修士学位論文

題目

ラウンドトリップエンジニアリングのための
ソフトウェア制約記述言語双方向変換

指導教員
楠本　真二　教授

報告者
花田　健太郎

平成 25 年 2 月 5 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 24 年度 修士学位論文

ラウンドトリップエンジニアリングのための
ソフトウェア制約記述言語双方向変換

花田　健太郎

内容梗概

近年 MDA(Model Driven Architecture) 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浴びており，それに伴い OCL(Object Constraint Language) から JML(JavaModeling Language) への変換技術に関する研究も行われている．我々の研究グループでも，過去の研究において OCL からの JML，および JML から OCL への変換手法の提案を行っている．

また，ラウンドトリップエンジニアリング (RTE) というソフトウェア開発手法に関する研究が近年注目を浴びている．RTE とは，要件定義，設計，コーディングを行き来しながら，それぞれの中間成果物を洗練させていく開発手法であり，クラス図のような静的モデルに着目した研究や．シーケンス図やステートチャート図のような動的モデルに着目した RTE 支援に関する研究は存在するが，OCL や JML のような仕様記述の側面に着目した研究は存在しなかった．

本研究では，仕様記述言語間での RTE の実現を目標として，OCL・JML 間の変換規則の改良，RTE を考慮した変換方法の設計・実装，複数のプロジェクトに対する適用実験を行った．既存研究の変換規則では，双方向変換を実現するには不十分な点が存在したため，それらを改善するための改良を行った．また，RTE では設計と実装を行き来する際に，それぞれ編集が加えられた箇所を相互に反映し合う．この際に，編集が加えられた箇所以外にも変換規則を適用すると，意味自体は変わらないが表現方法の異なる式が生成される可能性がある．これによって，開発者に何度も設計と実装を理解させ直させるという本来不要な作業を生んでしまう恐れがあるため，本実装では変更箇所以外の情報は元の状態を保つための手法の設計と実装を行った．その後，評価実験として，研究グループ内で JML を記述した 2 つのプロジェクトと，元々JML が付加されていた 7 つのオープンソースのプロジェクトをツールに適用し，JML から OCL への変換の成功率，逆変換結果の意味的な一致率，及び構文的な一致率を計測した．意味的な一致とは，双方向変換後の JML 式を検査ツールなどにかけた結果が，双方向変換前の JML と同様である式を指し，構文的な一致とは，括弧やスペースの数の違いなどを除いて，双方向変換の前後で完全に一致しているものを指す．実験の結果から，JML から OCL への変換は 93.5%の割合で成功した．また，その逆変換の構文的な一致率は 46.5%にとどまったものの，意味的な一致率は 93.5%であり，実用性は十分に示された．また，変換に要する時間は数秒であり，実用的な範囲で変換が行えることが確認できた．

主な用語

Object Constraint Language, Java Modeling Language, Round Trip Engineering, Model Translation, Model Driven Architecture, Design by Contract

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 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[4] 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)[5] から JML(Java Modeling Language)[6] への変換技術に関する研究も行われている [7, 8, 9, 10]．研究グループでは，過去に OCL から JML への変換ツールを Eclipse プラグインとして実装している [10, 11, 12]．OCL は UML 記述に対し，詳細な性質記述を付加するために設計された言語で，OMG(Object Management Group)[13] によって標準化されている．また，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制約記述を行う JML が提案されている．JML，OCL はともに Design by Contract[14] 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．

また，Round Trip Engineering (RTE) というソフトウェア開発手法に関する研究が近年注目を浴びている [15, 16, 17]．RTE とは，要件定義，設計，コーディングを行き来しながら，それぞれの中間成果物を洗練させていく開発手法である．例えば，MVC パターンに基づいたアプリケーションを対象とし，シーケンス図，状態遷移図とソースコード間の RTE を支援するツールなどが存在する [18]．この既存研究のように，ソフトウェアの動的な側面に着目した研究や，クラス図のような静的モデルに着目した研究は存在するが，OCL や JML のような仕様記述に着目した研究は存在しない．

研究グループでは，ソフトウェアの仕様記述，特に OCL・JML の観点から RTE 支援を目的とした OCL・JML 間の双方向変換手法を提案している [19, 20, 21]．OCL・JML 間の双方向変換を利用することで，実装と設計を行き来しながら仕様記述を洗練していくことが可能になる．また，双方向変換で実装・設計間での仕様記述のズレを無くすことにより，UML や OCL をシステムの仕様を理解するためのドキュメントとして用いることが可能になる，といった利点も生まれる．これまでの研究では，OCL から JML への変換，及び JML から OCL への変換をそれぞれ独立して実装した．そして，それらの 2 つの実装を組み合わせることによって OCL・JML 間の双方向変換の実現を試みた．この 2 つの実装は，単方向の変換に関しては問題なく変換できたが，それぞれの実装が双方向変換を考慮して実装されたものではなかったため，双方向の変換を実現する上で，いくつかの問題点が浮上した．

本研究では，双方向変換を行うにあたり問題となった点を踏まえた変換規則の定義，実装，及び評価実験を行った．評価実験には，研究グループ内で JML を記述した 2 つのプロジェクトと，7 つのオープンソースのプロジェクトを用いた．それらのプロジェクトをツールに適用し，JML から OCL への変換の成功率，逆変換結果の意味的一致率，及び構文的一致率を計測した．意味的一致とは，双方向変換の前後で式の形は異なるが，検査ツールなどを利用した際の実行結果が等価なものを指している．また構文的一致率とは，双方向変換の前後で括弧の数やスペースの数の違いなどを除いて完全に一致している式を指している．実験結果から，JML から OCL への変換は 93.5%の割合で成功したことを確認した．その逆変換の意味的な一致率は 93.5%であった．逆変換結果の式が，元の式と構

文的に一致した割合を計測したところ，プロジェクト全体で 46.5%が構文的に一致していることを確認した．構文的一致率は約半分にとどまってはいるが，意味的にはほとんどの式が元の式と等価であることから，検査ツールなどで実行可能な JML 式を出力していることは明らかであり，ツールの変換が実用上有用であることを示している．しかし，コードの可読性などの観点から，構文的一致率を高めるための改良を施すべきであると考えている．次に，プロジェクトに含まれる変換不能な式の種類についての調査を行った．その結果，手動でも JML から OCL への変換ができない式は全体のうち，約 0.52%であった．本研究での調査結果から本質的に対応不能な JML は全体の 1%にも満たない割合であり，OCL，JML 間での双方向変換を行うことに問題はないと考えられる．また，変換に要する時間は数秒であり，実用的な範囲で変換が行えることが確認できた．

以降，2 章では研究の背景となる諸技術と関連研究について述べる．3 章，4 章では OCL から JML への変換手法の提案，および JML から OCL への変換手法の提案についてそれぞれ述べる．5 章では，それぞれの変換手法に基づいて行った実装についての詳細を述べる．6 章では実験の計測内容，実験方法，実験結果を述べ，考察を行う．最後に 7 章で本研究のまとめを述べる．

2 背景

本章では研究の背景となる諸技術と関連研究について簡単に触れる．

2.1 Design by Contract

Design by Contract[14] とは，オブジェクト指向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や信頼性および再利用性を向上させることを目的とするものである．契約とは，クラスの呼び出し側がそのクラスを利用する時点において，ある条件（事前条件）を満たすことを保証すれば，呼び出されたクラスはある条件（事後条件）を満たすことを保証することである．この契約により，事前条件を満たさない場合は利用したいクラスを呼び出すまでに不具合が生じていることになる．また，事後条件を満たさない場合は呼び出されたクラスの実行中に不具合が生じていることになる．これによって，ソフトウェアにおける不具合箇所の特定を容易にすることができる．

2.2 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頻繁に使用される．

UML の定義は Meta-Object Facility (MOF)[22] のメタモデルを使用して行われている．UML モデルは，QVT (Queries/Views/Transformations)[23] 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．

一般に用いられている UML は第 2 下層の M1 が相当する．

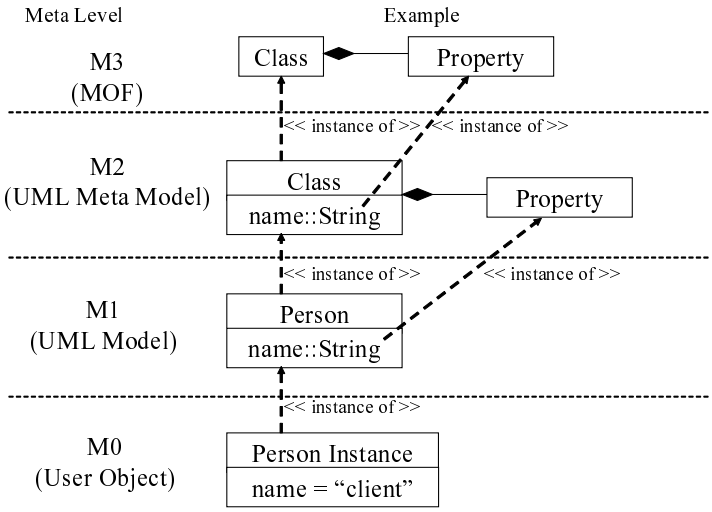

図 1: OMG の 4 層モデル

2.3 OCL

OCL[5] はソフトウェアのモデルを記述するためのモデリング言語の 1 つである．これは，OMG のオブジェクト指向分析/設計のための標準である UML の標準の追加機能として定義されている．

通常，UML 単体ではメソッドや属性に対する詳細な制約を持たせることはできず，自然言語での制約記述には曖昧さが残る．OCL は UML モデル内のモデル要素に対して正確に制約を与えることを目的に導入されており，条件式を宣言的な型付き言語で記述することにより，UML ダイアグラムに関する仕様をより厳密，かつ，詳細に表現できる．OCL を利用することにより，自然言語に比べてより厳密にモデルを定義することが可能になり，設計を実装に正しく反映することを支援できる．

また，OCL では Design by Contract の概念に基づき，クラスやオブジェクトのメソッドに対する事前条件，事後条件等を記述することができる．OCL は純粋な仕様記述言語であり，OCL で記述された式は副作用を一切もたらさないことが保証される．OCL 式が評価されると単純に値を返し，モデルの状態が変化することはない．これは，OCL 式がシステムのある状態の変化を定義していたとしても，OCL 式を評価することによるシステムの状態の変化は一切起こらない，ということを意味している．

初期の OCL は単なる UML の形式仕様記述言語としての拡張であったが，その後 UML だけでなく OMG の MOF のメタモデル全般を扱うようになった．また，OCL は OMG のモデル変換に関する推奨標準 QVT 仕様の一部である．他の多くのモデル変換言語，例えば ATLAS Transformation Language(ATL)[24] なども，OCL に基づいて構築されている．

2.4 JML

JML[6] は Java プログラムにおいて Design by Contract の概念に基づき，メソッドやオブジェクトの制約を事前条件，事後条件等の形で記述することができる．記述においては Java の文法を踏襲しており，初心者でも記述しやすいという特徴を持つ．また，JML 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，コンパイル時に影響はなく，アジャイル開発にも容易に適用できるようになっている．

JML 式は基本的には Java における bool 型を返すような任意の式で記述が行われる．条件記述のために以下のようなキーワードが拡張されている．

\ old(), \ result, \ forall(), \ exists().

\ old() は事後条件を記述する際に，メソッド実行前の変数値を参照する際に用いられる．\ result はメソッドの戻り値を参照する．\ forall() はコレクションのすべての要素がある条件を満たすことを指定したい場合に用いられる演算であり，\exists() はコレクション中にある条件を満たすオブジェクトが少なくとも 1 つ存在することを特定するために用いられる演算である．

JML は事前条件，事後条件などを記述するために，それぞれ ensures, requrires 等のキーワードを定義している．また，より実装に必要な条件を詳細に記述するために signals, pure, assignable, assert など豊富なキーワードを定義している．

また JML には，コード実行時に JML で記述した制約と実行時の値の矛盾を検出する JML Runtime Assertion Checker 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[25]，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[26] 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．

例として，図 2, 3 にそれぞれ同一メソッドに対する JML 記述と OCL 記述を示す．また表 1 に JML と OCL の記述の簡単な対応表を示す.

2.5 JML Runtime Assertion Checker

JMLrac(JML Runtime Assertion Checker) は JML 記述をもつ Java プログラムに対して，JML 記述に対する Java プログラムの実装の正しさを動的検査によりメソッド単位で行うことができるツール

表 1: JML と OCL 間の対応例

JML 記述	OCL 記述
requires	pre
ensures	post
\ old()	@pre
\ result	result

```
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 2: JML 記述例

である．

jml4c

jml4c は，JML 記述の付加された Java ソースコードに対するコンパイラであり，Java1.5 で新しく定義が行われた拡張 for 文や，ジェネリクスに対応している．JML で記述した制約を付加した Java ソースコードを入力すると，制約と実行時の値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．

jml4rt

jml4rt は，jml4c によってコンパイルされた実行ファイルを実行する．

2.6 モデル変換

モデル変換とは，あるメタモデルに従ったモデルを入力とし，別のメタモデルに従うモデルを出力する変換機構であり，モデル駆動型アーキテクチャ (MDA) の中心をなしている．モデル変換では，複数種類の入力モデルと複数種類の出力モデルを扱うこともある．

代表的なモデル変換として QVT[23] や ATL[24] などが挙げられる．これらは，MDA[1] 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

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

図 3: OCL 記述例

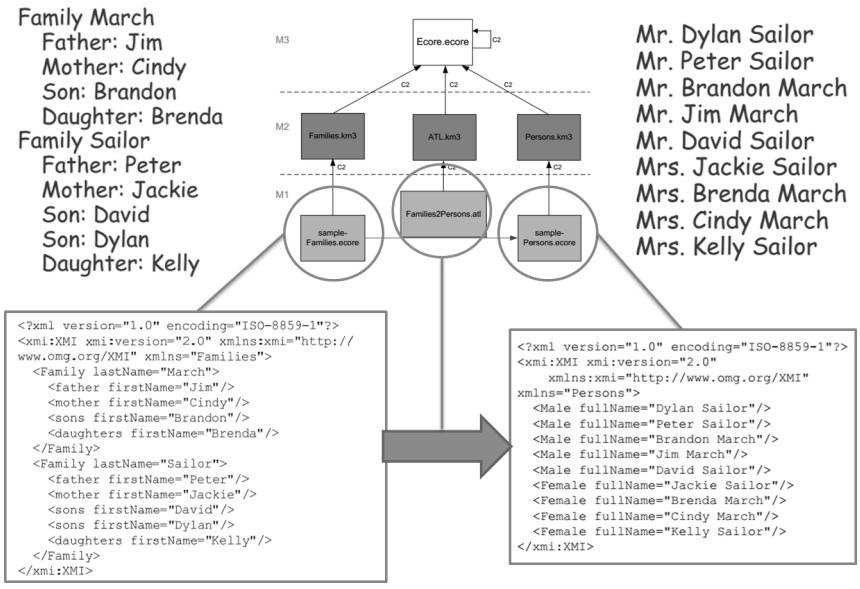

図 4: M2M 変換の例

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[2, 3] などが挙げられる．図 4 は，ATL による M2M 変換の一例である．この変換では，入力として sample-Families.ecore を受け取り，Families2Persons.atl によってモデル変換を行い，出力として sample-Persons.ecore を得ている．

2.7 ラウンドトリップエンジニアリング

ラウンドトリップエンジニアリング (RTE) とは，フォワードエンジニアリングとリバースエンジニアリングを反復しながら開発を行う手法である [15, 16]．フォワードエンジニアリングでは，クラス図やシーケンス図などのモデルを基にしてソースコードを開発し，リバースエンジニアリングでは，ソースコードからモデルを生成する. すなわち，RTE とはモデリング段階とコーディング段階を反復しながら開発を行う手法であると言える．このため，RTE は反復的なソフトウェア開発プロセ

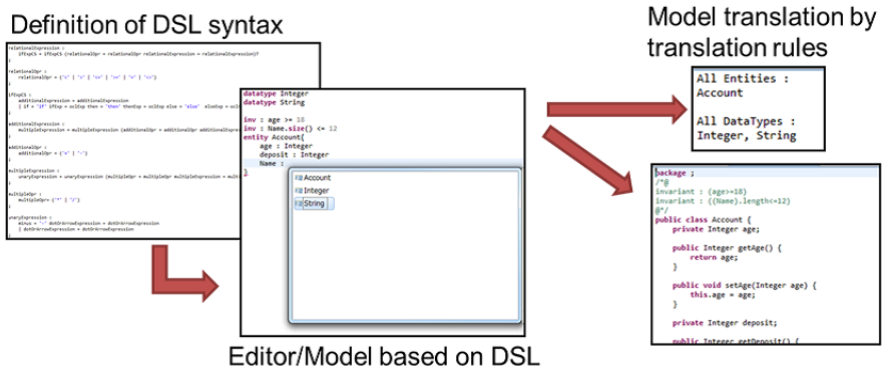

図 5: Xtext による開発の流れ

ス (プロトタイピングなど) に対して有効な手法であると言われている．

RTE による開発を行う際には，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 RTE では，モデルとソースコードの整合性を保つ必要性があり，自動的に整合性を保つ様々なツールが存在する．例えば，クラス図の変更をソースコードに自動的に反映させ，ソースコードの変更をクラス図に自動的に反映させるツールがある．そのようなツールを利用することで，クラス図とソースコードの整合性を保ちながら，モデリング段階とコーディング段階を往復することが可能になる．これらの機能は，静的な側面のモデルであるクラス図とソースコード間のサポートが中心となっている．また，動的モデルとソースコード間の RTE を支援するツールの提案も行われており，これらを組み合わせることで，より効率的な RTE が実現可能となる．

2.8 Xtext

Xtext[27] とは，モデルの構文定義や，構文定義に従ったモデルからテキストへの変換ルールの構築などをサポートするためのフレームワークである．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，定義した変換ルールに従って自動的にモデルをテキストに M2T 変換する．図 5 に Xtext による開発の流れを示す. まず変換元になるモデルの構文定義，及びモデルからテキストへの変換ルールの定義を行う．次に，モデルの構文定義から得られたエディタを利用して, 変換元となるモデルの記述を行う．モデルの記述を行うと，自動的に定義した変換ルールに従った変換後のテキストが得られる．このように，Xtext を利用した M2T 変換が行われる．

2.9 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 [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 [7] 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 [9] 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（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．

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

この場合，$sum + i$ の計算結果がメソッドに渡されるだけであり，$sum + i$ をメソッド内でループの度に繰り返し評価することができない．これは，本来期待した式の意味とは大きく異なってしまっている．

著者の研究グループでは，文献 [28] 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 [28] 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．

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．

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

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

図 6: iterate メソッド

作成されるメソッドは図 6 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．表 2 にそれぞれのツールが対応している演算を示す．表 2 の基本演算とは，論理演算，比較演算，算術演算，if 文などを指す.

また我々の研究グループでは，この提案手法に基づいた OCL から JML への変換ツールを Eclipse プラグインとして実装している [10, 11, 12]．図 7 に，既存ツールの概要を示す．既存ツールは MDA の主流であるモデル変換的な実装ではなく，抽象構文木を介したマッピングで実装されている．また，既存ツールは OCL から JML への単変換の達成を主な目的としており，単方向の変換に対する変換ルールの定義などは適切に行われている．しかし双方向の変換は考慮していないことや，ツールそのもののユーザビリティの低さなどが問題点として挙げられていた．また実装そのものに関しても，ANTLR[29] によって自動生成されたコードを元に実装を行っているため，コードの拡張性や可読性の低さも問題点として挙げられていた．そのため，本研究における OCL から JML への変換には，既存のツールは利用していない．また，構文定義，変換規則に関しては，双方向変換実現のために変更を加えた箇所が存在している．これらについては 5 章でより詳細に説明する．

以下で，OCL に関連する既存研究を紹介する．OCL は，ソフトウェア設計において非常に重要な役割を担っており，設計者にとって非常に有用なものであると考えられる．しかしながら，OCL の 1 つの欠点として，OCL は形式言語での一種であり，プログラマーやシステムの開発者にはあまり馴染みのない言語であることが挙げられる．

既存研究として，自然言語から OCL への変換が存在する [30]．この研究では，SBVR に従った自然言語から OCL への変換を実現している．この変換は，仕様を自然言語を用いて記述できるようにすることに加え，OCL の学習にも利用できることが期待できる．

また，Cheon らの研究では，OCL から複数の言語への変換を手動で行い，それらを用いて仕様検証を行った結果の比較を行っている [31]．この研究では OCL から (1) Java のソースコード，(2) アサーション，(3) JML，(4) AspectJ への変換を行っている．その結果，(1) と (2) に関しては，CPU やメモ

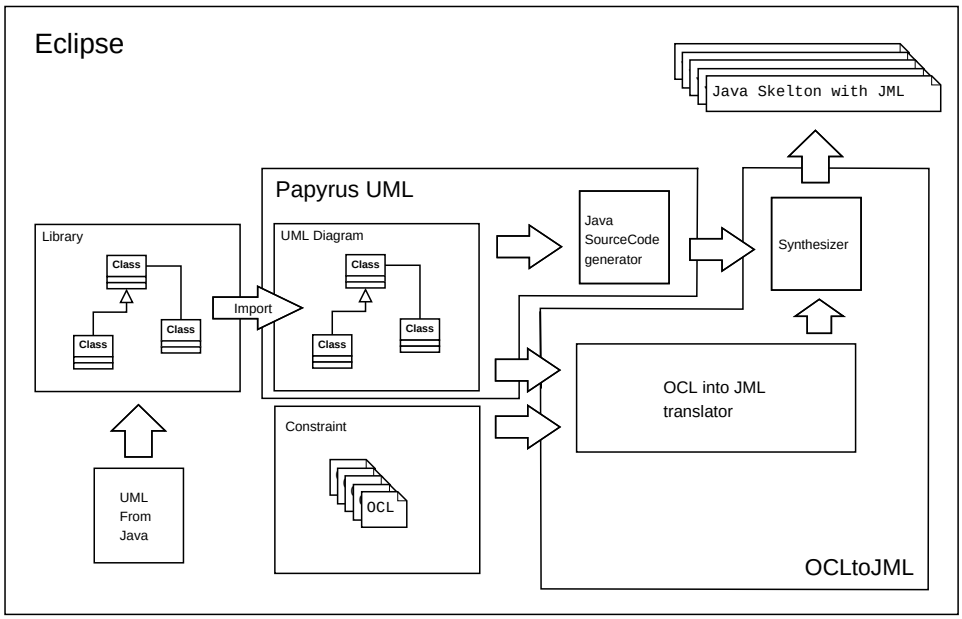

図 7: 既存ツールの概要

リ使用量の観点で有用であることが示された．また，(3) と (4) に関しては，変換やデバッグの容易さの観点で優れていることが示された．

また OCL が付加された UML クラス図から Alloy への変換手法も提案されている [32]．この変換を利用することで，設計者は Alloy による仕様検証を行うことが容易になる．さらに，文献 [33] では，UML と OCL から生成した Alloy を UML のオブジェクト図に戻す研究も行われている．この変換は，出力された Alloy のコードや，AlloyAnalyzer で検証した出力結果などユーザーが確認する必要を無くし，UML のオブジェクト図を見るだけで検証結果が確認できる状態を実現した．

以下で，モデル変換に関する既存研究を紹介する．モデル変換にかかるコストとして，変換の対象となるモデルが複雑で大規模な場合に，多くの時間とリソースを要するという問題点が挙げられる．そのような問題点を改善するために，部分評価に基づいた漸次的モデル変換手法を提案した研究が存在する [34]．モデル変換プログラムを部分評価し，その結果得られた残余プログラムを利用したモデル変換を行っている．

また，モデルとコードのトレーサビリティを確保することが，MDD においては重要であるが，これらは容易に失われてしまう．ユーザーが自動生成されたコードを任意に変更した場合や，コードの振る舞いを生成するためのテンプレートを変更した場合に失われる．そのような問題点を改善するために，モデルとコード間のトレーサビリティの不整合を減らすことを目的とした 2 階層のフレームワークが提案されている [35].

モデル駆動開発が一般的に大規模システム開発において注目されているが，小規模な開発に対するモデル駆動開発の有用性を示すことを目的とした研究が存在する [36]．この研究では，小規模な

表 2: 既存手法と研究グループの手法の比較

Feature	研究グループ	Moiseev と Russo	Hamie
基本演算	√	√	√
Collection	√	√	-
Iterate (forall etc.)	√	√	√
Iterate (collect)	√	√	-
Iterate (iterate)	√	-	-
Structures (Set etc.)	√	√	√
Structures (tuple)	√	√	-
Message	-	-	-

システム開発事例を対象として，モデル駆動開発を適用した場合とそうでない場合の比較を行っている．この研究では，ビジネスロジックのような複雑な部分を除けば，大部分はモデルから導出できることが確認できた．具体的には，既存の開発には 42 日を要したのに対し，MDD を利用した場合は 16 日 (このうち人手でのコーディングなどが 13 日) にとどまっており，小規模なシステム開発，短期間での開発にも MDD を適用することの有用性を示すことができている．

また，企業が MDD を導入することを妨げる要因で最も大きな理由は，企業における MDD の専門家の不足であると言われる．そこで，実際に MDD を利用している企業に対して調査を行い，MDD を導入するに当たり取り組むべき課題を示した研究が存在する [37]．環境，言語ごとに適切なツールを選択することが難しいことや，各ツールの互換性などが現場での課題として考えられていることが確認された．

Xtext を利用したモデル変換に関する既存研究も存在する．文献 [38] では，サービス指向アーキテクチャとプロセス指向アーキテクチャ間での双方向変換を，Xtext を用いて実現することを試みている．

表 3: 基本演算の μ 変換

$\mu(b_1$ `and` $b_2)$	=	($\mu(b_1)$ && $\mu(b_2)$)
$\mu(b_1$ `or` $b_2)$	=	($\mu(b_1)$ || $\mu(b_2)$)
$\mu(b_1$ `xor` $b_2)$	=	(($\mu(b_1)$ && !$\mu(b_2)$) | | ! ($\mu(b_1)$ && $\mu(b2)$))
$\mu(b_1$ `implies` $b_2)$	=	($\mu(b_1)$ ==> $\mu(b_2)$)
$\mu($ `not` $b_1)$	=	(!$\mu(b_1)$)
$\mu(b_1 = b_2)$	=	($\mu(b_1) = \mu(b_2)$)
$\mu(a_1 = a_2)$	=	($\mu(a_1)$ == $\mu(a_2)$)
$\mu(a_1 > a_2)$	=	($\mu(a_1) > \mu(a_2)$)
$\mu(a_1 < a_2)$	=	($\mu(a_1) < \mu(a_2)$)
$\mu(a_1 >= a_2)$	=	($\mu(a_1)$ >= $\mu(a_2)$)
$\mu(a_1 <= a_2)$	=	($\mu(a_1)$ <= $\mu(a_2)$)
$\mu(a_1 <> a_2)$	=	($\mu(a_1)$! = $\mu(a_2)$)
$\mu(c_1 = c_2)$	=	($\mu(c_1)$.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($\mu(c_2)$))
$\mu(a_1 + a_2)$	=	($\mu(a_1) + \mu(a_2)$)
$\mu(a_1 - a_2)$	=	($\mu(a_1) - \mu(a_2)$)
$\mu(a_1 * a_2)$	=	($\mu(a_1) * \mu(a_2)$)
$\mu(a_1/a_2)$	=	($\mu(a_1)/\mu(a_2)$)
$\mu(-a_1)$	=	($-\mu(a_1)$)

3 OCL から JML への変換についての提案手法

本章では OCL から JML への変換規則，変換に際して問題となった点，及びその解決方法を述べる．

3.1 変換の概要

まず初めに，変換手法全体の概要について述べる．基本演算のような直接的に 1 対 1 で対応できる変換の他にも，関連研究で述べたメソッド生成によるコレクションループの変換や，独自に定義したコレクション型を用いた変換などを定義している．また，双方向変換を行なう際に，ユーザによって変更が加えられていない式については，変換規則に適用するのではなく，変換元の式をそのまま出力することが望ましい．これを実現するために，変換結果の JML 式の上に，コメントとして元の

```
/*@ ensures translatedIteration1(\result) >= 100; @*/
private Collection sample(){}
private Integer translatedIteration1(Collection param_Collection){
    Integer res = 0;
    for (String e : param_Collection){
        res = (res + e);
    }
    return res;
}
```

図 8: iterate の変換例 1

OCL 情報と JML のハッシュコードを埋め込んでいる．変換結果の JML 式の上にコメントを埋め込むのは，どの OCL 式と，どの JML 式がそれぞれ対応しているのかというトレーサビリティを保つためである．これらについての詳細は，この章の残りの各節で説明していく．

3.2 基本演算の変換

まず基本演算について述べる．基本演算に関しては OCL・JML 間でほぼ 1 対 1 に対応させることができる．表 3 に変換規則の一部を示す．OCL 式から JML 式への変換関数の表記を μ で与えている．任意の型を持つ部分式を a_m，Bool 型を b_m，Collection 型を c_m としている．

3.3 コレクションループの変換

forall や exists などのコレクションループは，既存研究の手法 [28] で iterate に置き換えられ，意味的に等価な Java のメソッドを生成する．しかし，既存研究の方法では，対応できない部分がいくつか存在する．

まず，既存研究ではループの対象となるコレクションは自動生成されたメソッド内で変換しているが，式 5 のような式は既存の手法では変換できない．式 5 の先頭の result とは，OCL で戻り値を参照する際に用いられる予約語である．

$$result{->}\ iterate(e : Integer; res : Integer = 0 \mid res + e) >= 100 \tag{5}$$

JML で戻り値を参照するためには \result が用いられるので，OCL の result は JML の \result に変換される．しかし，Java のメソッド中などで \result は利用できない．そのため，既存の方法で式 5 を変換した場合，OCL の result が正しく扱えずに変換に失敗してしまう．本稿では，このような変換

```
Integer field;
Integer old_field;

/*@ ensures translatedIteration1(c,param,\result) > 10 ;@*/
private String sample(Integer param){
    old_field = field;
    return null;
}
private Integer translatedIteration1(
            Collection param_Collection,
                        Integer param, String result){
    Integer res = 0;
    for (String e : param_Collection){
        res = (e.equal(result) ? res + param : res + old_field);
    }
    return res;
}
```

図 9: iterate の変換例 2

を正しく行うために，コレクションを Java のメソッド内で変換するのではなく，自動生成されたメソッドの引数として与えている．式 5 は，図 8 のように変換することで，適切な変換を実現している．

式 6 のような iterate 演算が与えられた場合を考える．既存研究では，フィールド変数のみを iterate 内で利用する前提となっている．しかし，実際の JML 式では，戻り値，JML 挿入先のメソッドの引数，\old メソッドによるメソッド実行前の値なども含まれる．

$$c\text{-}> iterate(e : String; res : Integer = 0 \mid if\ e = result \\ then\ res + param\ else\ res + \backslash old(field)\ endif) > 10 \quad (6)$$

また，param を JML 挿入先のメソッドのパラメータ，field を int 型のフィールド変数とする．これらの値は既存研究における変換では対応できない．本稿の実装では，それらの値を自動生成されたメソッドの引数として保持することで変換に対応している．また，\old については，クラスのフィールド変数名の先頭に “old_” を追加した変数をメソッド呼び出し前の値を保持する変数として変換時に出力し，メソッドの先頭でフィールド変数の値を格納するようにしている．式 6 は，図 9 のように変換することで，適切な変換を実現している．

```
/*@ ensures translatedIteration1(c,\result);@*/
private String sample(){}
private Integer translatedIteration1(
            Collection param_Collection, String result){
    Boolean res1 = false;
    for (String e1 : param_Collection){
        res1 = (res1 || translatedIteration2(c2,e1,result));
    }
    return res1;
}
private Integer translatedIteration2(
            Collection param_Collection,
                        String e1, String result){
    Boolean res2 = false;
    for (String e2 : param_Collection){
        res2 = (res2 || (e1.equals(e2) ? true : false));
    }
    return res2;
}
```

図 10: iterate の変換例 3

次に式 7 のような式が与えられた場合を考える．

$$c_1 -> iterate(e_1 : String; res1 : Boolean = false \mid \\ res1 \ or \ c_2 -> iterate(e_2 : String; res2 : Boolean = false \mid \\ res2 \ or \ (if \ e_1 \ = \ e_2 \ then \ true \ else \ false \ endif))) \quad (7)$$

iterate が式 7 のように入れ子になっている場合，既存研究の手法では正しく変換できない．本稿の手法では，iterate が入れ子になっている場合，変換対象の iterate 演算よりも上層で定義された変数などについては，上述した例と同様に自動生成されたメソッドの引数として保持することで対応している．式 7 は，図 10 のように変換することで，適切な変換を実現している．

3.4 独自定義したコレクション型への変換

コレクション型についてはそれぞれ独自のクラスを定義し，定義したクラスを変換先として定義することで解決できると考えられる．OCL の Set 型に対応するクラスとして MySet 型，OrderedSet 型に対応するクラスとして MyOrderedSet 型，Bag 型に対応するクラスとして MyBag 型，Sequence 型に対応するクラスとして MySequence 型をそれぞれ定義する．変換はこれらのクラス間で一対一対応で行われるものとする．また，これまでの研究グループでのコレクション型の変換は表 4 のような μ 変換で行われていた．しかし，この変換定義は単方向では問題ないが，相互変換を繰り返すこ

表 4: 既存研究における Set 型の演算の μ 変換

$\mu(st_1$−>union(st_2))	=	$\mu(st_1)$.addAll($\mu(st_2)$)
$\mu(st_1$−>union(bg_1))	=	new List($\mu(st_1)$).addAll($\mu(bg_1)$)
$\mu(st_1$−>intersection(bg_1))	=	$\mu(st_1)$.retainAll($\mu(bg_1)$)
$\mu(st_1$−>including(a_1))	=	$\mu(st_1)$.add($\mu(a_1)$)
$\mu(st_1$−>excluding(a_1))	=	$\mu(st_1)$.remove($\mu(a_1)$)
$\mu(st_1$−>flatten())	=	μ(if st_1.selfType.elementType.oclKindOf(CollectionType) then st_1−>iterate(c; acc:Set() = Set{} | acc−>union(c−>asSet())) else st_1 endif)
$\mu(st_1$−>asSet())	=	$\mu(st_1)$
$\mu(st_1$−>asOrderedSet())	=	new ArrayList($\mu(st_1)$)
$\mu(st_1$−>asSequence())	=	new ArrayList($\mu(st_1)$)
$\mu(st_1$−>asBag())	=	new ArrayList($\mu(st_1)$)

とでコードの複雑さが増してしまう点，元のメソッドに一意に戻せない点などが問題点として挙げられる．また，new メソッドによるオブジェクトの生成は OCL では表現することができないため，既存研究のコレクション型の変換規則では，JML から OCL への逆変換を行うことができないものがいくつか存在する．これらの問題を解決するために，独自定義したクラス中に OCL のコレクション型の演算に意味的に対応するメソッドを OCL の演算と同一名で定義する．変換はそれぞれ OCL における”−>“と JML における”.“を置き換えるだけでよく，またメソッド中で意味的な解析を行うことも可能なことから，元の変換定義よりも相互変換を行う際には有用であると考えられる．

相互変換を繰り返した結果の OCL 式，および JML 式は他のツールにも問題なく適用できる．このことから，独自クラスを定義しても相互変換に問題はないことがわかる．

具体的な OCL のコレクション型の演算に対応した独自クラス内のメソッドの例を図 11 に示す．独自クラス内でメソッドを定義することにより，変換の簡単化に加えて，意味解析を Java 側の解析機で行うことが可能になる．union は，既存研究においての変換規則では，引数が Set 型なのか Bag 型なのかといった区別が必要となるが，図 11 のような形で union を定義することで，意味解析を完全

```
public class MySet extends HashSet{
  …
  public boolean NotEmpty(){
    return !(isEmpty);
  }

  public MySet union(MySet param){
    return this.addAll(param);
  }

  public MyBag union(MyBag param){
    return new MyBag(this).addAll(param);
  }
  …
}
```

図 11: 独自定義したクラスのメソッド例

```
op getAmount(name : String) : Integer{
  pre  : not name.oclIsUndefined() and name <> ''
  post : itemList->exists(i : Item |
         i.getTotalAmount() = result) or result = 0
}
```

図 12: OCL から JML への変換例 (入力)

に Java 側に依存させることが可能になる．

3.5 変換元の式情報の埋め込み

本実装では，OCL から JML への変換を行う際に，元の OCL 式をコメントの形で JML 中に保持する仕様になっている．逆変換を行う際にユーザーが編集をしていない式に関しては，この OCL 式をそのまま逆変換の結果として出力する．これにより，変更が加えられていない制約式に関しては，双方向変換を繰り返しても式が複雑化しないという利点が生まれる．また，本質的に対応不可能な JML 式に関しても，ユーザが変更を加えない限りは，逆変換できるようになるといった利点も生まれる．変更が加えられたかどうかを判別するために，JML 中にコメントとして，編集が加えられる前の JML 式のハッシュコードと，変換元の OCL 式を埋め込んでいる．変換対象の JML 式のハッシュ

```
//Original_OCL={pre: not name.oclIsUndefined() and name <> ''}
                                          ,JML_HashCode={378505836};
/*@ requires !(name == null) && !(name.equals("")); @*/
//Original_OCL={post: itemlist->exists(i:Item |
                    i.getAmount() = result) or result = 0}
                                          ,JML_HashCode={2016817273};
/*@ ensures translatedIteration3(itemlist, name, \result)
                                                  || \result==0; @*/
public Integer getAmount(String name) {
}
```

図 13: OCL から JML への変換例 (出力)

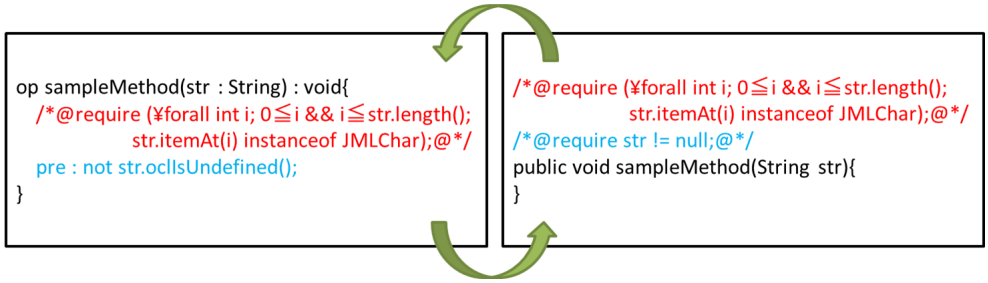

図 14: コメントとして JML 式を埋め込む変換の例

コードが埋め込まれた値と等しい場合には，埋め込まれた OCL 式を逆変換に利用し，異なる場合には，直接 JML から OCL への変換規則に従って変換する．JML から OCL への変換規則については，4 章で詳細に説明する．

具体的な変換例として，変換の入力を図 12 に，出力結果を図 13 に示す．図 13 の JML の事後条件の OCL への逆変換を行う際には，JML 式ではなくその上にコメントとして記述されている OCL 式を直接出力する．この例の場合は，“post : itemList −>exists(i : Item | i.getTotalAmount() = result) or result = 0” が出力される．

3.6 JML から変換された OCL の変換

JML から変換された OCL 式は，基本的には OCL から JML への変換規則に従って変換される．しかし，元の JML が複雑なため OCL で表現できないものに関しては，コメントとして直接 JML 式を OCL 中に保持している．コメントとして埋め込まれた式に関しては，直接 JML に変換する．この変換の概要を図 14 に示す．赤文字が JML から OCL への変換が直接的には不可能な式，青文字は直接

変換可能な式である．このような変換を実現することにより，双方向変換によって変換不可能な式の情報を失うことを防げる．手動でも変換不能な本質的に対応不可能な式も存在していることから，このように情報が失われないようにすることは有用であると考えられる．また，これらの変換についての詳細は 4 章で述べる．

4　JML から OCL への変換についての提案手法

本章では JML から OCL への変換規則，変換に際して問題となった点，及びその解決方法を述べる．

4.1　変換の概要

まず初めに，変換手法全体の概要について述べる．こちらも OCL 同様に，直接的に 1 対 1 で対応できる変換が多く存在する．しかし，その他にも双方向変換を実現するためには，OCL から JML への変換において，特殊な方法で行われた変換結果に対応する必要がある．具体的には，メソッド生成によって変換されたコレクションループへの対応や，独自に定義したコレクション型を用いた変換などへの対応が必要である．また，変更が行われていない式に，コメントとして変換元の OCL 情報が埋め込まれていた場合は，直接的に JML から OCL への変換を行うのではなく，埋め込まれた OCL 式を変換結果としてそのまま出力することが望ましい．その他，配列や JML のプリミティブな演算等，直接的には対応できない変換も多く存在する．そして，OCL の方が記述の粒度が荒く，JML の方が詳細な記述が行えるため，JML で記述された式の中には本質的に OCL で表現できないものも存在しているため，それらの式の扱いについても考察する必要がある．これらについての詳細は，この章の残りの各節で説明していく．

4.2　基本演算の変換

まず基本演算について述べる．基本演算に関しては，3 章で述べた通り，OCL・JML 間でほぼ 1 対 1 に対応させることができ，OCL から JML への変換の際に定義した表 3 とほぼ同様である．また，それ以外の変換規則の一部を表 5 に示す．JML 式から OCL 式への変換関数の表記を μ' で与えている．任意の型を持つ部分式を a_m，Bool 型を b_m としている．

4.3　配列の変換

Java では配列を用いることができるので，JML 中にも配列に関する制約式が出現することがある．しかし，OCL では配列に直接相当する概念は存在しない．そこで，配列はすべて OCL の Sequence として扱うこととしている．表 6 に変換規則を示す．JML 式から OCL 式への変換関数の表記を μ' で与えている．任意の型を持つ部分式を a_m，任意の型を TYPE としている．

ここでは配列の宣言部分，および配列を参照する際の変換のみを定義している．しかし，厳密には配列に対するメソッドもすべて考慮することが望ましい．例えば，“array.length” という式は Sequence として array を扱うのであれば，“array−>size()” に変換されるべきである．本研究の変換規則では，“array.lengh” については “array−>size()” に変換されるものとして扱っているが，すべてのメソッドに対応することはできていない．しかし，すべてのメソッドに対応するためには，Java のライブラリの情報なども解析しなければならず，現実的には実現は難しい．そのため，本研究で厳密に対応しているものは，“array.lengh” などの特定のメソッドのみにとどまっている．

表 5: JML から OCL への μ' 変換

$\mu'(b_1?b_2:b_3)$	=	if $\mu'(b_1)$ then $\mu'(b_2)$ else $\mu'(b_3)$ endif
$\mu'(b_1<==>b_2)$	=	$\mu'(b_1)$= $\mu'(b_2)$
$\mu'(b_1<=!=>b_2)$	=	$\mu'(b_1)$ <> $\mu'(b_2)$
$\mu'(b_1==>b_2)$	=	$\mu'(b_1)$ implies $\mu'(b_2)$
$\mu'(b_1<==b_2)$	=	$\mu'(b_2)$ implies $\mu'(b_1)$
$\mu'(b_1\&\&b_2)$	=	$\mu'(b_1)$ and $\mu'(b_2)$
$\mu'(b_1\|\|b_2)$	=	$\mu'(b_1)$ or $\mu'(b_2)$
$\mu'(b_1\|b_2)$	=	$\mu'(b_1)$ or $\mu'(b_2)$
$\mu'(b_1 \hat{} b_2)$	=	$\mu'(b_1$ xor μ'(b)
$\mu'(b_1\& b_2)$	=	$\mu'(b_1)$ and $\mu'(b_2)$
$\mu'(\backslash$result)	=	result
$\mu'(\backslash$old(a_1))	=	$\mu'(a_1)$@pre
$\mu'(\backslash$not_modified(a_1))	=	$\mu'(a_1)$ = $\mu'(a_1)$@pre
$\mu'(\backslash$fresh(a_1))	=	$\mu'(a_1)$.oclIsNew()

表 6: 配列に関する JML から OCL への μ' 変換

μ'(TYPE[] a_1)	=	$\mu'(a_1) : Sequence(\mu'$(TYPE))
μ'(TYPE a_1[])	=	$\mu'(a_1) : Sequence(\mu'$(TYPE))
$\mu'(a_1[a_2])$	=	$\mu'(a_1)$->at($\mu'(a_2)$)

4.4 コレクションループの変換

JML では，forall や exists のようなループは Java の for 文のように扱うことができる．しかし，OCL ではコレクション型に対して適用できるループのみ存在し，for 文のように扱うことはできない．そのため本研究では，コレクションに適用したループであることを前提に，構文を限定したループの変換規則を定義する．表 7，8 に変換規則を示す．JML 式から OCL 式への変換関数の表記を μ' で与えている．ループの対象となるコレクションを COLLECTION，任意の部分式を Expression，任意の型を TYPE としている．

表 7，8 からわかるように必ず条件式の部分に COLLECTION.contains(t) が記述されている形式に限定している．これはコレクション型に対するループ文で，contains メソッドがコレクションの要素を参照するために最低限必要な記述であるためである．しかし，このような制限はユーザーの制約記述を妨げる可能性が考えられる．

表 7: コレクションループに関する JML から OCL への μ' 変換表 1

μ'((\forall TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>forAll(μ'(t) : μ'(TYPE) | μ'(Expression1))

μ'((\exists TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>exists(μ'(t) : μ'(TYPE) | μ'(Expression1))

μ'((\max TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = null |

if res.oclIsUndefined() then μ'(Expression1)

else res.max(μ'(Expression1)) endif)

μ'((\min TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = null |

if res.oclIsUndefined() then μ'(Expression1)

else res.min(μ'(Expression1)) endif)

μ'((\sum TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 0 | res + μ'(Expression1))

μ'((\product TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 1 | res * μ'(Expression1))

μ'((\num_of TYPE t; COLLECTION.contains(t); Expression1) =

μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 0 |

if μ'(Expression1) then res + 1 else res endif)

4.5 ループ以外の JML のプリミティブな演算

JML にはコレクションループ意外にもプリミティブな演算が存在している．プリミティブな演算を含んだすべての記述に対応することは難しいため，ここでは構文の形式を限定した上で変換規則を定義する．表 9 にプリミティブな演算の変換規則の一部を示す．JML 式から OCL 式への変換関数の表記を μ' で与えている．任意の部分式を a_m，コレクション型を c_m，コレクション型以外の参照型やプリミティブな型を p_m としている．

4.6 OCL から変換された JML の変換

ここでは，OCL から変換された JML のうち，特に変換に工夫が必要なものに関しての説明を行う．

4.6.1 コレクションループを含む OCL から変換された JML の変換

コレクションループを含む式は，その式と意味的に等価な Java メソッドを生成し，そのメソッドを JML 側から呼び出す方法で対応している．この式を逆変換するためには，自動生成されたメソッ

表 8: コレクションループに関する JML から OCL への μ' 変換表 2

μ'((\forall TYPE t; COLLECTION.contains(t) && (Expression1); Expression2) =
μ'(COLLECTION)−>forAll(μ'(t) : μ'(TYPE) | μ'(Expression1) implies μ'(Expression2))

μ'((\exists TYPE t; COLLECTION.contains(t) && (Expression1); Expression2) =
μ'(COLLECTION)−>exists(μ'(t) : μ'(TYPE) | μ'(Expression1) μ'(Expression2))

μ'((\max TYPE t; COLLECTION.contains(t); && (Expression1); Expression2) =
μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = null |
if μ'(Expression1) then if res.oclIsUndefined() then μ'(Expression2)
else res.max(μ'(Expression2)) endif else res endif)

μ'((\min TYPE t; COLLECTION.contains(t); && (Expression1); Expression2) =
μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = null |
if μ'(Expression1) then if res.oclIsUndefined() then μ'(Expression2)
else res.min(μ'(Expression2)) endif else res endif)

μ'((\sum TYPE t; COLLECTION.contains(t); && (Expression1); Expression2) =
μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 0 |
if μ'(Expression1) then res + μ'(Expression2) else res endif)

μ'((\product TYPE t; COLLECTION.contains(t); && (Expression1); Expression2) =
μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 1 |
if μ'(Expression1) then res * μ'(Expression2) else res endif)

μ'((\num_of TYPE t; COLLECTION.contains(t); && (Expression1); Expression2) =
μ'(COLLECTION)−>iterate(μ'(t) : μ'(TYPE); res : Integer = 0 |
if μ'(Expression1) then if μ'(Expression2) then res + 1
else res endif else res endif)

ドの中身を解析しなければならない．メソッドの中身を無視して逆変換した場合，以下のように本来の意味が完全に失われてしまう．

1. “pre : c−>iterate(e : Integer; res : Integer = 0 | res + e) >= 10”
2. “requires translatedIteration1( ) >= 10”
3. “pre : translatedIteration1( ) >= 10”.

1 の式は，メソッドの自動生成を用いて 2 の式に変換される．2 の式の translatedIteration1 メソッドは，“pre : c−>iterate(e : Integer; res : Integer = 0 | res + e)” と意味的に等価な処理を行うメソッドとして生成されている．2 の式を，そのまま OCL 式に逆変換すると 3 の式のようになるが，OCL 側で translatedIteration1 メソッドの意味は保持することができない．そのため，本来の意味が失われてし

表 9: JML のプリミティブな演算に関する JML から OCL への μ' 変換の一部

$\mu'(\backslash\text{type}(a_1)\ ==\ \backslash\text{type}(a_2))\ =\ \mu'(a_1).\text{oclsTypeOf}(\mu'(a_2))$

$\mu'(\backslash\text{type}(a_1)\ !=\ \backslash\text{type}(a_2))\ =\ \text{not}\ \mu'(a_1).\text{oclsTypeOf}(\mu'(a_2))$

$\mu'(\backslash\text{typeof}(c_1)\ ==\ \backslash\text{typeof}(c_2))\ =$

$\mu'(c_1)->\text{asSequence}()->\text{first}().\text{oclsTypeOf}(\mu'(c_2)->\text{asSequence}()->\text{first}())$

$\mu'(\backslash\text{typeof}(c_1)\ !=\ \backslash\text{typeof}(c_2))\ =$

$\text{not}\ \mu'(c_1)->\text{asSequence}()->\text{first}().\text{oclsTypeOf}(\mu'(c_2)->\text{asSequence}()->\text{first}())$

$\mu'(\backslash\text{typeof}(p_1)\ ==\ \backslash\text{typeof}(p_2))\ =\ \mu'(p_1).\text{oclsTypeOf}(\mu'(p_2))$

$\mu'(\backslash\text{typeof}(p_1)\ !=\ \backslash\text{typeof}(p_2))\ =\ \text{not}\ \mu'(p_1).\text{oclsTypeOf}(\mu'(p_2))$

$\mu'(\backslash\text{nonnullelements}(a_1))\ =$

$(\text{not}\ \mu'(a_1).\text{oclIsUndefined}())\ \text{and}\ (\mu'(a_1)->\text{forAll}(\text{e : OclAny} \mid \text{not e.oclIsUndefined}()))$

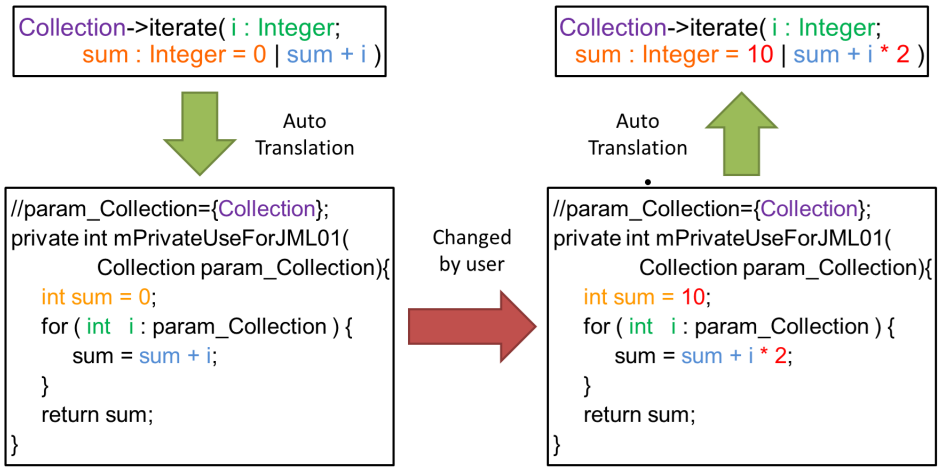

図 15: 自動生成されたメソッドの変換例

まうことになる．

このような状態を防ぐために，本実装では OCL から JML への変換の際に，自動生成されたコレクションループと意味的に等価なメソッドに限定してその処理の中身を解析することにしている．ただし，メソッドの中身を自由に編集することはできず，あくまでもテンプレートの形に従った変更のみを許すこととしている．図 15 に具体例を示す．色の付いた箇所がそれぞれ対応している箇所となっている．紫がコレクション，緑がコレクションの中身の要素，橙色がループの戻り値，水色が処理の中身となっている．また，図の赤色はユーザーによって変更が加えられた箇所を表している．この色が付けられた箇所に着目して，Java メソッドから OCL の iterate への逆変換を行っている．ユーザーはこの色が付けられた箇所には変更を加えることが可能となっている．また，生成されたメソッドの先頭に，コメントの形で元のコレクションの名前が格納されているのは，引数としてコレクションを

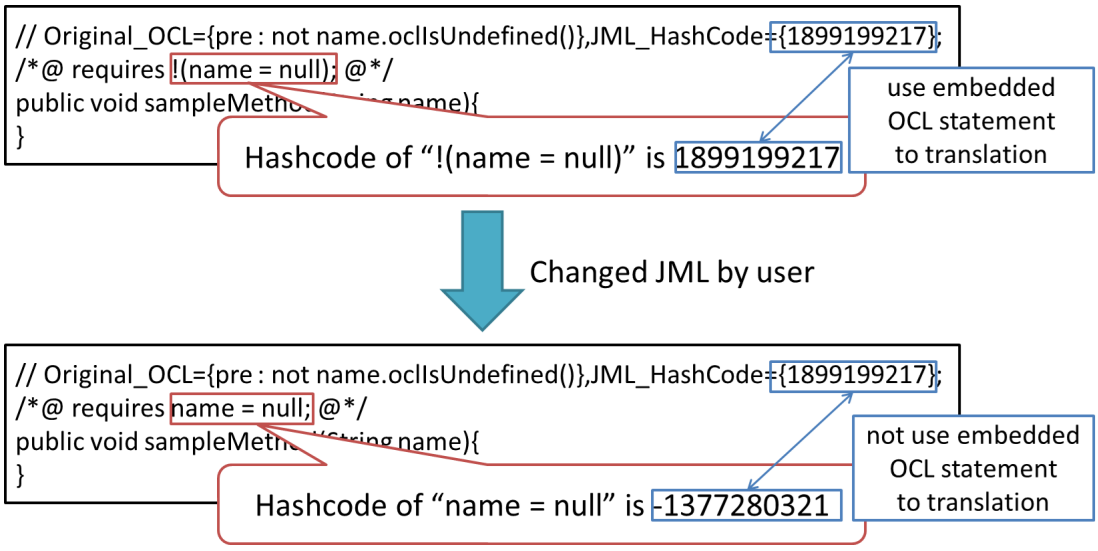

図 16: JML が変更されたか判別する方法

受け取る仕様にとなっていて本来のコレクション名が解析できない状態になるのを防ぐためである．ユーザーが対象のコレクションを変更したい場合は，コメントの中身のコレクション名を変更すればよい．

4.6.2 独自定義されたコレクション型の変換

3 章で述べた独自定義されたコレクション型の変換に関して述べる．まず表 10 に変換表を示す．

表 10: 独自クラスの μ' 変換の一部

$\mu'(st_1 \text{->union}(st_2))$	=	$\mu'(st_1).\text{union}(\mu'(st_2))$
$\mu'(st_1 \text{->intersection}(bg_1))$	=	$\mu'(st_1).\text{intersection}(\mu'(bg_1))$
$\mu'(st_1 \text{->including}(a_1))$	=	$\mu'(st_1).\text{including}(\mu'(a_1))$
$\mu'(st_1 \text{->excluding}(a_1))$	=	$\mu'(st_1).\text{excluding}(\mu'(a_1))$

表 10 からわかるように，メソッドに関しては，すべて “.” を “–>” に置き換えるだけでよい．また，型自体は MySet を Set，MyBag を Bag のように OCL のコレクション型に戻せばよい．

4.6.3 コメントとして埋め込まれた変換元の OCL の扱い

3 章で述べたように，OCL から JML への変換の際に変換元の OCL と，変換後の JML 式のハッシュコードをコメントとして埋め込んでいる．JML から OCL への変換の際に，この埋め込まれたハッシュコードの値と，変換対象の式のハッシュコードを比較することで，JML 式に変更が加えられたかどうかを判定している．処理の概要を図 16 に示す．

```
/*@require (\forall int i; 0 ≦ i && i ≦ str.length();
                        str.itemAt(i) instanceof JMLChar);@*/
public void sampleMethod(String str){
}
```

図 17: OCL への変換が本質的に不可能な JML 式 (入力)

```
op sampleMethod(str : String) : void{
    /*@require (\forall int i; 0 ≦ i && i ≦ str.length();
                        str.itemAt(i) instanceof JMLChar);@*/
}
```

図 18: OCL への変換が本質的に不可能な JML 式 (出力)

上の JML 式は “!(name == null)” から変更が加えられておらず，ハッシュコードの値が埋め込まれたものと等しいので，埋め込まれた OCL 式をそのまま出力することで逆変換を実現している．一方，下の JML 式は “!(name == null)” から “name == null” に変更が加えられており，ハッシュコードの値も埋め込まれたものとは異なる．よって，下の式は変更が加えられたものと判定され，変換規則に従って OCL への変換が行われることになる．このように変更したかどうかを識別し，JML から OCL への逆変換を行っている．

4.7 本質的に JML から OCL への変換が行えない式の扱い

JML は OCL よりも複雑な式を記述することが可能である．そのため JML では表現できても，OCL に変換することが本質的にできない式が存在する．図 17 の JML は，OCL に変換することが本質的に不可能な例である．

JML では，図 17 のように Java の for 文の様な記述を行うことが可能である．また，例のように String 型に適用することも，配列に適用することも可能である．しかし，OCL のループ演算はコレクション型に対応するもの以外は存在しない．そのため，図 17 のような式は，手動でも OCL 式として記述し直すことはできない．しかし，変換できない式もその情報が失われないように保持し，逆変換の際にその情報を復元できるようにすることが望ましい．そこで本実装では，変換が行えない式に関しては，JML 式の形のまま OCL 中に保持させて，逆変換の際に保持している JML 式を利用するようにしている．図 17 を本実装で変換すると，図 18 のような出力が得られる．このように，直接保持した JML 式を OCL から JML への変換に利用することで，変換不能な式の情報も保つことを可能にしている．

5 実装

本章では，実装に関して詳細に述べる．本研究では OCL・JML 間の双方向変換ツール実現のために Xtext を用いて実装を行っている．図 19 にその概要を示す．Xtext を用いて，OCL と JML の構文

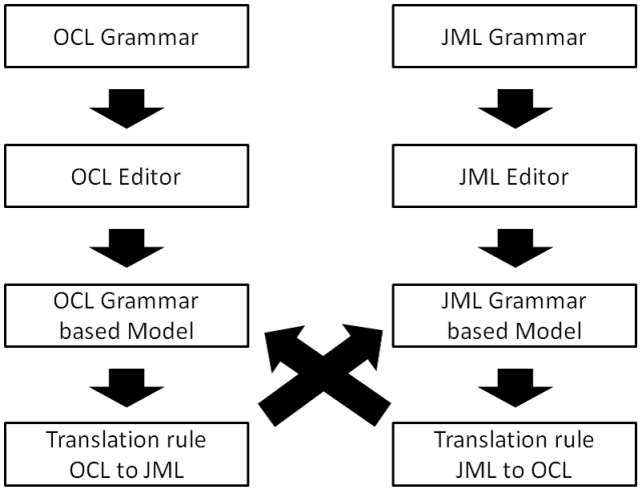

図 19: 実装の概要

を定義し，OCL から JML，および JML から OCL への変換規則をそれぞれ定義する．定義した構文，および変換規則を組み合わせることで双方向変換を実現する．変換規則の詳細については第 3 章，第 4 章で説明している．この実装方法には以下のような利点が挙げられる．

- モデルの構文定義は，テキストへの変換ルールの実装から独立しているため構文定義そのものの再利用性が高い．
- Xtext ではモデルの構文定義を行うことで，コード補完や文法エラーの検出を行えるユーザビリティの高いエディタが得られる．

実装の規模については，OCL の構文規則が 60 個，JML の構文規則が 98 個となった．また，OCL から JML への変換規則の行数は 3958 行となり，JML から OCL への変換規則の行数は 2423 行となった．

5.1 実装方針

ここでは，OCL から JML，及び JML から OCL への変換の方針について述べる．

5.1.1 OCL から JML への変換

まず，OCL を付加できる UML クラス図のモデルの構文定義を行った．UML の部分に関しては，Xtext で既存のものが存在するので流用し，OCL 部分を追加する形で構文定義を行った．OCL 部分

```
ifExpCS :
    additionalExpression = additionalExpression
    | if = 'if' ifExp = oclExp then = 'then' thenExp = oclExp else = 'else'  elseExp = oclExp endif = 'endif'
;

additionalExpression :
    multipleExpression = multipleExpression (additionalOpr = additionalOpr additionalExpression = additionalExpression)?
;

additionalOpr :
    additionalOpr = ('+' | '-')
;

multipleExpression :
    unaryExpression = unaryExpression (multipleOpr = multipleOpr multipleExpression = multipleExpression)?
;

multipleOpr :
    multipleOpr= ('*' | '/')
;

unaryExpression :
    minus = '-' dotOrArrowExpression = dotOrArrowExpression
    | dotOrArrowExpression = dotOrArrowExpression
;
```

図 20: UML 及び OCL モデルの構文定義の一部

```
ER:
    equal = '==' null = 'null'
    | equal = '==' relationalexpr = RelationalExpr
;
NR:
    notequal = '!=' null = 'null'
    | notequal = '!=' relationalexpr = RelationalExpr
;
RelationalExpr :
    shiftexpr = ShiftExpr small = '<' shiftexpr2 = ShiftExpr
    | shiftexpr = ShiftExpr big = '>' shiftexpr2 = ShiftExpr
    | shiftexpr = ShiftExpr smallequal = '<=' shiftexpr2 = ShiftExpr
    | shiftexpr = ShiftExpr bigequal = '>=' shiftexpr2 = ShiftExpr
    | shiftexpr = ShiftExpr instance = '<:' shiftexpr2 = ShiftExpr
    | shiftexpr = ShiftExpr (instanceo = 'instanceOf' typespec = TypeSpec)?
;
```

図 21: Java スケルトン及び JML モデルの構文定義の一部

に関しては，メソッド名や引数の数，戻り値の型などの場合分けを特に考慮した．変換ルールの作成は構文定義に依存するので，構文定義の段階で詳細な場合分けを行うことで意味的な解析にかかるコストが削減され，構文定義を再利用することの有用性が高まると考えられる．また，生成されるエディタのエラー検出やコンテンツアシストは構文定義に依存するので，構文定義の段階で場合分けを厳密に定義するほどコンテンツアシスト機能などが充実したエディタが得られる．これらのことから，メソッド名なども考慮した構文定義を行うことは，ユーザビリティの観点や実装の再利用性の観点から有用であると考えられる．詳細な変換規則などについては 3 章で記述したものになっている．

```
var str = new String();
if(i.embedded_JML != null){
    str = '''«i.embedded_JML.toString»'''.toString().substring(3,i.embedded_JML.toString().length()-3).replaceAll("\r\n","");
    jml_hashcode = str.replaceAll(" ","").hashCode();
    str = '''/*@ «str.toString.trim()» @*/'''.toString();
}
else{
    str = '''invariant«IF i.simpleNameCS != null»«i.simpleNameCS.compile»«ENDIF» «i.oclExp.compile»;'''.toString();
    str = '''«str.toString.replaceAll("\r\n","")»'''.toString();
    jml_hashcode = str.replaceAll(" ","").hashCode();
    str = '''/*@ «str.toString» @*/'''.toString();
}
if(jml_hashcode < 0){
    jml_hashcode = jml_hashcode * -1;
}
'''
    //Original_OCL={«tmp_oclstring»},JML_HashCode={«jml_hashcode»};
    «str.toString»
    «IF i.invOrDefCS != null»«i.invOrDefCS.compile»«ENDIF»
'''
}
```

図 22: OCL から JML への変換規則の定義の一部

```
def compile(FormalParameter f){
    if (f.lblace == null){
        '''«f.variabledeclaratorid.id.toString.replace("\r\n","")» : «f.type.compile.toString.replace("\r\n","")»'''
    }
    else{
        '''«f.variabledeclaratorid.id.toString.replace("\r\n","")» : Sequence(«f.type.compile.toString.replace("\r\n","")»)'''
    }
}

def compile(InvariantClause i){
    var str =
    '''
        «IF i.eContainer != null»
            inv: «i.predicate.compile»
        «ENDIF»
    '''.toString();
    currenttype = "other";
    '''«str.toString»'''
}
```

図 23: JML から OCL への変換規則の定義の一部

5.1.2 JML から OCL への変換

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．このとき変換を簡単化するために，元の構文から意味が変わらない範囲での変更を加えた．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保持しておく．実際の変換時には，型の情報が必要な部分はその保持している情報を元に 4 章で定義した形に変換していく．

5.2 モデルの構文定義

まず，UML と OCL が記述可能なモデルの構文定義を行った．UML に関しては，Xtext で既存のものが存在するので流用し，OCL 部分を追加する形で構文定義を行った．追加した OCL は，文献 [5]，及び文献 [39] に従った EBNF をベースに定義を行っている．しかし，仕様書の OCL はそのままでは不備が存在したことや，構文定義の段階で場合分けを詳細にすることで意味解析を簡単化す

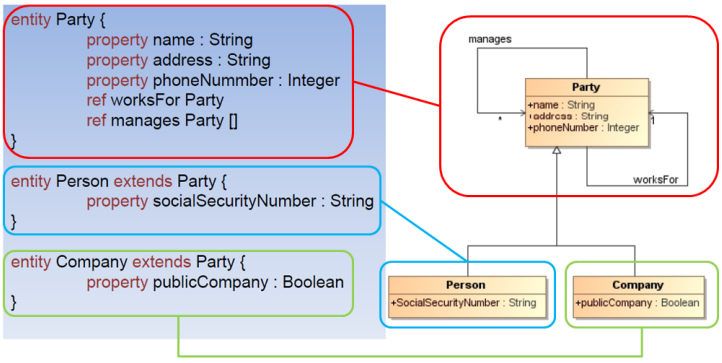

図 24: テキスト型モデル

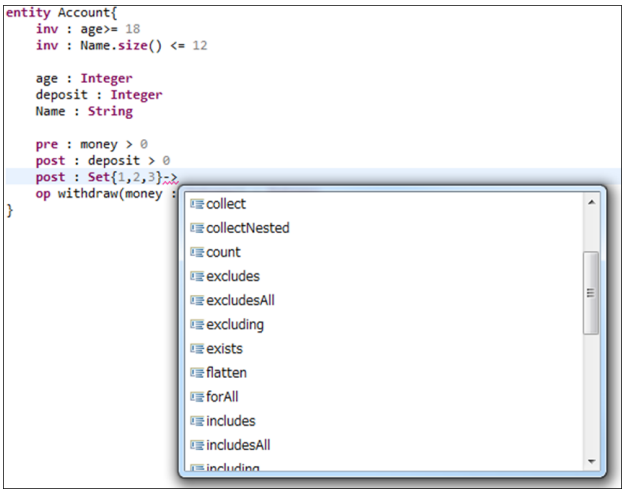

図 25: コンテンツアシスト機能

るため，元の EBNF に意味が変わらない範囲での改良を加えている．Xtext による構文記述の例として，UML 及び OCL モデルの構文定義の一部を図 20 に示す.

次に，Java のスケルトンコードと JML が記述できるモデルの構文定義を行った．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[40] 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を扱っている．本来 JML は Java ソースコード中にコメントの形式で “/*@” と “@*/” の間に記述するものである．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．Xtext による構文記述の例として，Java スケルトン及び JML モデルの構文定義の一部を図 21 に示す.

```
entity Storage{
    inv: allitemlist->forAll(i:Item,j:Item |
            allitemlist->indexOf(i) <> allitemlist->indexOf(j) implies i.getName() <> j.getName())
    inv: containerlist->forAll(ci:ContainerItem,cj:ContainerItem |
            containerlist->indexOf(ci) <> containerlist->indexOf(cj) implies ci.getContainerID() <> cj.getContainerID())
    containerlist : Sequence
    allitemlist : Sequence
```

図 26: OCL から JML への変換例 (入力)

```
class Storage {
    //Original_OCL={inv: allitemlist->forAll(i:Item,j:Item |
            allitemlist->indexOf(i) <> allitemlist->indexOf(j) implies i.getName() <> j.getName()
    )},JML_HashCode={1306241286};
    /*@ invariant translatedIteration1(allitemlist); @*/
    //Original_OCL={inv: containerlist->forAll(ci:ContainerItem,cj:ContainerItem |
            containerlist->indexOf(ci) <> containerlist->indexOf(cj) implies ci.getContainerID() <> cj.getContainerID()
    )},JML_HashCode={1915621652};
    /*@ invariant translatedIteration2(containerlist); @*/
    private MySequence containerlist;
    private MySequence old_containerlist;
    private MySequence allitemlist;
    private MySequence old_allitemlist;
    //param_Collection={allitemlist};
    private boolean translatedIteration1(MySequence<Item> param_Collection){
        boolean res = true;
        for(Item i : param_Collection){
            for(Item j : param_Collection){
                res = (res && (allitemlist.indexOf(i)!=allitemlist.indexOf(j) ?
                            !(i.getName().equals(j.getName())) : true
                ));
            }
        }
        return res;
    }
    //param_Collection={containerlist};
    private boolean translatedIteration2(MySequence<ContainerItem> param_Collection){
        boolean res = true;
        for(ContainerItem ci : param_Collection){
            for(ContainerItem cj : param_Collection){
                res = (res && (containerlist.indexOf(ci)!=containerlist.indexOf(cj) ?
                            !(ci.getContainerID().equals(cj.getContainerID())) : true
                ));
            }
        }
        return res;
    }
```

図 27: OCL から JML への変換例 (出力)

5.3 変換規則の定義

まず，OCL から JML への変換規則を定義した．詳細な変換規則については 3 章で述べたものに従っている．意味的な変換を適切に行うために，UML 部分の情報を利用している．具体的には，変数の型情報，メソッドの戻り値の情報などを保持して，変換時に適用する変換規則を識別している．Xtext による記述例として，OCL から JML への変換規則の一部を図 22 に示す．

次に JML から OCL への変換規則を定義した．詳細な変換規則については 4 章で述べたものに従っている．意味的な変換を適切に行うために，Java スケルトン部分の情報を利用している．OCL から JML 同様に，変数の型情報，メソッドの戻り値の情報などを保持して，変換時に適用する変換規則を識別している．Xtext による記述例として，定義した変換規則の一部を図 23 に示す．

```
class Storage{
    private LinkedList containerlist;
    private LinkedList allitemlist;

    /*@ invariant \typeof(containerlist) == \type(ContainerItem); @*/
    /*@ invariant \typeof(allitemlist) == \type(Item); @*/
    /*@ invariant (\forall Item i,j;
            allitemlist.contains(i) && allitemlist.contains(j) && i!=j;!i.getName().equals(j.getName())); @*/
    /*@ invariant (\forall ContainerItem ci, cj;
            containerlist.contains(ci) && containerlist.contains(cj) && ci!=cj; ci.getContainerID()!=cj.getContainerID()); @*/
```

図 28: JML から OCL への変換例 (入力)

```
entity Storage{
    containerlist : Sequence
    allitemlist : Sequence
    inv: containerlist->asSequence()->first().oclIsTypeOf(ContainerItem)
    inv: allitemlist->asSequence()->first().oclIsTypeOf(Item)
    inv: allitemlist->forAll(i : Item, j : Item |
            (i <> j) implies (not(i.getName() = j.getName())))
    inv: containerlist->forAll(ci : ContainerItem, cj : ContainerItem |
            (ci <> cj) implies (ci.getContainerID() <> cj.getContainerID()))
```

図 29: JML から OCL への変換例 (出力)

5.4 意味解析

この節では，変換時の意味解析について述べる．例として，2 つのオブジェクトが同値であることを評価する演算を挙げる．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．変換を正しく行うためには，クラス変数，メソッドの引数，メソッドの戻り値の型などの情報を保持し，それらを踏まえた変換を行わなければならない．本実装では，変数名やメソッド名をキーとしてその型名を返す HashMap を作成し，変換時にそれらの変数やメソッドが現れたとき，Map から型を取得し，意味解析を行っている．これは OCL から JML への変換でも，JML から OCL への変換でも同様である．

5.5 ツールの動作例

本ツールでは，入力データはテキスト型モデルとして作成する．図 24 にテキスト型モデルの例を示す．図 24 の左側がテキスト型モデルであり，右側のグラフィカルなモデルと意味的に等価な記述が行われている．

次に，入力データの作成時のコンテンツアシスト機能の使用例を図 25 に示す．図 24 にテキスト型モデルの例を示す．ツールは Eclipse プラグインとなっており，Eclipse 上のエディタでモデルを記述する際に，定義した構文に従った要素を入力候補として補完するコンテンツアシスト機能が備わっている．図 25 では，OCL の Set クラスが取りうるメソッドの候補を補完している．この機能により，定義に従った，すなわち変換が可能な入力モデルの記述が容易になる

また，ツールによる変換の例として，図 26，27 に OCL から JML への変換を示す．図 26 は，実験対象で用いる在庫管理システムのクラスの Storage クラスの一部を示している．在庫管理システム

についての詳細は 5 章で述べる．図 27 が出力された Java のスケルトン及び JML の一部である．変数名の先頭に old_を付加している変数が新たに生成されているが，これはコレクションループと意味的に等価な Java のメソッド中に \old で変数が参照された場合に対応するために生成されたものであり，詳細は 3 章で述べている．2 つのメソッドはコレクションループと意味的に等価なメソッドであり，それぞれ JML から参照されていることがわかる．また，それぞれの JML 式の上にコメントの形で変換元の OCL 式と，変更前の JML 式のハッシュコードが埋め込まれている．これらの埋め込まれた情報は逆変換の際に利用されるものであり，詳細は 4 章で述べている．

次に，図 28，29 に JML から OCL への変換の実行例を示す．こちらも OCL から JML への変換例と同様に，実験対象で用いる在庫管理システムのクラスの Storage クラスの一部を示している．この例では，ハッシュコードと変換元となる OCL が 1 つも埋め込まれていない．そのため，すべての式が定義した変換ルールに従って JML から OCL へ変換されている．

6 評価実験

本章では，評価実験に関して詳細に述べる．本研究では合計 9 個の JML が記述された Java プロジェクトに対する適用実験を行った．プロジェクトのうち 2 個は我々の研究グループで作成されたプロジェクトとなっている．残りの 7 個はオープンソースのプロジェクトを用いた．

6.1 計測内容

本研究では以下の点について計測を行う．

- 変換成功率
  JML から OCL への変換の際の変換成功率を計測する．
- 逆変換構文的一致率
  JML から変換された OCL を，再び JML へ逆変換した際に，元の構文とほぼ一致していた式の割合を計測する．実験における構文的一致とは，改行，スペース，タブ，括弧に関するもの以外に違いがないものを指している．意味は変わらないが，括弧が増えた場合なども完全一致からは除外する．例えば，“(X == 10)” と “X==10” は構文的一致とみなす．
- 逆変換意味的一致率
  JML から変換された OCL を，再び JML へ逆変換した際に，構文的には一致していないが，意味的には変換元の式と等価な式の割合を計測する．例として，“a != null” が元の式で，逆変換後の式が “!(a == null)” である場合を挙げる．この 2 つの式は，意味的には同じことを指しているが，構文的には異なる形式を取っている．また，構文的に一致しているものは意味的にも一致していることから，本実験では意味的一致率の方が構文的一致率よりも高くなる．
- 本質的に変換不可能な式の割合
  OCL から JML はすべて変換可能であるが，JML から OCL は本質的に変換不可能なものが存在するので，実プロジェクトにおいてそのような式が含まれている割合を計測する．

6.2 実験対象

実験対象のプロジェクトについて説明する．

まず，我々の研究グループで作成された 2 つのプロジェクトについての説明を行う．1 つは在庫管理プログラム [41] である．在庫管理プログラムは倉庫内の商品管理を行うためのシステムである．図 30 にプログラムの UML クラス図を示す．在庫管理プログラムは 7 クラスから構成されており，付加されている JML 数は 142 となっている．また，付加されている JML 式の正しさは Esc/java2，及び JML4c による検証により保証されている．

もう 1 つは，文部科学省の IT スペシャリスト育成推進プログラム (IT Spiral)[42] の教材として作成された教務システムである．図 31 と図 32 に教務システムの一部のクラス図を示す．このシステム

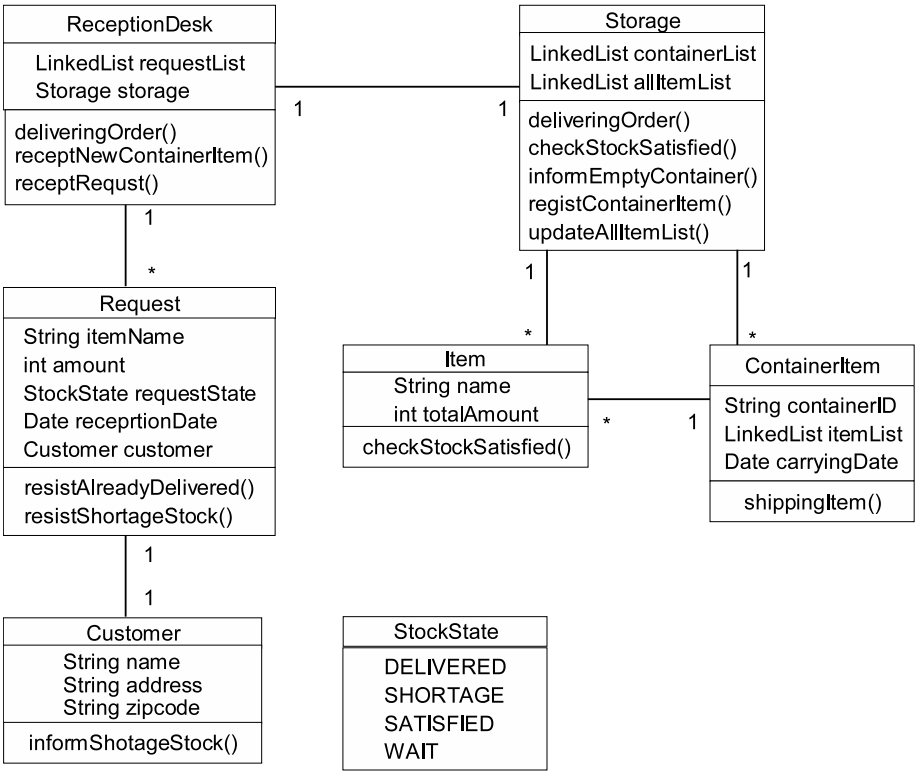

図 30: 在庫管理システムの UML クラス図

は Java で作成されており，hibernate を用いたデータベースプログラミングが行われている点が特徴として挙げられる．また，実装フェーズにおいて作成されたシステムのクラス数は約 200 クラスであり，付加されている JML 数は 468 となっている．

それ以外の 7 つのプロジェクトは，すべて github[43] に公開されていたオープンソースを利用している．プロジェクト名はそれぞれ，PokerTop，101JMLSpecifications，consultorOrtografico，Zinara，Lenguajes_III，P2-master，extweka である．それぞれの JML が付加されていたクラス数と付加された JML 数を表 11 に示す.

6.3 実験環境

実験環境として CPU は Intel ®Core™2 Duo E7300 @2.66GHz 2.67GHz， メモリは 4.00GB，OS は Windows 7 64bit を用いた.

6.4 実験結果

実験は以下の手順で行った.

1. JML が付加された Java プロジェクトの各クラスの JML をツールに適応可能な形式に変更.

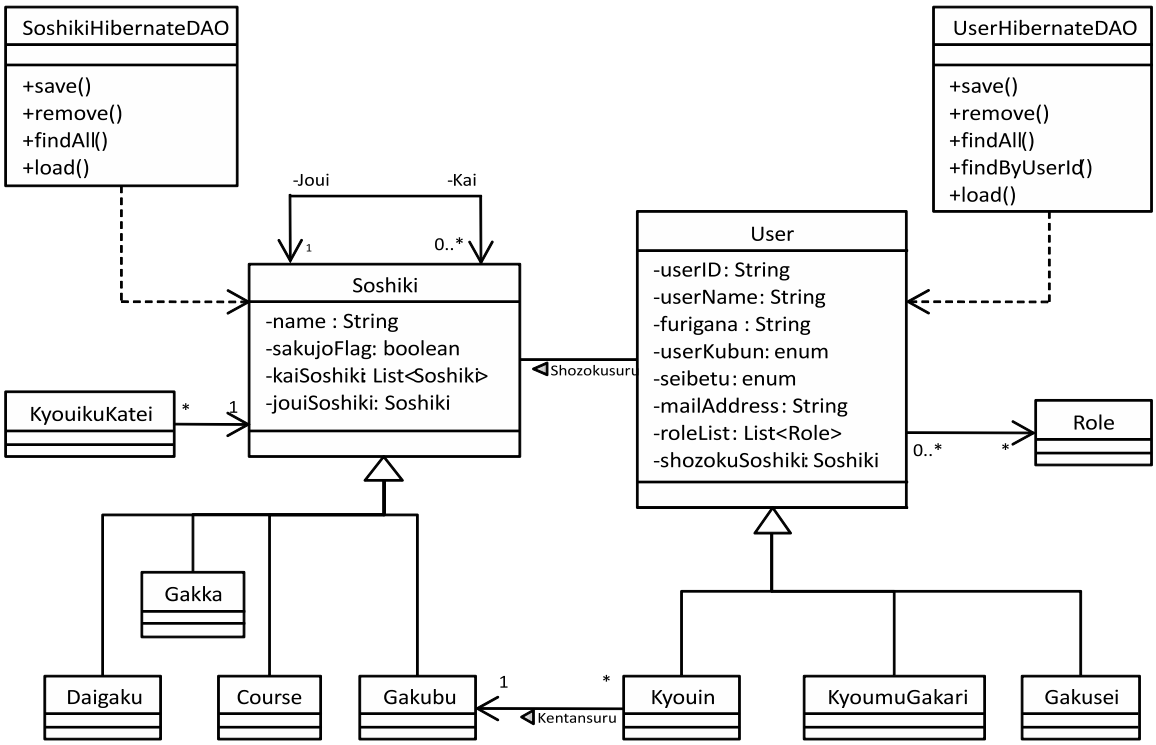

図 31: 教務システムのクラス図 1

2. 変更したプロジェクトをツールに適用し，OCL が付加された UML クラス図に相当するテキスト型モデルを生成.

3. JML から OCL への変換成功数を計測.

4. 生成されたテキスト型モデルをツールに適用し，JML が付加された Java スケルトンを逆変換によって生成.

5. OCL から JML への逆変換結果の意味的一致率，構文的一致率を計測.

まず，1. について説明する．本ツールの JML は 1 つ 1 つの事前条件や事後条件を “/*@” と “@*/” で囲まなければならず，複数の JML 式をまとめて “/*@” と “@*/” の間に記述することはできない．各 JML 式を分割しているのは，OCL から JML を生成する際に埋め込まれる OCL 式が，どの JML と対応しているかのトレーサビリティを保つためである．しかし，実プロジェクト中では複数の JML 記述をまとめて “/*@” と “@*/” の間に書くことが多いことや，ユーザの記述の自由度を制限してしまうことから改善すべき点であると考えている．

2. について説明する．すべての JML が付加された Java のクラスに変換を適用した．変換に要した時間は 2 秒程度であり，変換にかかる時間は実用的な範囲であると考えられる．また，生成された

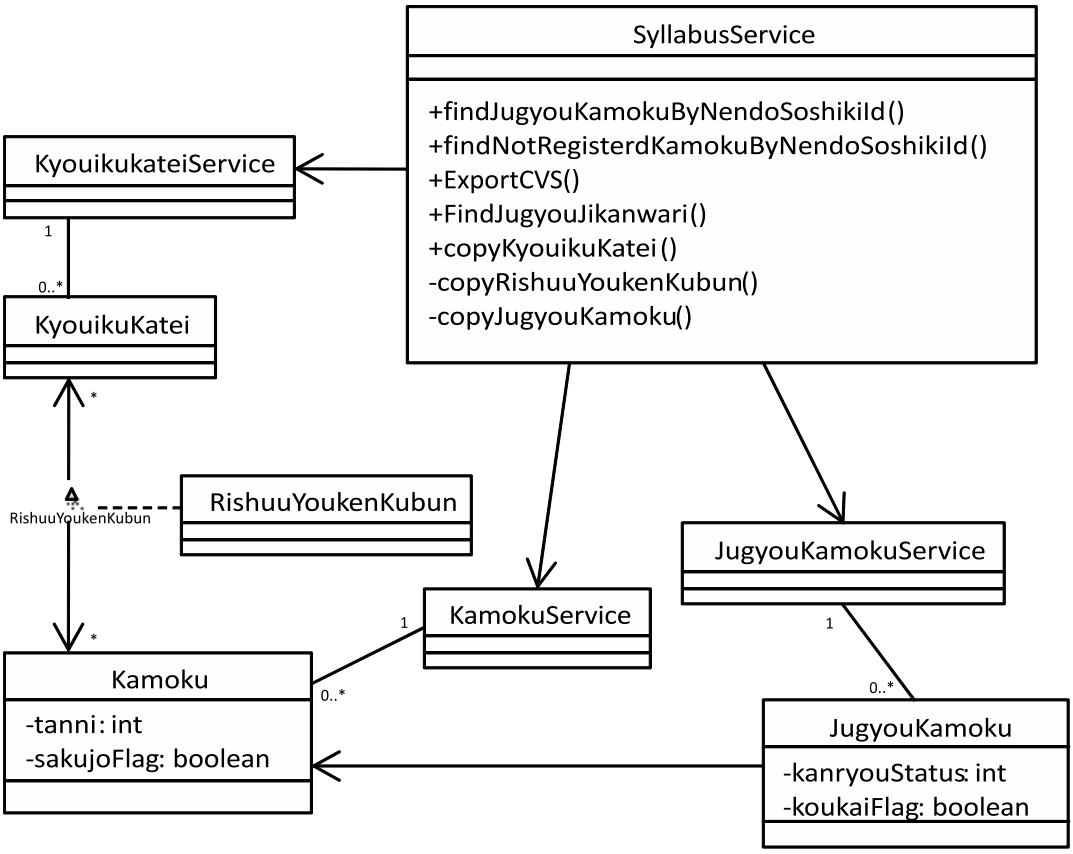

図 32: 教務システムのクラス図 2

OCL が付加された UML クラス図に相当するテキスト型モデルは，本ツールで定義した構文定義に従ったものであるため，そのまま逆変換を行うことが可能になっている．

3. について説明する．ここでは OCL への変換に成功した式の数を計測した．成功したかどうかは，人手で 1 つ 1 つの式の確認を行った．式の意味が正しく保たれているものは変換に成功したものとして計測している．

4. について説明する．JML から OCL への変換に成功した式をすべてツールに適用して逆変換を行った．変換に要した時間は JML から OCL への変換と同様に 2 秒程度であり，変換にかかる時間は実用的な範囲であると考えられる．生成された JML が付加された Java スケルトンは，本ツールで定義した構文定義に従ったものであるため，そのままツールに適用できる形式になっている．

5. について説明する．ここでは JML への逆変換に成功した式の数を計測した．成功したかどうかは，3. の場合と同様に，人手で 1 つ 1 つの式の確認を行った．

表 12 に計測した JML から OCL への変換数を示し，表 13 に逆変換した結果の式の意味的一致率と構文的一致率を示す．表 13 における構文的一致率の母数は，意味的一致数となっている．また，表 14 に変換できなかった式のうち，手動でも変換不能な本質的に対応不能な式の割合を示す．以下に

表 11: オープンソースプロジェクトのクラス数，JML 数

Class	クラス数	JML 数
PokerTop	9	113
101JMLSpecifications	3	30
consultorOrtografico	6	124
Zinara	28	126
Lenguajes_III	16	65
P2-master	5	56
extweka	10	233
TOTAL	77	747

それぞれ JML から OCL への変換率，逆変換の意味的一致，及び逆変換の構文的一致についての詳細を述べる．

表 12: JML から OCL への変換成功率

プロジェクト名	総 JML 数	変換成功数	変換成功率
在庫管理プログラム	142	142	100%
教務システム	468	444	94.5%
PokerTop	113	113	100%
101JMLSpecifications	30	30	100%
consultorOrtografico	124	84	67.7%
Zinara	128	106	82.8%
Lenguajes_III	65	65	100%
P2-master	56	56	100%
extweka	233	230	98.7%
TOTAL	1359	1270	93.5%

6.4.1 JML から OCL への変換成功率

全 10 クラスのうち 5 クラスは変換率は 100%であった．これらのクラスに含まれている式の大半は基本演算が占めていた．また，含まれていた \forall などのコレクションループに関しても，4 章で定義した変換規則に従ったもののみであった．

次に，変換率が 100%ではなかった残りの 4 クラスについて順に述べていく．まず，教務システム

表 13: 逆変換の意味的一致率と構文的一致率

プロジェクト名	逆変換式数	意味一致数	意味的一致率	構文一致数	構文的一致率
在庫管理プログラム	142	142	100%	62	43.7%
教務システム	444	444	100%	214	48.2%
PokerTop	113	103	91.2%	74	65.5%
101JMLSpecifications	30	26	86.7%	9	30%
consultorOrtografico	84	82	97.6%	63	75%
Zinara	106	96	90.6%	36	34.0%
Lenguajes_III	65	65	100%	0	0%
P2-master	56	30	53.6%	18	32.1%
extweka	230	200	87.0%	115	50%
TOTAL	1270	1188	93.5%	591	46.5%

について述べる．教務システムの JML のうち 26 個の式に関して変換を行うことができなかった．変換できなかった 26 個のうち，7 個が配列に対するコレクション演算を含むものであった．例えば，式 8 のような式である．

$$/*@ensures\ (\backslash forall\ int\ i;\ i >= 0 \&\& i\ <\ this.keikaku.length();\\ this.keikaku[i].equals(keikaku[i]));@*/ \tag{8}$$

このような式は，手動であれば配列を Sequence として扱うことで，式 9 のように置き換えることは可能である．

$$post\ :\ self.keikaku{->}\ forall(JugyouShousaiKeikaku\ j\ |\\ (keikaku.contains(j))\ and\ (self.keikaku->indexOf(j)\ =\ keikaku->indexOf(j))) \tag{9}$$

しかし，このような式すべてに対して変換規則を定義し，すべてを自動化することは厳密には対応不可能である．また，残りの 19 個の式はクラスリテラルを含む式である．具体的には，式 10 のような式である．

$$/*@\ requires\ getDAO(SoshikiDAO.class)\ !=\ null;@*/ \tag{10}$$

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．また，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．

次に，consultorOrtografico について述べる．consultorOrtografico の JML のうち，39 個の式に関して変換を行うことができなかった．変換できなかった 39 個の式のうち，36 個の式は配列に関するコ

表 14: 変換できなかった式のうち本質的に変換不能な式の割合

プロジェクト名	OCL への変換失敗数	変換不能式の数	変換不能式の割合
在庫管理プログラム	0	0	0%
教務システム	7	0	0%
PokerTop	0	0	0%
101JMLSpecifications	0	0	0%
consultorOrtografico	39	3	7.69%
Zinara	22	4	18.18%
Lenguajes_III	0	0	0%
P2-master	0	0	0%
extweka	3	0	0%
TOTAL	71	7	9.86%

レクション演算を含むもので，3 個の式は new メソッドを用いたオブジェクト生成を含んだものである．配列に関する 36 個の式に関しては，教務システム同様に手動であれば OCL に書き換えることは可能だが，自動化は難しいものである．しかし，オブジェクト生成を含む 3 個の式は，手動でも意味的に等価な式を記述することは不可能である．これは，OCL が実行型の言語ではなく，オブジェクトの生成などに相当する演算を表現することができないためである．具体的には，式 11 のような式を OCL へは変換できない.

$$/*@\,requires\,bf(p)\;\&\&\;!vocabulario.has(new\;JMLString(p));@*/ \tag{11}$$

次に，Zinara について述べる．consultorOrtografico の JML のうち，22 個の式に関して変換を行うことができなかった．変換できなかった 22 個の式に関しては，配列に関するループ演算を含む式が 6 個，定義した形式と異なるループ演算を含む式が 4 個，型の比較に関する演算を含む式が 8 個，\lockset という JML 独自の演算を含む式が 4 個である．配列に関するループ演算は他の 2 つのプロジェクト同様に自動での対応が難しいものである．定義した形式と異なるループ演算の例として，式 12 を示す.

$$/*@\,requires\,(\backslash forall\,T\;t;\;t.x\;!=\;0);@*/ \tag{12}$$

コレクション型に対するものではなかったため，手動でも変換することはできないものである．型の比較に関しては，本研究で定義した形式に従っていないものが 4 つ含まれていた．本研究では，表 9 に示しているような，型の比較に関する演算の変換を定義していた．しかし，このプロジェクトでは式 13 のような式が記述されていた.

$$/*@\,requires\,\backslash typeof(a)\;==\;\backslash type(T3)[\,];@*/ \tag{13}$$

このような型の演算の比較すべてに対応することは困難であるため，本研究では変換可能な形式を限定して定義している．実験の結果から，全 1359 の JML 式のうち 4 個のみが変換できない型の比較を含んだ式であり，ほとんどの型の比較を含んだ式は定義した形式に従っていたことから，限定的な定義であったとしても十分な有用性を持っていると考える．

\lockset を含んだ式が 4 つ含まれていた．具体的には式 14 のような式である．

$$/*@\, requires\ \backslash max(\backslash lockset)\ <=\ this; @*/ \tag{14}$$

現状，\lockset の変換に対応することはできていない．このような OCL では対応の取れない式の扱いをどのように行うかは今後の課題の 1 つであると考えられる．

最後に，extweka について述べる．extweka 変換できなかった 3 個の JML 式は，すべて配列に対するコレクション演算を含む式であった．これらの 3 個の式は，式 8 のような式であり，教務システムの箇所で述べたように，このような式すべてに対して変換規則を定義し，すべてを自動化することは厳密には対応不可能である．

6.4.2 逆変換の意味的一致率

逆変換の意味的一致率は，全プロジェクトで 93.5%と非常に高い数値を示した．意味的に一致しなかったパターンは以下の 2 つである．

- 参照型の比較に “==” を用いている式
- 元の式の記述が誤っていた式

以下に，これらを具体例を挙げて説明する．

本稿では参照型の変数などを “=” を用いて比較している OCL 式は “equals” メソッドを利用する形に変換している．そのため，元の JML 式で “equals” メソッドを用いずに “==” を用いて比較を行っているものに関しては意味的に一致しない．具体的には，式 15 のような式は，双方向変換によって式 16 のように形式で出力される．

$$/*@\, ensures\ \backslash result\ ==\ seats;\ @*/ \tag{15}$$

$$/*@\, ensures\ \backslash result.equals(seats);\ @*/ \tag{16}$$

この 2 つの式における “\result” と “seats” は共に参照型であるため，双方向変換後は equals メソッドを用いた式が出力されている．しかし，元の式は “==” を用いているため，これらの式は意味的に一致しているとはみなせない．この比較は本来求めている式の意味とは異なる．しかし，OCL から JML への自動変換において，これを厳密に区別することは難しい．現在は，元の演算の情報を変換結果に埋め込む，といった方法での対応が望ましいのではないかと考えている．

また，元の式が間違っているために，変換結果が意味的に一致しなかったパターンも存在する．具体的には，式 17 のような式は，双方向変換によって式 18 のような形式で出力される．

$$/*@\,ensures\,result\ ==\ (0\ <=\ ix\,\&\&\,ix\ <\ DIM\ *\ DIM);\ @*/ \tag{17}$$

$$/*@\,ensures\,\backslash result\ ==\ (0\ <=\ ix\,\&\&\,ix\ <\ DIM\ *\ DIM);\ @*/ \tag{18}$$

この式は，プロジェクト P2-master に含まれてた式であるが，result という変数は，プロジェクト中に存在していなかった．おそらく本来は “\result” と書くべきつもりであったところのバックスラッシュが抜けていたものだと思われる．JML から OCL への変換では，result の文字列をそのまま result へ変換したため，再び JML へ変換した際に “\result” として出力されている．本来であれば，存在していない変数などはエラーとして出力すべきであると考えており，ツールの実装は改善すべき点であると考えられる．

6.4.3 逆変換の構文的一致率

逆変換の構文的一致率は最大で 76.83%，最小で 0%という結果になった．また，全体での構文一致率は 49.74%であり，半分以上の式が本来の構文とは異なる形で出力されていることが確認された．構文的に一致しなかったパターンは以下のものである．

- forall などのループ演算を含む式
- “!=“を含む式
- 条件演算子 “&” を含む式
- 配列を含む式
- シングルクオテーションで囲んだ文字型を含む式
- type メソッドを用いた型の比較を含む式

以下で，これらを詳細に説明する．

実験対象中で多く見られたものは，主に forall などのループ演算を含む式と，“!=” を含む式であった．forall などのループ演算は OCL から JML への変換の際に，元のループと意味的に等価なメソッドを生成し，そのメソッドを JML 側から呼び出す方法で対応している．そのため，元の JML 式が式 19 の場合，双方向変換後の JML 式は式 20 のようになる．

$$/*@\,requires\,(\backslash forall\ Item\ i,j;\ il.contains(i)\ \&\&\ il.contains(j)\ \&\&\ il.indexOf(i)\ !=\ il.indexOf(j);\ !i.getName().equals(j.getName()));\ @*/ \tag{19}$$

$$/*@\,requires\,translatedIteration4(il,ID,il,d);\ @*/ \tag{20}$$

式 20 の translatedIteration4 メソッドはここでは省略しているが，式 19 と意味的に等価なメソッドである．この 2 つの式は構文的には異なるが意味的には等価であり，JML の検査ツールにかけた場合も同様の結果が得られる．しかし，双方向変換後の式本来の意味を確認するためにはメソッドの中身を確認しなければならないことなどから，可読性は元の式と比較して低下している

“!=“を含む式である式 21 は，双方向変換によって式 22 のような形式で出力される．

$$/*@\,invariant\,totalamount\ >=\ 0\,\&\&\,name\,!=\ null\,\&\&\,!name.equals(\text{""});\,@*/ \tag{21}$$

$$/*@\,invariant\,totalamount\ >=\ 0\,\&\&\,(name\ ==\ null)\ ==\ false \\ \&\&\,!(name.equals(\text{""}))\,;@*/ \tag{22}$$

双方向変換の結果，“name != null” が “(name == null) == false” となっていることがわかる．これは意味的には等価であり，JML の検査ツールにかけた場合も同様の結果が得られる．しかし，“!=“を含む式はほとんどすべてのプロジェクトに出現すること，構文一致率 0%のプロジェクト Lenguajes_III に含まれる 65 個の JML 式がすべて “!=“を含む式であったこと，ループ式の場合と違って単に冗長になってしまっているだけであることから，“!=“を含む式は双方向変換後も元の形を保てる変換規則に修正すべきであると考える．

また，条件演算子の “&” を含むものも双方向変換によって構文的に一致しなかった．具体的には，式 23 のような式が，双方向変換によって式 24 のような形式で出力される．

$$/*@\,requires\,0\ <=\ newSmallBlind\,\&\,small\ <\ newSmallBlind;\,@*/ \tag{23}$$

$$/*@\,requires\,0\ <=\ newSmallBlind\,\&\&\,small\ <\ newSmallBlind;\,@*/ \tag{24}$$

この 2 つの式は，どちらもブール型の比較を行っているので，“&” を用いても “&&” を用いても同様の結果が得られる．このため，構文的には一致していないが，実行結果は等価であるので変換そのものは成功していると言える．しかし，整数型のビット毎の AND 演算を行いたい場合には，この変換は失敗しているものであるとみなされる．今回の実験対象の中にはそのような式は存在していなかったため，変換そのものはすべて成功したと考えているが，ビット演算を含む場合には本来の式の意味は失われてしまうと考えられる．OCL で直接的にビット演算に相当する比較を表現することはできないため，そのような式に対しては元の演算がビット演算であった情報を変換結果に埋め込む，といった形で対応する以外の方法での改良を今後の課題の 1 つとして考えている．

配列に関する式の大半も双方向変換によって構文が異なった形で出力される．例えば，配列の長さを返す length メソッドを含んだ式なども本来の構文とは異なった形で出力される．具体的には，式 25 のような式は，双方向変換によって式 26 のような形式で出力される．

$$/*@\,requires\,allCards.length\ ==\ FULL\_DECK;\,@*/ \tag{25}$$

$$/*@\,requires\,allCards.size()\;==\;FULL\_DECK\,;@*/ \tag{26}$$

この式に含まれている allCards は，元の式である式 25 では配列である．しかし，配列は本研究の変換に適用することでコレクション型に変換される．つまり，式 26 における allCards はコレクション型となっている．そのため，式 25 では要素数を返すために length，式 26 では size メソッドが用いられている．実験対象として用いたプロジェクトに含まれていた JML 式についてはこの変換方法を適用することで問題なく変換を行なうことができたが，配列の要素数を固定しているような場合には正しく変換できない．これについては，OCL 中で直接的にコレクションの要素数を限定することはできないため，元の情報を埋め込んでおいて size メソッドの代わりに直接配列のサイズを埋め込むなどの形で対応する方法が考えられる．

元の式がシングルクオテーションで囲った文字型を含む場合も，双方向変換の結果は元の式とは構文的に一致しない．具体的には，式 27 のような式は，双方向変換によって式 28 のような形式で出力される．

$$/*@\,ensures\,0\;<=\;\backslash result\;-\;'a'\;\&\&\;\backslash result\;-\;'a'\;<=\;25;\;@*/ \tag{27}$$

$$/*@\,ensures\,0\;<=\;\backslash result\;-\;\text{“}a''\;\&\&\;\backslash result\;-\;\text{“}a''\;<=\;25;\;@*/ \tag{28}$$

文字型も文字列型も OCL では区別できないために同一に扱う．そのため双方向変換の結果，元の式ではシングルクオテーションで囲った文字型として扱っていたものは，ダブルクオテーションで囲った 1 文字の文字列型となる．これも意味的には等価であるが，構文的には異なっているものとみなされる．しかし，可読性の観点などからもほとんど影響はないため，ほとんど問題はないと考えられる．

type メソッドを用いた型の比較を含む式も双方向変換後は元の構文と異なる形で出力された．具体的には，式 29 のような式は，双方向変換によって式 30 のような形式で出力される．

$$/*@\,invariant\,\backslash typeof(itemlist)\;==\;\backslash type(Item);\;@*/ \tag{29}$$

$$/*@\,invariant\,itemlist.getClass().equals(Item);\;@*/ \tag{30}$$

6.5 考察

ここでは，実験結果に関する考察を述べる．

6.5.1 実験結果

まず，JML から OCL への変換率は合計で約 93.5%であった．また，残り 6.5%はほとんどが配列に対するループ演算を含む式であった．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 が存在している．本研究では，4 章の変換規則で述べた通り，配列は Sequence として変換している．単純に配列型の変数を参照しているようなものに関

しては変換規則を定義しているが，配列に対するループ演算のような式には対応できていない．このような式の変換への対応方法として，構文の形式を限定して変換規則を定義する，といった方法が考えられるが，記述の自由度とトレードオフの関係にあるため，どこまで制限を設けるか，といった点が課題であると考えている。

逆変換の意味的一致率は約 93.5%であった．逆変換に失敗したものの中には，開発者の記述ミスなども含まれていた．また，意味的な不一致が起こった場合，現状のツールではその変換ミスを検出できない．そのため，意味的な変換ミスの検出は人手に頼らざるを得ない．Esc/java2 などの JML 用の検査ツールに双方向変換前後の式を適用した結果を確認することで意味的不一致の検出は行えるが，実用上の観点からは有用な方法であるとは考えにくく，今後対応すべき課題の一つであると考える．

プロジェクト全体での逆変換による構文的一致率は 46.5%であった．半分以上の式は双方向変換によって構文的には異なった式で出力された．この中には，“!=” を含む式のように単に冗長になってしまい可読性を低下させてしまうパターンが存在していることや，場合によっては変換失敗となるパターンが存在する可能性があることから，今後改善していくべき点であると考えている．

次に，本質的に対応不可能な式について述べる．変換できなかった式のうち，本質的に変換不能な式は約 9.86%で，全 JML における割合は 0.515%と非常に低い割合となっている．本質的に対応不能な JML 式の割合が大きすぎる場合は，JML から OCL への変換を実現することは現実的な解ではないように思われる．しかし，本研究での調査結果から本質的に対応不能な JML は全体の 1%にも満たない割合であった．このことから，OCL，JML 間での双方向変換を行うこと自体に問題はないと考えられる．

双方向変換によって，意味はほぼ同等であるが，式の形式が異なってしまう JML について述べる．本研究では，配列の変換はこれに当たる．配列は，本研究では OCL の Sequence に変換される．そして OCL の Sequence は独自定義したクラスである MySequence に変換される．このため，元々配列であった JML は双方向変換によって，独自定義した MySequence となってしまう．しかし，この変換によって本来の配列であったという情報は失われるが，JML の検査ツールで実行した場合の出力結果は元の式の場合と変わらない．このことから，この変換方法は妥当であると考える．

6.5.2 内的妥当性の脅威

ここでは，内的妥当性の脅威について述べる．

実験に用いたプロジェクトのうち，在庫管理プログラムと教務システムに付加されている JML は，研究グループのメンバーによって記述されたものである．そのため，自分たちの実験に都合の良い JML 記述が行われたのではないかと考えられかねない．しかし，これらの JML は本研究における実験対象とすることを目的として記述されたものではない．このことから，本ツールに都合の良い記述を過去の時点で行っているということは考えにくいことは明らかである．また，これらの JML は複数人で記述されたものであり，JML 記述自体も特定の形に偏っておらず，様々な形式の JML が存在しているため，実験対象として用いるのには適当であると判断できる．これらの点から，実験対象と

して，これらのプロジェクトを用いることは妥当であると考えられる．

また，これまでの過去の研究グループでの相互変換についての研究における実験 [20, 21] では，グループ内で作成したプロジェクトである在庫管理プログラムや教務システムを用いているのみであった．そのため，自分たちのツールの変換に都合の良いデータを恣意的に用意しているのではないか，ということが問題点として指摘されていた．本研究では実験対象として，複数のオープンソースプロジェクトを用いている．これらのプロジェクトに含まれる JML は変換ツールとは関係の無いものであり，これまで指摘されていた実験対象の一般性に関する問題点は改善されたと考える．

6.5.3 外的妥当性の脅威

ここでは，外的妥当性の脅威について述べる．

本実験において利用している実験対象はオープンソースのプロジェクトではあるが，プログラムの属性に偏りがあると考えられかねない．しかし，実験対象として利用したオープンソースプロジェクトは 7 つあり，またそれらの 7 つのプロジェクトは特定の開発者によって作成されたものではなく，それぞれが様々な開発グループの開発者たちによって作成されたものである．このことから，実験に用いたオープンソースプロジェクトに含まれるプログラムの属性に大きな偏りがあるとは考えにくく，妥当性は保たれていると考えられる．

また，その他の外的妥当性の脅威として，実験結果の計測方法，具体的には JML から OCL および OCL から JML への変換が意味的に一致していると判断する方法が挙げられる．現状，OCL と JML の間の対応関係が公式に定義されてはおらず，研究者が独自に定義しているに過ぎない．しかし，この 2 つの言語の間での変換などは，我々の研究グループも含めて様々な研究グループで研究されていることから，既存研究における定義などを流用することは有効だと考える．本研究では，既存の規則などを流用できる箇所は流用しており，変換の妥当性を判断する際にも様々な既存研究における定義などを判断材料として利用している．また，OCL と JML のリファレンスはそれぞれ存在しており，それらも判断材料として利用している．これらのことから，実験結果の計測方法は妥当であると考える．

7 あとがき

本稿では，OCL・JML 間の双方向変換手法を提案し，実装を行った．これまでの研究では，双方向変換を行うにあたりいくつかの問題点が存在していたため，それらの問題点の改善を行っている．また，それらの改善を含めた実装の有効性を評価するための評価実験を行った．評価実験として，実装したツールを複数のプロジェクトに適用した．実験の結果から，JML から OCL への変換は 93.5%の割合で成功し，その逆変換結果の意味的一致率は 93.5%であることを確認した．しかし，構文的な一致率は 46.5%と半分以下の結果となった．次に，プロジェクトに含まれる変換不能な式の種類についての調査を行った．その結果，手動でも JML から OCL への変換ができない式は全体のうち，約 0.52%であった．本研究での調査結果から，本質的に対応不能な JML は全体の 1%にも満たない割合であり，OCL，JML 間での双方向変換を行うこと自体に問題はないと考えられる．また，変換に要する時間は非常に短く，実用的な範囲で変換が行えることが確認できた．

今後の課題としては，構文的な一致率を高めるために変換規則に改変を加えることを考えている．双方向変換によって，半分以上の式が本来の構文と異なった形式で出力されており，可読性の観点から考えるとより高い構文的一致を実現できる変換規則が望ましいと考えられる．また，現状対応しているものが事前条件，事後条件，不変条件の 3 つのみであるので，JML のその他の構文にも対応することを検討している．例えば，assignable キーワードや signals キーワードへの対応方法を考察中である．また，現状のツールは Xtext を用いた M2T (Model to Text) な手法で実装されている．今後は新たな実装方針として，M2M (Model to Model) な手法での双方向変換の実現を考えている．その他に，JML 以外の言語と OCL 間での双方向変換を実現することも今後の課題の 1 つとして考えている．具体的には，SPEC#などの制約記述言語と OCL 間での双方向変換を実現していくことを検討している．

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 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，終始熱心かつ丁寧なご指導を頂きました 同 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して多大なるご助言，ご助力を頂きました 同 井垣 宏 特任准教授に深く感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 同 肥後 芳樹 助教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.

最後に，本学基礎工学部所属時より現在に至るまで，講義，演習，実験等を通じてお世話頂きました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] 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. Hucking, 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/spec/OCL/2.0`.

[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] M. Rodion and R. Alessandra. Implementing an OCL to JML translation tool. 電子情報通信学会技術研究報告, 第 106 巻, pp. 13–17, 2006.

[8] A. Hamie. Translating the object constraint language into the java modelling language. In *Proc. of the 2004 ACM symposium on Applied computing*, pp. 1531–1535, 2004.

[9] C. Avila, Jr. G. Flores, 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.

[10] K. Hanada, K. Okano, S. Kusumoto, and K. Miyazawa. Practical application of a translation tool from uml/ocl to java skeleton with jml annotation. In *14th International Conference on Enterprise Information Systems (ICEIS2012)*, pp. 389–394, 6 2012.

[11] 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. *IEICE technical report*, Vol. 110, No. 169, pp. 53–58, 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と評価. *IPSJ SIG Technical Report*, Vol. 2010-SE-170, No. 19, nov 2010.

[13] Object Management Group. OMG Modeling Specifications. http://www.omg.org/.

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

[15] N. Medvidovic, A. Egyed, and D. S. Rosenblum. Round-trip software engineering using uml: From architecture to design and back, 1999.

[16] S. Sendall and J. Küster. Taming model round-trip engineering. In *Proc. of Workshop Best Practices for Model-Driven Software Development*, pp. 1–13, 2004.

[17] A. Henriksson and H. Larsson. A definition of round-trip engineering, 2003.

[18] 今関雄人, 高田眞吾. ソフトウェアの動的モデルた着目したラウンドトリップェンジニアリングの支援 (動向/テスト/ツール). 情報処理学会研究報告, Vol. 2007, No. 97, pp. 47–54, 9 2007.

[19] 榛葉浩章, 花田健太郎, 岡野浩三, 楠本真二. 制約記述言語 ocl と jml のモデル駆動開発技法に基づいた双方向の変換手法の提案. 電子情報通信学会技術研究報告, 第 111 巻, 3 2012.

[20] K. Hanada, H. Shinba, K. Okano, and S. Kusumoto. Implementation of a prototype bi-directinal translation tool between ocl and jml. *International Workshop on Informatics 2012 (IWIN2012)*, pp. 121–127, 9 2012.

[21] K. Hanada, H. Shimba, K. Okano, and S. Kusumoto. A bi-directional translation tool between ocl and jml considering reverse translation. In *The 4th International Workshop on Empirical Software Engineering in Practice (IWESEP2012), poster presentation*, 10 2012.

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

[23] Object Management Group. Queries/Views/Transformations. http://www.omg.org/spec/QVT/1.1/PDF/.

[24] F. Allilaire, I. Kurtev, and J. Bezivin. Atl: A model transformation tool. *Science of Computer Programming*, Vol. 72, No. 1-2, pp. 31–39, 2008.

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

[26] 中島震. プログラム簡易検証ツール esc/java2. コンピュータソフトウェア, Vol. 24, No. 2, pp. 22–27, 2007.

[27] Eclipse Foundation. Xtext - Language Development Framework. http://www.eclipse.org/Xtext/.

[28] 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.

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

[30] M. G. Lee I. S. Bajwa, B. Bordbar. OCL Constraints Generation from Natural Language Specification. In *14th IEEE The Enterprise Computing Conference (EDOC 2010)*, pp. 204–213, 2010.

[31] Y. Cheon, C. Avila, S. Roach, and C. Munoz. Checking design constraints at run-time using OCL and AspectJ. *International Journal of Software Engineering*, Vol. 3, No. 1, pp. 5–28, 2009.

[32] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, Vol. 4735, pp. 436–450, 2007.

[33] S.M.A. Shah, K.Anastasakis, and B. Bordbar. From uml to alloy and back. In *6th Workshop on Model Design, Verification and Validation (MODEVVA 09) published in ACM International Conference Proceeding Series*, Vol. 413, pp. 1–10, 2009.

[34] A. Razavi and K. Kontogiannis. Partial evaluation of model transformations. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE 2012, pp. 562–572, Piscataway, NJ, USA, 2012. IEEE Press.

[35] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE 2012, pp. 540–550, Piscataway, NJ, USA, 2012. IEEE Press.

[36] T. Kapteijns, S. Jansen, S. Brinkkemper, H. Houet, and R. Barendse. A comparative case study of model driven development vs traditional development: The tortoise or the hare. *4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI"*, pp. 22–33, 2009.

[37] S. Teppola, P. Parviainen, and J. Takalo. Challenges in deployment of model driven development. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pp. 15 –20, 9 2009.

[38] A. Tanaka and O. Takahashi. Experimental transformations between business process and soa models. *International Workshop on Informatics 2011*, pp. 104–112, 9 2011.

[39] J. Warmer and A. 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.

[40] G. T. Leavens and E. Poll and C. Clifton and Y. Cheon and C. Ruby. JML Reference Manual. http://www.dc.fi.udc.es/ai/tp/practica/jml/JML/docs/jmlrefman/jmlrefman/.

[41] 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.

[42] 文部科学省. IT Spiral. http://it-spiral.ist.osaka-u.ac.jp/.

[43] github. https://github.com/.