML 17
9 変換例：出力された OCL 18
10 変換が正しく行われた例 (実験 1):変換前の JML 19
11 変換が正しく行われた例 (実験 1):変換後の OCL 19
12 変換が正しく行われなかった例 (実験 1):変換前の JML 20
13 変換が正しく行われなかった例 (実験 1):変換後の OCL 20
14 変換が正しく行われた例 (実験 2)：変換前の OCL 21
15 変換が正しく行われた例 (実験 2)：変換後の JML 22
16 変換が正しく行われなかった例 (実験 2)：変換前の OCL 23
17 変換が正しく行われなかった例 (実験 2)：変換後の JML 23

1　まえがき

近年ソフトウェア開発の現場では軽量開発が使われるようになり，その重要性は増してきている．RTE (Round-trip Engineering)[1][2] は設計段階と実装段階を往復しながらソフトウェア開発を行う手法である．これを要求変更の多いソフトウェア開発に対して適用すると，開発期間の短縮や効率化をすることができる．RTE では設計段階のモデルと実装段階のソースコードで整合性がとれている必要があるため，一方の変更を他方に自動的に反映する仕組みが必要となる．そのためにはモデルとソースコードの間の双方向の変換が必要となる．

設計情報と実装情報の間の変換においては，UML (Unified Modeling Language) のクラス図と Java の間の変換 [3][4] や，UML のシーケンス図やステートマシン図のような動的な情報を表すモデルと Java の間の相互変換 [5] が研究されている．著者の研究グループでは制約記述に着目し，UML のクラス図に対して制約を記述する OCL(Object Constraint Language)[6] から Java のクラスやメソッドに対して制約を記述する JML(Java Modeling Language)[7] への変換を実現している [8][9]．しかし既存の手法では変換を構文変換で行っており，構文定義と変換ルールの定義が分離されていないため再利用性が低いことが課題としてあげられる．また JML から OCL への変換についての研究は存在しておらず，制約記述レベルでの RTE が実現できない状況となっている．

そこで本研究では制約記述レベルでの RTE の実現の支援を目的として，JML から OCL への変換をモデル変換的な手法で実現した．これにより構文モデル (メタモデル) の定義と変換ルールの定義が分離でき，実装の再利用性の向上が見込める．例として，JML から OCL 以外の言語へ変換をするときに JML の構文定義を再利用して，変換ルールの定義を新たにしなおすだけで他言語への変換が可能となる．また Xtext[10] を使用することによって，定義した構文モデルに従った JML の記述をサポートするエディタが生成されるので，ユーザは変換可能な JML を正しく入力することができる．このエディタを利用することにより，ツールそのもののユーザビリティの向上が見込める．

今回は試作型として述語の構文の中で Java を踏襲している部分である基本演算と，result や \old() などの JML 特有の特殊演算，OCL の collection 演算に対応するループ演算の変換を実装した．変換は JML の事前条件，事後条件，不変条件，述語の構文の中で OCL に変換可能なものを対象としている．

以降，2 章では研究の背景となる諸技術について述べる．3 章では JML から OCL への変換手法について説明する．4 章では作成したツールの実装について説明する．5 章では作成した変換ツールの評価実験と結果についての考察を述べる．最後に 6 章で本報告のまとめを述べる．

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 RTE

RTE (Round-trip Engineering) はソフトウェア開発手法の 1 つで設計段階と実装段階を往復しながら開発を行う手法である．要求変更の多いソフトウェア開発に適している．RTE をそのようなソフトウェア開発に適用すると開発期間の短縮や効率化をすることができる．RTE では IDE 等の RTE 開発支援ツールを用い，設計・コードのいずれかの変更の一部を他方に自動的に反映していくものが一般的である．そのためには設計段階のモデルと実装段階のソースコードで整合性を保つ必要があり，モデルとソースコードの間の双方向の変換が実現されていなければならない．

2.2 Design by Contract

Design by Contract(以降，DbC とする)[11] 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．

2.3 UML

UML は， Object Management Group(OMG) がオブジェクトモデリングのために標準化した仕様記述言語である．UML 2.0 では 13 種類の図 (ダイアグラム) を定義しており，クラス図，状態遷移図、シーケンス図などがよく使われる．UML の定義は Meta-Object Facility (MOF)[12] のメタモデルを使って行っている．UML モデルは，QVT (Queries/Views/Transformations) などの変換言語を使って Java などに自動的に変換できる．この機構を使い，クラス図からソースファイルのスケルトンを生成することもできる．OMG は UML を 4 階層のアーキテクチャで定義している (図 1).

1. MOF : M3 層に相当し，UML メタモデルを記述するための言語を定義する．

2. UML メタモデル : MOF のインスタンス．M2 層に相当し，UML モデルを記述するための言語を定義する．

3. UML モデル : UML メタモデルのインスタンス．M1 層に相当し，オブジェクトモデルを記述するための言語を定義する．

4. オブジェクトモデル : UML モデルのインスタンス．M0 層に相当し，特定のオブジェクトを表現する．

一般に呼ばれている UML は第二下層の M1 が担当する．本研究では M1 層のクラス図を対象とする．

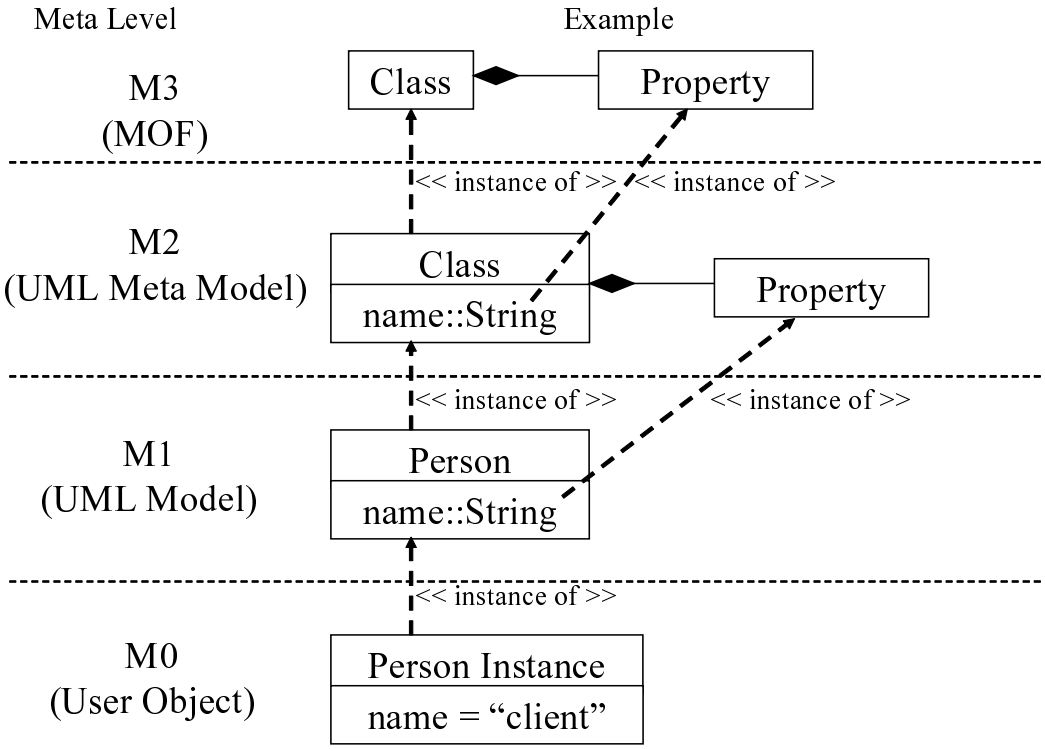

図 1: OMG のモデル 4 層

2.4 OCL

OCL は UML のクラス図に対し，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

のような性質を持つか，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 2 に OCL による制約の記述例を示す．図 2 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，および 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．

2.5 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 3 に JML による制約の記述例を示す．図 3 の例では，2.4 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．

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

図 2: OCL による制約の記述例

```
public class Account{
   int balance;
   /*@
        requires balance - val > 0;
        requires val > 0;
        ensures \result == \old(balance) - val;
        ensures balance == \old(balance) - val;
   @*/
   public int withdraw(int val){
      balance -= val;
      return balance;
   }
}
```

図 3: JML による制約の記述例

2.6 モデル変換

モデル変換はあるメタモデルに従ったモデルを入力とし，別のモデルに従うモデルやコードを出力する．入力と出力のモデルが同じ変換を内因的と呼び，入力と出力のモデルが異なる変換を外因的と呼ぶ．代表的なモデル変換として QVT[15] や ATL[16] などが挙げられる．これらは，MDA (Model Driven Architecture)[17] におけるモデル変換である．モデル変換にはモデルからモデルへの変換である Model2Model 変換 (M2M)，モデルからコードへの変換である Model2Text(M2T) が存在する．M2T 変換機能を提供するツールとして，UML2Java [18] などが挙げられる．

2.7 Xtext

Xtext は Eclipse のモデリングプロジェクトの 1 つでテキストベースの DSL (Domain Specific Language) 開発ツールである．モデルの構文定義や，構文定義に従ったモデルからコードへの変換ルールの構築などをサポートする．Xtext ではモデルの構文定義を行うことで，コード補完機能やエラー検出機能などを備えたエディタ，構文解析をするパーサ，モデルをコードに変換するジェネレータなどを生成することができる．生成されたエディタ上で入力となるテキスト型モデルを記述すると，自動的に変換ルールを用いてモデルをコードに変換

する．

変換の流れを図 4 に示す．まずユーザは生成されたエディタ上でテキスト型モデル (ソースコード) を記述する．このとき生成されたエディタには構文定義された言語のコンテンツアシストをする機能が組み込まれているため，ユーザは定義に従ったモデルを入力することができる．コンテンツアシストとは Eclipse の機能で，入力可能な候補を表示し補完する機能である．入力されたモデルはパーサによって解析され，抽象構文木として出力される．次にジェネレータはその抽象構文木の情報と変換ルールを用いて変換先の言語のコードを出力する．

2.8 関連研究

OCL から JML への変換については文献 [19] において Hamie が構文変換技法に基づいた変換法を提案している．Rodion と Alessandra らは文献 [19] を基に，文献 [20] において OCL における 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．しかし，いずれの方法も Collection 演算への対応が不十分であり，iterate 演算への対応がされていない．文献 [9] では宮澤らが未対応であったそれらの演算に対しての変換法を提案し，Eclipse プラグイン化を行っている．

RTE については Medvidoc らが文献 [1] において UML とオブジェクト指向言語の間の RTE 手法を提案している．また Sendall らは文献 [2] においてモデル間の RTE 手法を提案している．

Xtext については田中らが文献 [21] において，Xtext を用いた Business Process と SOA (Service-Oriented Architecture) モデルの間の変換方法を提案している．

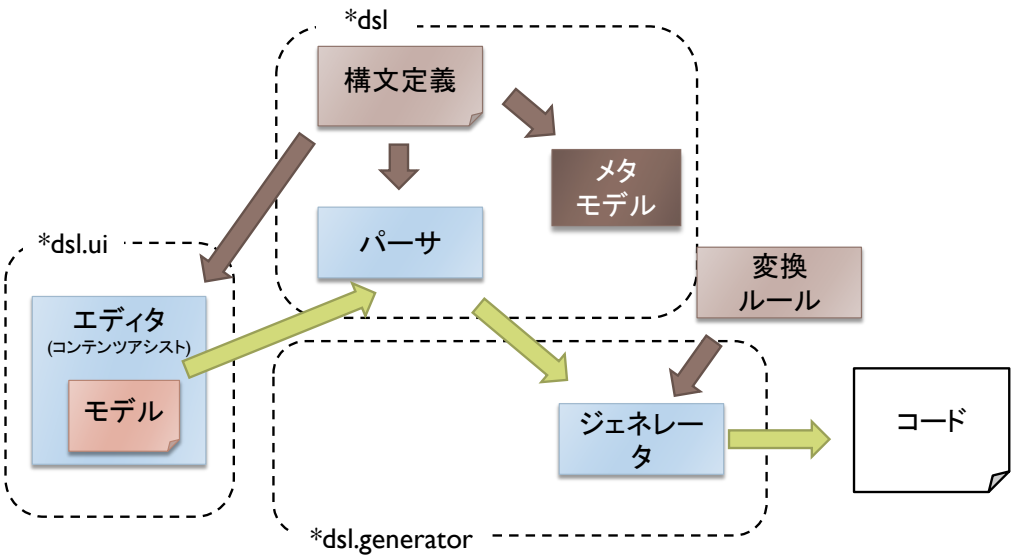

図 4: Xtext による変換の流れ

3　JML から OCL への変換手法

この章では JML から OCL への変換手法について述べる.

3.1　変換の対象とする JML 構文規則

JML から OCL への変換で対象とする範囲は事前条件，事後条件，不変条件，述語の構文である．JML Reference Manual[22] における JML の構文から対象となる範囲の構文を抽出した．対象範囲の構文規則を表 1，2，3，4 に示す．”[ ]”は省略可能であること,”[ ]...”は 0 回以上の繰り返し，”|”はその前後にある終端記号や非終端記号のいずれかを選択することを表す．また JML から OCL への変換を考慮したときに対応できない構文に関しては除外した．例として，JML では代入演算が許されているが OCL では副作用が発生する演算は存在していないので変換することはできない.

表 1: 基本演算に対する JML 構文 (boolean 型)

predicate ::= spec-expression
spec-expression-list ::= spec-expression [, spec-expression] ...
spec-expression ::= expression
expression-list ::= expression [, expression] ...
expression ::= assignment-expr
assignment-expr ::= conditional-expr
conditional-expr ::= equivalence-expr [? conditional-expr : conditional-expr]
equivalence-expr ::= implies-expr [equivalence-op implies-expr]...
equivalence-op ::= <==> | <=! =>
implies-expr ::= logical-or-expr [==> implies-non-backward-expr] |
logical-or-expr <== logical-or-expr [<== logical-or-expr] ...
implies-non-backward-expr ::= logical-or-expr [==> implies-non-backward-expr]
logical-or-expr ::= logical-and-expr ['||' logical-and-expr] ...
logical-and-expr ::= inclusive-or-expr [&& inclusive-or-expr] ...
inclusive-or-expr ::= exclusive-or-expr ['|' exclusive-or-expr] ...
exclusive-or-expr ::= and-expr [^ and-expr] ...
and-expr ::= equality-expr [& equality-expr] ...

表 2: 基本演算に対する JML 構文 (全ての型)

equality-expr ::= relational-expr [== relational-expr] ... |
relational-expr [!= relational-expr] ...

relational-expr ::= shift-expr < shift-expr | shift-expr > shift-expr |
shift-expr <= shift-expr | shift-expr >= shift-expr |
shift-expr <: shift-expr | shift-expr [instanceof type-spec]

shift-expr ::= additive-expr

additive-expr ::= mult-expr [additive-op mult-expr] ...

additive-op ::= + | -

mult-expr ::= unary-expr [mult-op unary-expr] ...

mult-op ::= * | / | %

表 3: 基本演算に対する JML 構文 (全ての型)

unary-expr ::= (type-spec) unary-expr | + unary-expr |
- unary-expr | unary-expr-not-plus-minus

unary-expr-not-plus-minus ::= ! unary-expr | (built-in-type) unary-expr |
(reference-type) unary-expr-not-plus-minus | postfix-expr

postfix-expr ::= primary-expr [primary-suffix] ...

primary-suffix ::= . ident | ([expression-list])

primary-expr ::= ident | constant | true | false | this |
null | (expression) | jml-primary

constant ::= java-literal

java-literal ::= integer-literal | floating-point-literal | boolean-literal |
string-literal | null-literal

表 4: 特殊演算に対する JML 構文 (全ての型)

jml-primary ::= result-expression | old-expression | not-modified-expression |
fresh-expression| spec-quantified-expr

result-expression ::= \result

old-expression ::= \old (spec-expression [, ident])

not-modified-expression ::= \not_modified (store-ref-list)

fresh-expression ::= \fresh (spec-expression-list)

spec-quantified-expr ::= (quantifier quantified-var-decls ; [[predicate] ;] spec-expression)

quantifier ::= \forall | \exists | \sum

quantified-var-decls ::= type-spec quantified-var-declarator
[, quantified-var-declarator] ...

quantified-var-declarator ::= ident

store-ref-list ::= store-ref [, store-ref] ...

store-ref ::= store-ref-expression

store-ref-expression ::= store-ref-name [store-ref-name-suffix] ...

store-ref-name ::= ident | this

store-ref-name-suffix ::= . ident | . *

ident ::= letter [letter-or-digit] ...

letter ::= _, $, a through z, or A through Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter-or-digit ::= letter | digit

type-spec ::= type

type ::= reference-type | built-in-type reference-type ::= name

name ::= ident [. ident] ...

built-in-type ::= boolean | byte | short | int| long | float | double

3.2 変換における仮定

前述したようにJMLはJavaのソースコード中に記述するのでJavaのメソッドを呼び出すことができる．しかしJavaの標準クラスライブラリには多数のクラスやメソッドが存在していて全てのクラスやメソッドに対応するのは現実的でない．そこで今回は試作型としてJML Reference Manualに記載されている述語 (predicate) の部分の演算のみを対象とする．変換にあたり以下の条件を仮定し，制限を設けた．またJMLの型はJavaの型と一致する．

1. 型はプリミティブ型以外にもユーザ定義型（オブジェクト名）を自由に記述できるようにし，メソッド名も自由に記述できるようにして制限は与えない．

2. 対象 (部分）式に型が定義でき，変換時にその型情報が利用できるものとする．JMLの型としてboolean，int，byte，short，long，float，double，String，ユーザ定義型 (オブジェクト) を仮定する．JMLの型とOCLの型の対応はbyte，short，int，longはInteger型，booleanはBoolean型，float，doubleはReal型，StringはString型にそれぞれ対応する．JMLのchar型についてはOCLに対応する型が存在しないので今回は変換対象外とする．

3. 表1，2，3，4からは型的に不整合な式が導出されうるが，そのような型不整合な式は対象としない．例外としてbyte，short，int，long，とfloat，doubleの組み合わせは認める．これはOCLでそれぞれの型に対応するInteger型とReal型の間での演算が許されているためである．

4. 今回はJavaのcollectionや標準クラスライブラリは対象外とするのでユーザ定義として扱われる. ただしStringクラスのequalsメソッド，およびcollectionに対してJML特有のループ演算をしているものに関しては変換の対象に含める．

5. nullはOCLではUndefinedが一部対応していて，Javaでの変数がnullかどうかという演算はOCLではoclIsUndefined( )という演算が対応する．しかしnullという値に直接対応する表現は存在しない．よってnullを含む演算でoclUndefined( )に変換できるものは変換し，それ以外のnullについては変換対象外としてエラーを出力する．

3.3 変換方法

ここでは表1，2，3，4にあげた構文のうち主要な構文に対して変換法を与える．JMLからOCLへの変換関数をμで与える．以下では型boolean，byte，short，int，long，float，double，String，ユーザー定義クラスを持つ部分式をそれぞれbo_m，by_m，sh_m，i_m，l_m，

表 5: μ 変換：基本演算 (boolean 型)

$\mu(bo_1?bo_2:bo_3)$	=	if $\mu(bo_1)$ then $\mu(bo_2)$ else $\mu(bo_3)$ endif
$\mu(bo_1<==>bo_2)$	=	$\mu(bo_1)$= $\mu(bo_2)$
$\mu(bo_1<=!=>bo_2)$	=	$\mu(bo_1)$ <> $\mu(bo_2)$
$\mu(bo_1==>bo_2)$	=	$\mu(bo_1)$ implies $\mu(bo_2)$
$\mu(bo_1<==bo_2\ )$	=	$\mu(bo_2)$ implies $\mu(bo_1)$
$\mu(bo_1\&\&bo_2\ )$	=	$\mu(bo_1)$ and $\mu(bo_2)$
$\mu(bo_1\|\|bo_2\ )$	=	$\mu(bo_1)$ or $\mu(bo_2)$
$\mu(bo_1\|bo_2)$	=	$\mu(bo_1)$ or $\mu(bo_2)$
$\mu(bo_1\ \hat{}\ b_2\ )$	=	$\mu(bo_1$ xor $\mu(bo_2)$
$\mu(bo_1\&\ bo_2\ )$	=	$\mu(bo_1)$ and $\mu(bo_2)$

f_m，d_m，st_m，u_m で表す (m = 1，2，3，. . .)．また，任意の型を持つ部分式を同様に a_m で表す.

3.3.1 基本演算

表 1 の全ての部分式には型として boolean しかあてはまらないため，その場合のみを考えれば十分で，変換は容易である (表 5)．<== に関しては両辺の式の順序が逆転するので注意が必要である．

表 2 の部分式には型として全ての型が考えられる．まず byte，short，int，long，float，double，boolean の場合を考えると，いずれの場合も表 6 の変換となる．次にそれ以外の型の場合を考えると=，<>，<:，instanceof の場合の演算のみが可能で，それ以外の演算子の場合は型不整合とみなしエラーメッセージを出力して変換を行わない．型が String の場合は+演算が追加される (表 7)．JML における String の+演算に対応する OCL の表現は.concat 演算となる．

表 3 の部分式も型として全ての型が考えられる．byte，short，int，long，float，double の場合を考えると演算子+，-，キャストの演算が可能である．(表 8) 次に boolean について考えるとキャスト，!の演算が可能である (表 9)．String 型とユーザ定義型についてはキャスト，メソッド呼び出しが可能である (表 10)．JML における String 型の比較は，equals メソッドを用いると値同士の比較となり，== 演算を用いるとインスタンス同士の比較となる．一方 OCL では任意の型において=演算で値の比較が可能である．よってメソッド呼び出しについては equals メソッドを呼び出している場合のみ=演算へと変換した．また Java の keyword や変数名, メソッド名 (ident) などは表 11 のように変換する．

表 6: μ 変換：基本演算 (short，int，long，float，double，boolean)

$\mu(a_1==a_2)$	=	$\mu(a_1)= \mu(a_2)$
$\mu(a_1!=a_2)$	=	$\mu(a_1)<> \mu(a_2)$
$\mu(a_1<a_2)$	=	$\mu(a_1)< \mu(a_2)$
$\mu(a_1>a_2)$	=	$\mu(a_1)> \mu(a_2)$
$\mu(a_1<=a_2)$	=	$\mu(a_1)<= \mu(a_2)$
$\mu(a_1>=a_2)$	=	$\mu(a_1)>= \mu(a_2)$
$\mu(a_1<:a_2)$	=	$\mu(a_1)$.oclIsKindOf($\mu(a_2)$)
$\mu(a_1$ instanceof a_2)	=	$\mu(a_1)$.oclIsTypeOf($\mu(a_2)$)
$\mu(a_1+a_2)$	=	$\mu(a_1)+ \mu(a_2)$
$\mu(a_1-a_2)$	=	$\mu(a_1)- \mu(a_2)$
$\mu(a_1*a_2)$	=	$\mu(a_1)* \mu(a_2)$
$\mu(a_1/a_2)$	=	$\mu(a_1)/ \mu(a_2)$
$\mu(a_1\%a_2)$	=	$\mu(a_1)$.mod($\mu(a_2)$)

表 7: μ 変換：基本演算 (String，ユーザ定義型)

$\mu(a_1==a_2)$	=	$\mu(a_1)= \mu(a_2)$
$\mu(a_1!=a_2)$	=	$\mu(a_1)<> \mu(a_2)$
$\mu(a_1<:a_2)$	=	$\mu(a_1)$.oclIsKindOf($\mu(a_2)$)
$\mu(a_1$ instanceof $a_2)$	=	$\mu(a_1)$.oclIsTypeOf($\mu(a_2)$)
(String) $\mu(s_1+s_2)$	=	$\mu(s_1)$.concat($\mu(s_2)$)

表 8: μ 変換：基本演算 (short，int，long，float，double)

$\mu(+a_1)$	=	$+\mu(a_1)= \mu(a_2)$
$\mu(-a_1)$	=	$-\mu(a_1)<> \mu(a_2)$
$\mu((\text{type})a_1$)	=	$\mu(a_1)$.oclAsType(μ(type))

表 9: μ 変換：基本演算 (boolean)

$\mu((\text{type})b_1)$	=	$\mu(bo_1)$.oclAsType(μ(type))
$\mu(!b_1)$	=	$not\mu(bo_1)$

表 10: μ 変換：基本演算 (String，ユーザ定義型)

$\mu((\text{type})a_1)$	=	$\mu(a_1)$.oclAsType(μ(type))
$\mu(a_1.a_2(\ ))$	=	$\mu(a_1).\mu(a_2)(\ )$

表 11: μ 変換：基本演算 (keyword)

(変数，メソッド名)μ(ident)	=	ident
μ(true)	=	true
μ(false)	=	false
μ(this)	=	self
μ(bollean)	=	Boolean
μ(byte)	=	Integer
μ(short)	=	Integer
μ(int)	=	Integer
μ(long)	=	Integer
μ(float)	=	Real
μ(double)	=	Real
(数値) μ(123)	=	123
μ(”str”)	=	’str’

3.3.2 特殊な演算

表 4 の部分式も型として全ての場合が考えられる．spec-quantified-expr 以外はすべての型で同じように変換できる (表 12).

spec-quantified-expr については JML の forall や exist，sum 演算は for ループのような幅広いループ演算であるのに対して，OCL の forAll や exists，sum 演算は 1 つのコレクションの要素全てに対する演算しかできない. よって全ての場合を変換することはできず，また JML で複数のコレクションに対して 1 度に演算を行っている場合などは高度な意味解析をして OCL 側では 2 つの演算に分割したり，条件を移動させるという操作が必要になってしまう.. 今回は 1 つのコレクションに対してその要素がコレクションに含まれているか調べてから演算をしている，という限定的な形のときのみ OCL に変換した（表 13）．変換可能な場合と不可能な場合のコード例を図 5，6 に示す．また表 13 において T は任意の型，e は変数名，TList は T 型のコレクション，expr は任意の論理式とする.

```
(\forall Item i;ItemList.contains(i);i.price > 0)
```

図 5: 変換可能なコード例

```
(\forall Item i,j;ItemList.contains(i) || BookList.contains(j);
                                        i.price > 0 && j.price > 0)
(\forall Item i;ItemList.contains(i) && i.price > 0;
                                                  i.price < 1000)
```

図 6: 変換不可能なコード例

表 12: μ 変換：特殊演算 (全ての型)

μ(\result)	=	result
μ(\old(a_1))	=	$\mu(a_1)$@pre
μ(\not_modified(a_1))	=	$\mu(a_1) = \mu(a_1)$@pre
μ(\fresh(a_1))	=	$\mu(a_1)$.oclIsNew()

表 13: μ 変換：特殊演算 (collection 型)

μ(\forall T e;TList.contains(e);expr)	=	μ(TList)->forAll(μ(e): μ(T) | μ(expr))
μ(\exists T e;TList.contains(e);expr)	=	μ(TList)->exist(μ(e): μ(T) | μ(expr))
μ(\sum T e;TList.contains(e);expr)	=	μ(TList)->sum(μ(e): μ(T) | μ(expr))

4 実装

ここでは実装について詳細に述べる．

4.1 実装方針

まず Xtext を用いて Java スケルトンコードと JML が記述できる構文モデルを定義する．このとき変換がしやすくなるように元の構文から意味が変わらない程度に変更を加えた．変換ルールの定義では，まず Java スケルトンコードの部分を解析して，クラスの中の変数やメソッドの型情報をマップに保持しておく．このマップの情報は変換ルールの中に保持される．このようにすることで変数名やメソッド名をキーとして型情報を得ることができる．実際の変換の時は型の情報が必要な部分はそれを引き出しながら 3 章で定義した形に変換していく．

```
JmlPrimary :
    resultexpression = ResultExpression
    | oldexpression = OldExpression
    | notmodifiedexpression = NotModifiedExpression
    | freshexpression = FreshExpression
    | specquantifiedexpr = SpecQuantifiedExpr
;
ResultExpression :
    result = '\\result'
;
OldExpression :
    old = '\\old' parel = '(' specexpression = SpecExpression (comma = ',' ident = Ident)? parer = ')'
;
NotModifiedExpression :
     notmodified = '\\not_modified' parel = '(' storereflist = StoreRefList parer = ')'
;
FreshExpression :
    fresh =  '\\fresh' parel = '(' specexpressionlist = SpecExpressionList parer = ')'
;
```

図 7: JML の構文モデルの記述例

4.2 JML の構文モデルの記述

JML と Java のスケルトンコードが記述できるモデルの構文定義を行った．最終的に記述した構文規則の数は 90 となった．Java スケルトンの部分はクラス宣言，修飾子，クラスのフィールドとメソッドの宣言のみの構文モデルを定義をした．JML の部分に関しては JML の Reference Manual で定義されている事前条件，事後条件，不変条件，述語の部分の構文と演算のみを扱った．本来 JML は Java ソースコード中にコメントという形で /@ と @/ の間に記述するものである．しかし Xtext によって生成されたエディタでは，/ 以降の文字列はコメントとして判断されてしまうため，今回は JML をコメントの形で記述しない構文モデルとした．この部分に関しては今後の拡張で対応していく必要がある．また，その中でも OCL で表現できないと判断したものについては除外し，変換できる構文モデルのみを記述した．これによって，自動生成されるエディタのコンテンツアシスト機能では変換できる JML 式のみが記述できるようになり，ユーザビリティが向上する．実際の記述は EBNF をベースにした言語で構文規則を記述する．記述例を図 7 に示す．

4.3 JML から OCL への変換ルールの記述

JML から OCL への変換では，基本演算に関してはほぼ全て 1 対 1 の対応がとれるので JML の演算子を OCL への演算子へと変換することで変換が実現できる．要素の順序や ( ) をつけるなど多少の変更が必要なものも存在する．

+演算の変換に関しては型の情報が必要となる．変換をする前に Java のスケルトンコー

```
class Account {
    int num;
    String str;
    Boolean b;

    invariant num > 0;

    requires num1 > 0 && num2 > 0;
    ensures  \result == num1 + num2;
    int sum(int num1,int num2){

    }

    ensures \result.equals(str+str2);
    String output(String str2){

    }
}
```

図 8: 変換例：入力として与えた JML

ドを解析して，クラス，フィールド，メソッド，引数，それぞれの名前と型情報をマップに保持しておく．変換の際にマップに保存してある情報を用いて型の判別を行う．JML における+演算の場合を例に説明すると以下の 3 つのパターンが考えられる．

1. 2 つの被演算子の型が数値型の場合には+演算子をそのまま出力する
2. 2 つの被演算子の型が文字列型の場合には文字列の連結を行う concat 演算に変換する
3. 2 つの被演算子の型が異なる場合はエラーを出力する

forall などのループ演算に関しては今回は限定的な形の場合のみの変換を実装する．その条件はループ演算中での変数定義が 1 種類のみで，1 つ目の条件節は collection 変数.contains() という条件式の形のときのみである．それ以外の場合は変換エラー出力をする．最終的な記述した変換ルールの数は 45 となった．変換例を図 8，9 に示す．

```
context Account
inv: num > 0

context Account::sum(num1:Integer,num2:Integer)::Integer
pre: num1 > 0 and num2 > 0
post: result = num1 + num2

context Account::output(str2:String)::String
post: result = str.concat(str2)
```

図 9: 変換例：出力された OCL

5 評価実験

5.1 実験概要

本研究では作成したツールを小規模なプロジェクトに対して適用した．実験 1 は実験対象プログラムに記述された JML を作成したツールで変換して，どの程度意味的に正しく変換が行えるかどうかを検証する．実験 2 では RTE 適用可能性を確かめるために作成したツールで JML から OCL へ変換した結果を既存の変換ツールで再び JML へと変換し，どの程度元の表現を再現できるか検証する．OCL から JML への変換には文献 [23] のツールを用いた．OCL から JML への変換ツールには入力としてテキストで表されたクラス図の情報と OCL が必要となる．著者の作成したツールにおいては変換時にクラスの情報は出力されないので，Java のソースコードを元にして情報を付加して実験を行った．

5.2 実験対象

実験対象としては在庫管理プログラムを使用する．これは著者の研究グループで過去の研究において実験に使用された Java のプログラムである．これには過去の研究で既に OCL, JML ともに付加されている．変換対象は 6 クラス 39 メソッドで事前条件，事後条件，不変条件，が記述されている．今回は対象範囲を事前条件，事後条件，不変条件に絞っているため，そのような条件に記述されている式のみを対象とする．

5.3 計測内容

作成したツールで変換した結果を評価する指標として以下の 2 点を計測する．

変換率 JML から OCL への変換が成功し，意味的に正しく変換できた数．

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

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

5.4 実験結果

5.4.1 実験 1

在庫管理プログラムに記述されている事前条件，事後条件，不変条件の式の数は全部で 132 個で，それらについて変換を適用した．意味的に正しく変換できた式の数は 103 個で変換率は 78.0% となった．以下に正しく変換できた場合とできなかった場合の具体例を示す．正しく変換できなかった場合は forall などのループ演算で変数が複数宣言されていたり条件が複雑な場合，\type 演算，\typeof 演算，String 型と数値型の+演算などがあげられる．実験結果の例を図 10，11，12，13 に示す.

```
requires name != null && !name.equals("");
requires num > 0;
ensures (\exists Item i; itemList.contains(i);
    i.getName().equals(name)&&\result == \old(num-i.getAmount()));
int shippingItem(String name, int num) {

}
```

図 10: 変換が正しく行われた例 (実験 1):変換前の JML

```
context ContainerItem::shippingItem(name:String,num:Integer)::Integer
pre: not name.oclIsUndefined() and not(name = '')
pre: num > 0
post: itemList->exists(i:Item|i.getName() = name
            and result = (num- i.getAmount())@pre)
```

図 11: 変換が正しく行われた例 (実験 1):変換後の OCL

```
ensures \result.matches("ContainerID." + containerID + "CarryingDate : "
    + carryingDate + "\n{1}");
ensures (\forall Item i; itemList.contains(i);\result.matches("^\t"
    + i + "\n{1}"));
String toString() {

}
```

図 12: 変換が正しく行われなかった例 (実験 1):変換前の JML

```
context ContainerItem::toString()::String
post: result.matches('ContainerID.'[type error][type error]
        [type error][type error])
post: itemList->forAll(i:Item|result.matches('^\t'[type error][type error]))
```

図 13: 変換が正しく行われなかった例 (実験 1):変換後の OCL

5.4.2 実験 2

実験 1 で JML から OCL への変換で意味的に正しく変換できた式の数は 103 個である．そのうち OCL から JML の変換で意味的に正しく変換できた式の数は 98 個で逆変換率は 95.1% となった．JML から OCL への変換で正しく変換できなかったものを含めると 74.1% となる．ただし OCL から JML への変換において，入力した OCL は全て変換可能と認識された．しかし変換結果には意味的に正しくないものも含まれていた．正しく変換できなかったものとしては，キャストの演算，not 演算，\old 演算があげられる．変換結果の例を図 14, 15 に示す．

5.5 考察

5.5.1 実験 1

実験結果の変換率を見てみると変換率は 78.0% となっている．今回は試作型として基本演算と一部の特殊演算の変換を実現したので全てを変換することはできなかった．しかし 78.0% という変換率は JML 記述の多くを基本演算と一部の特殊演算が占めていることを表している．

在庫管理プログラムに記述されている JML 式では，基本演算と \result 演算，\old 演算が多くを占めている．これらの演算が，ほかのプログラムにおいても大部分を占めているのであれば，変換の有効性が確かめられると考えた．そこで他のプログラムについても調査した．調査対象は JML 公式サイトの sample プログラムで，org/jmlspecs/samples 以下の 17

```
entity ReceptionDesk{
    Name : LinkedList
    Storage : Storage

    post: result = storage
    op getStorage() : Storage

    post: requestList->forAll(r:Request|r.getAmount() > 0)
    op deliverOrder() : List

    pre: not(c.oclIsUndefined())
    post: storage.getContainerItemList().contains(c)
    op receptNewContainerItem(c : ContainerItem)

    pre: not(r.oclIsUndefined())
    post: storage.checkStockSatisfied(r) implies result
    op receptRequest(r : Request)
}
```

図 14: 変換が正しく行われた例 (実験 2)：変換前の OCL

個のパッケージについて調査した．結果は表 14 のようになった．表 14 において JML の式の数は ensures や requires の後ろに書かれている述語を 1 つの式として数えた．基本演算，\result 演算，\old 演算で記述されている式の数についても同様に，ensures や requires の後ろに書かれている述語を 1 つの式として数えた．占有率は JML の式の数のうち，どれだけ基本演算，\result 演算，\old 演算で記述されている式が存在するかである．結果として，17 個のパッケージのうち 16 個が占有率が 75% 以上となった．これらのことから，基本演算，\result 演算，\old 演算に対応していれば，変換の大部分をサポートできると考えられる．作成したツールはこれらの演算に対応していることから，その有効性が確認できたといえる．

次に変換できなかったものについて述べる．まず JML の \type と \typeof について述べる．これらは引数として与えた変数の型名やリストなどの要素の型名を返す演算である．OCL では，型名を求める演算は存在していないため直接的に対応することができない．これには JML の解析の時点で引数の型の情報を求めておき、OCL には直に型名を出力するという対応が考えられる．しかしこれでは型を求める演算という情報が失われてしまう問題がある．次に String 型と数値型の+演算について述べる．JML では+で変数をつないで異なる型の変数の内容を結合することができる．一方 OCL では基本的に同じ型同士の演算しか規定していないため表現することはできない．これは変換メソッドを挿入することで対応可能である．また，forall などのループ演算については限定的な形の場合のみ変換しているため対応できていないものが存在している．現状は対応していないが，変数が 2 つ以上含まれているようなコレクション演算の場合は，複数の文に分割したり，条件が複雑な場合は条件を移動

```
package ;
public class ReceptionDesk {
/*@
@*/
    private LinkedList Name;

    public LinkedList getName() {
        return Name;
    }

    public void setName(LinkedList Name) {
        this.Name = Name;
    }

    private Storage Storage;

    public Storage getStorage() {
        return Storage;
    }

    public void setStorage(Storage Storage) {
        this.Storage = Storage;
    }

    /*@
    ensures ((\result)==storage);
    @*/
    public Storage GetStorage() {
        return null;
    }
    /*@
    ensures (\forall Request r;(requestList).has(r);(((r).getAmount())>0));

    @*/
    public List DeliverOrder() {
        return null;
    }
    /*@
    requires !(((c) == null));
    ensures ((storage).getContainerItemList).contains(c);
    @*/
    public void ReceptNewContainerItem(ContainerItem c) {
    }
    /*@
    requires !(((r) == null));
    ensures ((storage).checkStockSatisfied(r)==>\result);
    @*/
    public void ReceptRequest(Request r) {
    }
}
```

図 15: 変換が正しく行われた例 (実験 2)：変換後の JML

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

図 16: 変換が正しく行われなかった例 (実験 2) : 変換前の OCL

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

図 17: 変換が正しく行われなかった例 (実験 2) : 変換後の JML

表 14: 基本演算，\result，\old が占める割合

パッケージ名	JML の式の数	基本演算，\result，\old で記述されている式の数	占有率
dbc	64	62	96.8%
digraph	130	117	90.0%
dirobserver	21	19	90.4%
jmlkluwer	45	29	64.4%
jmlrefman	41	37	90.2%
jmltutorial	21	16	76.1%
iterator	18	18	100%
list	49	44	89.7%
list2	39	31	81.5%
node	20	16	80.0%
node2	18	14	77.7%
misc	58	46	79.3%
prelimdesign	135	119	88.4%
reader	16	14	87.5%
sets	74	62	83.7%
stacks	83	65	78.3%
table	54	43	79.6%

させる必要が生じる．これらに対応するには高度な意味解析が必要となるが，今後の拡張で対応していくことを予定している．

5.5.2 実験 2

実験結果の逆変換率は JML から OCL で意味的に正しく変換できたものについては 95.1% となった．変換結果には正しくない表現になっているものが存在したが，これは逆変換に使用したツールが試作段階で，デバッグが完了していなかったり実装の漏れが存在するためである. 入力として与えた OCL は変換可能と認識されたので，品質に問題はないが，変換部分が不完全なのでこのような結果になったと考えられる．これらのことから本ツールで生成された OCL そのものの品質に問題がないことが確認できた.

変換できなかったものについては，変換は 1 対 1 で対応しているので実装の際のミスである．よって追加実装やデバッグを行うことによって正しい変換を実現できる.

しかし RTE 実現を考えると，1 つの表現を変換して，その逆変換をしたときに形式的にも同じ表現に戻ることが理想的である．逆変換側の変換方法を加味した，形式的にも同じ表現に戻るような変換の実現が今後の課題としてあげられる.

6　あとがき

本研究では近年重要性が増してきている RTE の支援を目標として，JML から OCL への変換手法の提案, ツールの実装, および小規模なプロジェクトに対して評価実験を行った.

評価実験については，JML から OCL への変換では 78.0% の割合で意味的に正しく変換することができた．JML 記述の多くを基本演算，\result 演算，\old 演算が占めていて，それらの演算に関してほぼ変換が可能であったことから実装したツールの有効性を確認することができた．OCL から JML の変換では，JML から OCL へ正しく変換できたもののうち、逆変換で JML へ変換可能と認識された割合は 100% となり，本研究で実装したツールによって生成された OCL の品質に問題がないことを確認できた.

今後の方針として，JML が付加された Java から OCL が付加された UML への変換を意識して,Java の Collection への対応や，OCL から JML への変換で 1 つの表現が複数表現になるものの逆変換への対応などが考えられる．また RTE 実現を意識して，OCL と JML の間の変換を繰り返し適用しても同じ表現に戻るように双方向の変換を調整していくことも考えられる.

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 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して， 熱心かつ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.

本研究に関して，厳しくも的確なご助言を頂きました 井垣 宏 特任准教授 に深く感謝申し上げます.

本研究を行うにあたり，日常の議論の中でご助言を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究の全過程を通し多大なるご助言や励ましを頂き，時には苦労を共にして頂いた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博士前期課程１年の 花田 健太郎 氏に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等で多くの知識や示唆を頂きました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] N. Medvidovic, A. Egyed, and D.S. Rosenblum. “Round-Trip Software Engineering Using UML:From Architecture to Design and Back”. In *Proceedings of the 2nd Workshop on Object-Oriented Reengineering*, pp. 1–8. ACM Press, 1999.

[2] S. Sendall and J. Küster. “Taming Model Round-Trip Engineering”. In *Proceedings of Workshop ’Best Practices for Model-Driven Software Development*, pp. 1–13, 2004.

[3] G.Engels, R.Hüking, 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.

[4] W.Harrison, C.Barton, and M.Raghavachari. “Mapping UML designs to Java”. In *Proc. of the 15th ACM SIGPLAN conference on Objected-oriented programming,systems,languages,and applications*, pp. 177–187, 2000.

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

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

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

[8] 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.

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

[10] Eclipse Foundation. “Xtext-Language Development Framwork”. http://www.eclipse.org/Xtext.

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

[12] Object Management Group. “Meta Object Facility(MOF)2.0 Core Specification”, 2004. http://www.omg.org/docs/ptc/03-10-04.pdf.

[13] Y.Cheon and G.T. Leavens. “A simple and practical approach to unit testing:The JML and JUnit way”. *ECOOP 2002 Object-Oriented Programming*, pp. 1789–1901, 2006.

[14] 中島震. “プログラム簡易検証ツール ESCJava2”. コンピュータソフトウェア, Vol. 24, No. 2, pp. 22–27, 2007.

[15] 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/.

[16] 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.

[17] 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.

[18] “uml2java”. http://uml2java.sourceforge.net/.

[19] A Hamie. “Translating the Object Constraint Language into the Java Modeling Language”. In *Proc. of the 2004 ACM symposium on Applied computing*, pp. 1531–1535, 2004.

[20] M. Rodion and R. Alessandra. “Implementing an OCL to JML translation tool”. 信学技報, Vol. 106, No. 426, pp. 13–17, 2006.

[21] T. Akira and T. Osamu. “Experimental transformations between Business Process and SOA models”. In *Proceedings of International Workshop on Informatics*, pp. 105–113, 2011.

[22] “JML Reference Manual”, 2011. http://www.eecs.ucf.edu/ leavens/JML//OldReleases/jmlrefman.pdf.

[23] 花田健太郎, 岡野浩三, 楠本真二. “OCL から JML への DSL を用いた変換ツールの試作型の実装”. ウィンターワークショップ 2012・イン・琵琶湖 論文集, No. 1, pp. 105–106, 2012.