特別研究報告

題目

OCL から JML への変換ツールの Eclipse プラグイン化と教務システムを対象とする適用実験

指導教員
楠本　真二　教授

報告者
花田　健太郎

平成 23 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 22 年度 特別研究報告

OCL から JML への変換ツールの
Eclipse プラグイン化と教務システムを対象とする適用実験

花田　健太郎

内容梗概

近年 MDA(Model Driven Archtecture) 関連技術の発展により，UML などの上位設計仕様記述から実装レベルのプログラム言語など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(JavaModelling Language) への変換技術も研究が行われている．従来研究では OCL から JML への変換において，データの集まりを扱うコレクションに関するいくつかの重要な機能，とりわけ，iterate 演算に非対応であった．研究グループではこの問題に対し，生成される Java スケルトンコードに OCL の iterate 演算と意味的に等価なメソッドを記述するという方法で対応し，その手法に基づいた OCL からの JML への変換ツール (以後 OCLtoJML) を実装した．

研究グループの開発において，OCLtoJML は Eclipse プラグインとして実装することを目的としていたが，変換機能の実装までにとどまっており，プラグイン化は行われていない．また OCLtoJML の問題点として，入力する OCL 文に外部ライブラリが利用できないという点が挙げられる．OCLtoJML では，OCL 文の意味解析には入力された UML 中のクラスの情報を用いて行われるが，一般的に UML 内に外部ライブラリの詳細なクラス情報を持たせることがないため，このような制限が存在してしまう．さらに OCLtoJML の実用性を示すための評価実験として小規模プロジェクトに対しての適用は行われていたが，大規模な実プロジェクトに対しての適用は行われていなかった．

本研究では研究グループの目的の達成および問題改善のため，まず OCLtoJML の Eclipse プラグイン化を行った．また外部ライブラリのメソッドには，Eclipse プラグインとして実装されている Java から UML への変換を行う既存ツールを OCLtoJML 用に拡張し，生成された外部ライブラリのクラス情報を入力 UML に追加することで対応した．さらに OCLtoJML の性能評価を行うために，実プロジェクトとして，和歌山大学の教務システムに対して OCLtoJML の適用実験を行った．従来は 3 クラス 20 メソッドを用いて適用実験を行ったのに対し，本研究では 60 クラス 384 メソッドに対して OCLtoJML を適用し，変換速度と生成される JML の品質評価を行った．その結果，十分に実用的な時間で変換が完了することが確認され，また生成された JML は単体テストを行うにあたり十分な品質を持っていることが示された．

主な用語

モデル変換, OCL, JML, Java, UML, Eclipse プラグイン

目次

1 まえがき **1**

2 背景 **3**

2.1 Design by Contract . . . 3
2.2 UML . . . 3
2.3 OCL . . . 4
2.4 JML . . . 5
2.5 JML Runtime Assertion Checker . . . 7
2.6 OCLtoJML . . . 7
2.7 関連研究 . . . 8

3 実装 **9**

3.1 OCLtoJML の Eclipse プラグイン化 . . . 9
3.1.1 目的 . . . 9
3.1.2 Papyrus UML . . . 9
3.1.3 画面構成 . . . 10
3.1.4 入力仕様 . . . 10
3.1.5 開発手順の概要 . . . 11
3.2 OCLtoJML の対応クラスの拡張 . . . 12
3.2.1 目的 . . . 12
3.2.2 実現方法 . . . 13
3.2.3 適用例 . . . 15
3.2.4 開発手順の概要 . . . 16

4 評価実験 **18**

4.1 実験概要 . . . 18
4.2 実験対象 . . . 19
4.3 実験準備 . . . 19
4.3.1 対象プロジェクトの Java コード . . . 20
4.3.2 JML . . . 21
4.3.3 OCL . . . 21
4.3.4 UML . . . 22
4.3.5 テストケースクラス . . . 22
4.4 計測結果 . . . 22
4.4.1 変換時間 . . . 22

4.4.2 変換率 22
4.4.3 再現率 24
4.5 考察 28

5 あとがき **29**

謝辞 **30**

参考文献 **31**

図目次

1 OMG の 4 層モデル 4
2 JML 記述例 6
3 OCL 記述例 6
4 OCLtoJML 概要 7
5 OCLtoJML の画面構成 10
6 OCLtoJML の入力画面 11
7 開発手順の概要 12
8 外部ライブラリのメソッドを用いた OCL の構文解析 13
9 外部ライブラリのスーパークラスのメソッドを用いた OCL 13
10 Java2UML の変換対象選択ダイアログ 15
11 外部ライブラリ作成も含めた開発手順の概要 16
12 教務システムのクラス図 1 19
13 教務システムのクラス図 2 20
14 JML と hibernate のアノテーションが混在した例 21
15 変換できない JML の記述例（クラスリテラル） 23
16 変換できない JML の記述例（配列） 24
17 適用範囲の拡張により変換可能になった OCL 24
18 適用範囲の拡張により変換可能になった OCL から生成された JML 25
19 iterate 演算を含む元の JML 26
20 iterate 演算を含む OCL 26
21 iterate 演算を含む OCL から変換された JML 26
22 意味的に一致するパターンの元の JML 27
23 意味的に一致するパターンの OCL 27
24 意味的に一致するパターンの OCL から変換された JML 27

表 目 次

1 JML と OCL 間の対応例 5
2 既存手法と OCLtoJML の比較 8
3 OCLtoJML による OCL から JML への変換率 22
4 OCL で記述できなかった条件式の内容 23
5 OCLtoJML による JML の再現率 25

1　まえがき

近年 MDA(Model Driven Architecture)[1] 関連技術の発展により，UML などの上位設計仕様記述から実装レベルのプログラム言語など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 Modelling Language)[6] への変換技術も研究が行われている．OCL は UML 記述に対し，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)[7] によって標準化されている．より実装に近い面での制約記述言語として，Java プログラムに対して JML が提案されている．JML，OCL ともに Design by Contract[8] の概念に基づきクラスやメソッドの仕様を与えることができる．

OCL から JML への変換については Hamie が文献 [9] において構文変換技法に基づいた OCL から JML への変換法を提案しており，Rodion と Alessandra らが文献 [10] において，Hamie の研究の拡張とツールの実装を示している．また，Avila らが文献 [11] において型の扱いなどについて改善を示しているものの，いずれの方法もデータの集まりを扱う Collection の対応が不十分であり，iterate 演算における一部の演算のみ対応している．しかし，iterate 演算 は広く用いられる演算であるため，対応すべき問題である．

著者の所属する研究グループは，iterate 演算を含む OCL 記述が付加されたクラス図に対して，JML 記述への変換ツール（以後 OCLtoJML）を実装し [12]，小規模プロジェクトを対象として評価実験を行った [13]．しかし，既存の OCLtoJML に関しては以下の 3 点の問題点が挙げられる．また本研究における外部ライブラリとは，Java の標準ライブラリやサードパーティの作成したライブラリを指すものとする．

1. OCLtoJML は Eclipse プラグインとして設計されているが，実装は変換機能のみでプラグインとしては未実装
2. 入力する OCL 文に外部ライブラリの情報が利用不可
3. 大規模な実プロジェクトに対する適用実験が行われておらず，性能評価の指標としては不十分

そこで本研究では研究グループの目的の達成および問題解決のため，以下の 3 点を行った．

1. OCLtoJML を Eclipse プラグインとして実装し，組み合わせて使用するモデリングツールとして Papyrus UML[14] を用いることで一連の設計を Eclipse 上のみで行うことが可能な環境を実現
2. 外部ライブラリのクラス情報を UML クラス図に追加するためのツールの作成
3. 実プロジェクトとして，ある大学の教務システムに対して OCLtoJML の適用実験を行い，実用的な時間で変換が完了することと，生成された JML が十分な品質を持っていることを確認

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 章ではプラグイン化および外部ライブラリを利用するために作成したツールについて説明する．4 章では OCLtoJML の評価実験についての計測内容，方法，結果を述べ，考察を行う．最後に 5 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 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り，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．また，事前条件，事後条件の他に不変条件が挙げられる．これはクラスの実行が正常に行われている間は常に真理値が真である条件を記述したものである．これらによって，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)[15] 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)[16] 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．

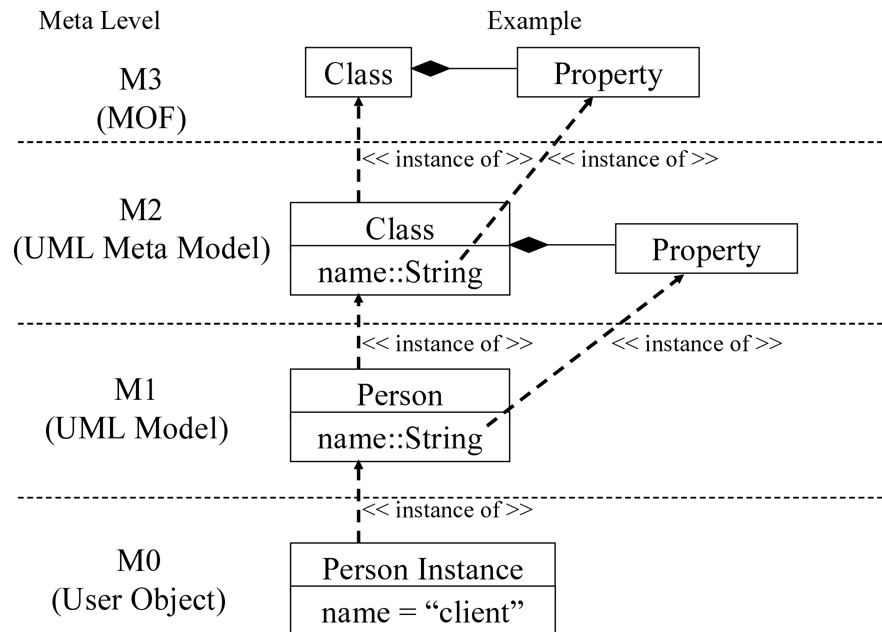

図 1: OMG の 4 層モデル

2.3　OCL

OCL 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

表 1: JML と OCL 間の対応例

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

Language(ATL)[17] なども，OCL に基づいて構築されている．

2.4 JML

JML 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 Run-time 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[18]，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[19] 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．

```
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

```
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

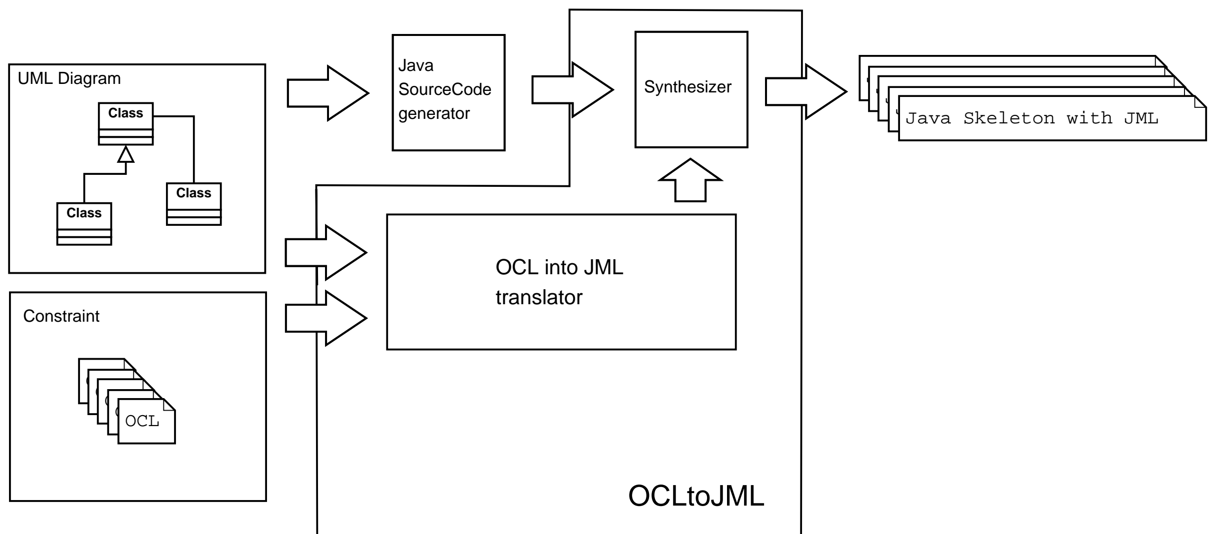

図 4: OCLtoJML 概要

2.5 JML Runtime Assertion Checker

JMLrac(JML Runtime Assertion Checker)[20] 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である．本研究では JML が付加された Java プログラムのコンパイルに jml4c を用いた．また，jml4c によって生成された実行ファイルを jml4rt を用いて実行した．

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．本研究で用いている JMLrac は jml4rt である。

2.6 OCLtoJML

OCLtoJML は，我々の研究グループが作成したツールであり，OCL で記述の行われた制約を，JML で記述の行われた制約へ変換することができる．入出力仕様を以下に示す.

入力仕様

表 2: 既存手法と OCLtoJML の比較

Feature	OCLtoJML	Moiseev と Russo	Hamie
基本演算	√	√	√
Collection	√	√	-
Iterate (forall etc.)	√	√	√
Iterate (collect)	√	√	-
Iterate (iterate)	√	-	-
Structures (Set etc.)	√	√	√
Structures (tuple)	√	√	-
Message	-	-	-

UML クラス図のファイルパス，OCL のファイルパス，UML と等価な Java スケルトンコードが格納されたディレクトリのパスの 3 つを入力として要求する．ツールの入力となる UML のクラス図は，XML で記述が行われたものであり，その形式は，OMG によって定められている UML の国際標準規格である UML2[21] に準拠したものとする．

出力仕様

入力で与えた OCL に記述された制約と意味的に等価な，JML で記述された制約が付加された Java スケルトンコードを出力する．

UML の作成，OCL の作成，Java スケルトンコードの作成は OCLtoJML の対象外である．

2.7 関連研究

文献 [10] では，OCL から JML への変換法とツールの実装を示している．文献 [9] において構文変換技法に基づいた OCL から JML への変換法を提案している．しかしながら，いずれの方法も Collection 演算の対応が不十分であり，とりわけ iterate 演算の対応がされていない．また文献 [9] における構文変換技法は概論のみであり，精緻な変換方法は触れられていない．

OCLtoJML は，これらの既存の変換ツールにおいて未対応だった iterate 演算に対応している．具体的には，iterate 演算に対応したメソッドを Java プログラム中に挿入し，そのメソッドを JML から呼び出すことで変換を実現している．

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．

3 実装

3.1 OCLtoJML の Eclipse プラグイン化

研究グループで作成した OCLtoJML の Eclipse プラグイン化を行った．

3.1.1 目的

OCLtoJML を Eclipse プラグイン化することによって，システムの開発効率を向上させることを目的としている．一般的に Java の開発には Eclipse を用いることが多いので，Eclipse 上で UML や OCL の記述が行うことができ，さらにそこから JML の付加された Java スケルトンコードが得られるといった一連の流れを全て Eclipse 上で行うことで，開発効率が向上すると考えられる．

現在の OCLtoJML では OCL から JML への変換部分は作成されているが，Eclipse プラグインとしての実装は行われていない．Eclipse プラグイン化を行った場合，Eclipse 上で OCLtoJML が使用できるようになるのはもちろんのこと，既存の Eclipse プラグインとして実装されているモデリングツールと組み合わせて使用することで，OCLtoJML では実装されていない設計モデルの作成機能も含めた 1 つの開発環境を得ることができる．この開発環境を用いて Java によるソフトウェアの開発を行うことは上で挙げたような利点を得られるものだと考えられる．

3.1.2 Papyrus UML

OCLtoJML と組み合わせて使用するモデリングツールには Papyrus UML を選択した．Papyrus UML はオープンソースの EPL ライセンスに準じる Eclipse 用の UML モデリングツールであり，以下のような機能を備えている．

- クラス図の描画
- ユースケース図の描画
- デプロイ図の描画
- コンポジット図の描画
- コンポーネント図の描画
- アクティビティ図の描画
- シーケンス図の描画
- Java スケルトンコードの自動生成

Eclipse プラグインとして実装されているモデリングツールは他にも様々なものが存在するが，UML ファイルの XML 形式が UML2 に準拠していないものもある．また，作成した UML を Java のスケ

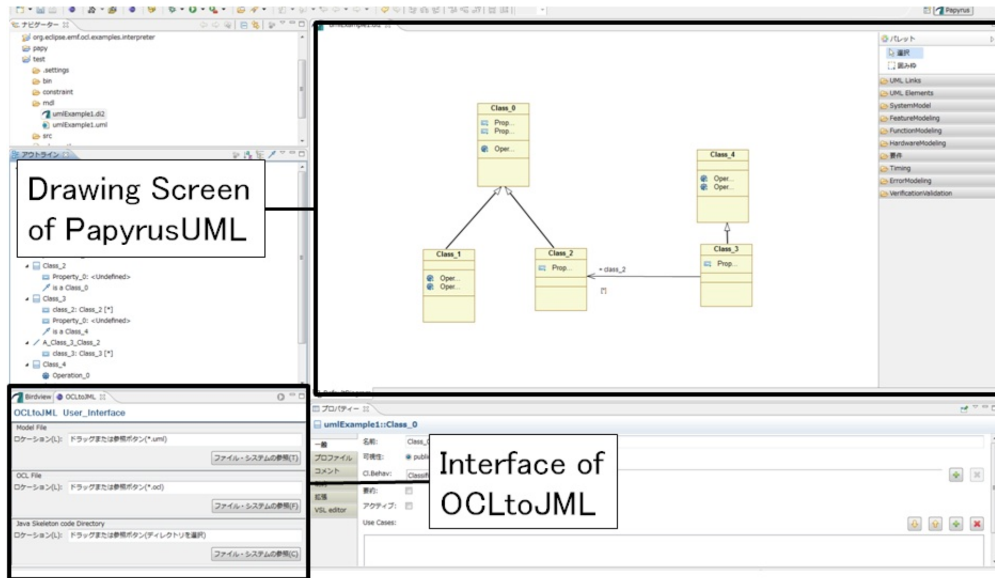

図 5: OCLtoJML の画面構成

ルトンコードに変換する機能を備えているツールは少なかった．Papyrus UML は UML から Java スケルトンコードを出力する機能を持ち，かつ出力する UML の形式が UML2 に準拠している．したがって，2.6 節で述べた OCLtoJML の入力仕様を満たすツールとして，Papyrus UML が適していると考えた．

3.1.3　画面構成

図 5 に Papyrus UML と併せた OCLtoJML の画面構成を示す．右上の Drawing Screen of Papyrus UML の部分は Papyrus UML の UML 描画画面である．PapyrusUML は直接は Papyrus UML において利用する UML のファイルである di2 ファイルを画面に表示・編集を行うが，di2 ファイルへ加えた変更は di2 ファイル保存時に自動的に uml ファイルに反映されるため，実質的には uml ファイルを変更することと同じである．

また左下の Interface of OCLtoJML が OCLtoJML の入力インターフェースの部分である．ファイルの入力，実行はこの部分から行う．

3.1.4　入力仕様

ファイルの入力は図 6 のようにファイルシステムの参照，またはパッケージエクスプローラーからのドラッグ&ドロップで行う．Model File には拡張子が.uml のファイル，OCL File には拡張子が.ocl のファイル，Java Skeleton code Directory にはディレクトリをドラッグ&ドロップ可能で，それぞれ要求されている拡張子以外のファイルやディレクトリをドラッグ&ドロップしても反映されない．3

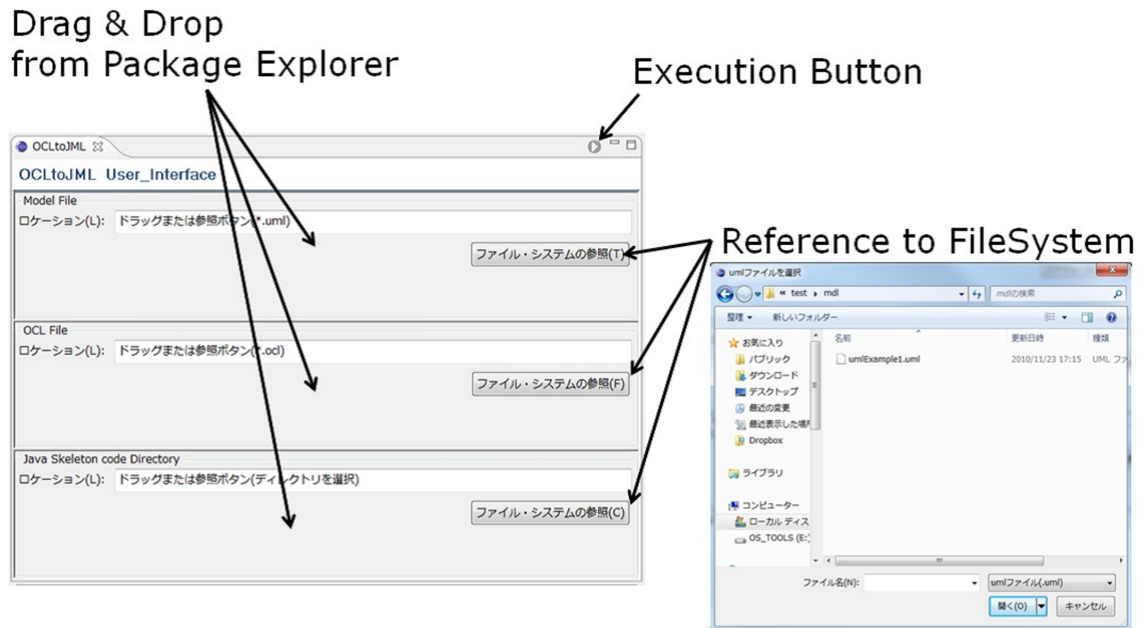

図 6: OCLtoJML の入力画面

か所全ての入力を行うと実行ボタンが有効になるので，実行ボタンを押せば入力されたそれぞれのファイルとディレクトリのパスが変換部分に入力として与えられ，変換を開始する．

3.1.5 開発手順の概要

一連の開発手順は以下のとおりである．

1. Papyrus UML で UML クラス図を作成.
2. Java スケルトンコードを UML クラス図より Papyrus UML のコード生成機能を用いて生成.
3. UML に付加する OCL 文のテキストを作成.
4. UML，OCL，Java スケルトンコードを格納したディレクトリの 3 つをそれぞれ入力する．
5. 正しく 3 か所に入力が行われれば，実行ボタンが有効になるので実行ボタンを押して変換を開始する．
6. OCL と UML から OCL の構文解析を行い，OCL を JML へ変換し，その JML を Java スケルトンコードに付加する．

図 7 と一連の開発手順からわかるように，OCLtoJML をプラグイン化したことにより，これらの一連の流れは全て Eclipse 上で行える．これは Java によるシステムの開発効率の向上に繋がると考えられる．

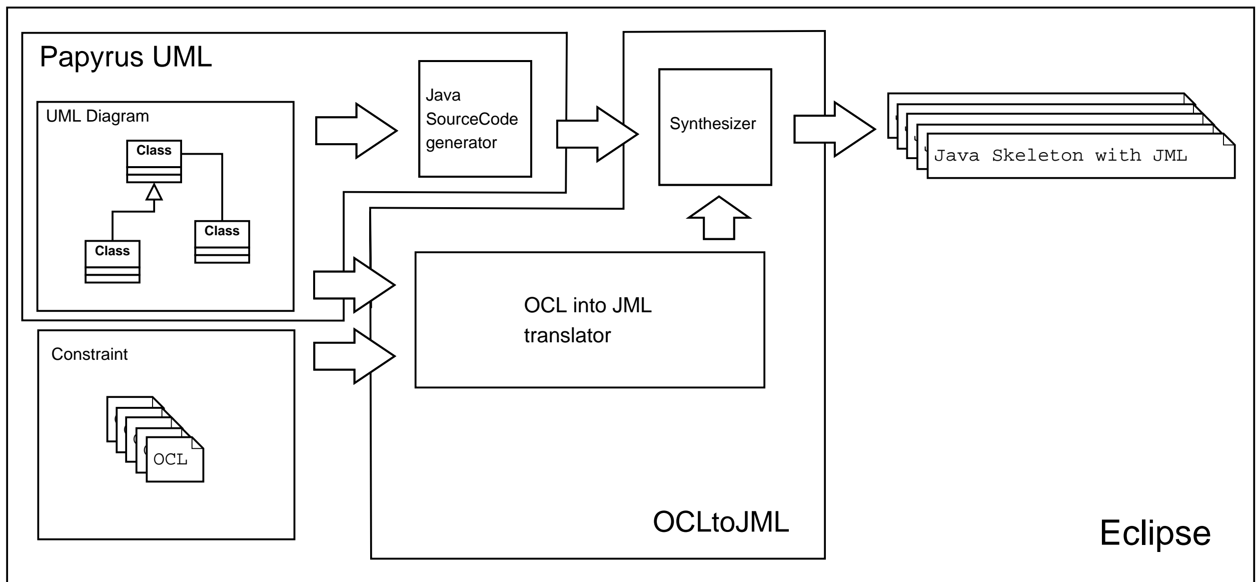

図 7: 開発手順の概要

3.2　OCLtoJML の対応クラスの拡張

3.2.1　目的

外部ライブラリの情報に適用可能な OCLtoJML を実現することによって OCLtoJML の実用性を高めることを目的としている．

既存研究における OCLtoJML では外部ライブラリの情報を OCL に使用することはできなかった．これは OCLtoJML の構文解析の方法に原因がある．OCLtoJML では，OCL の意味解析を行う際にメソッドの戻り値の型などを与えられた UML の中から決定しようとする．一般的に UML 中に外部ライブラリのクラス情報まで詳細に記述されることはない．そのため，変換対象の OCL に外部ライブラリのメソッドは使えない．しかし，外部ライブラリの情報が UML 中に含まれていれば，外部ライブラリを使用することは容易に実現可能であることがわかる．図 8 に外部ライブラリの情報を含んだ UML を用いた構文解析の簡単な例を示す．仮に，図 8 において String クラスが定義されていなければ，length() メソッドの戻り値の型を取得できずにエラーを出力してしまう．しかし，String クラスのメソッドの情報が存在していることにより，構文解析を正しく行うことが可能になっている．

また，外部ライブラリの情報を UML に保持させる際の注意点として，必要最低限の情報のみを持たせて，不必要な外部ライブラリのクラス情報を UML に持たせてはならないという点が挙げられる．なぜなら，外部ライブラリのクラス情報が大きすぎる場合，その情報がメモリを大量に消費し，メモリエラーを起こしてしまうためである．このような理由から保持する外部ライブラリは必要最低限であることが望ましい．

ここで OCLtoJML が構文解析に必要とする外部ライブラリの情報である以下の 2 点について簡単

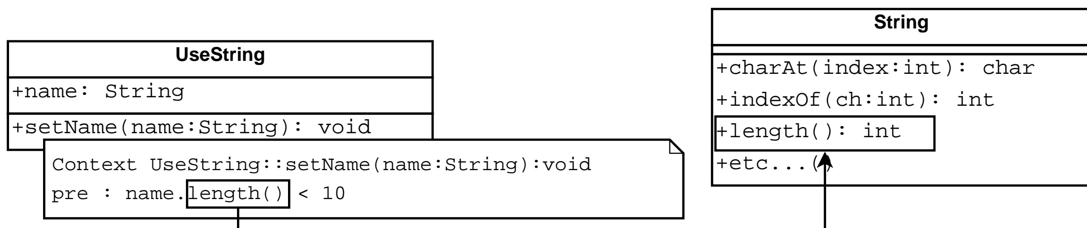

図 8: 外部ライブラリのメソッドを用いた OCL の構文解析

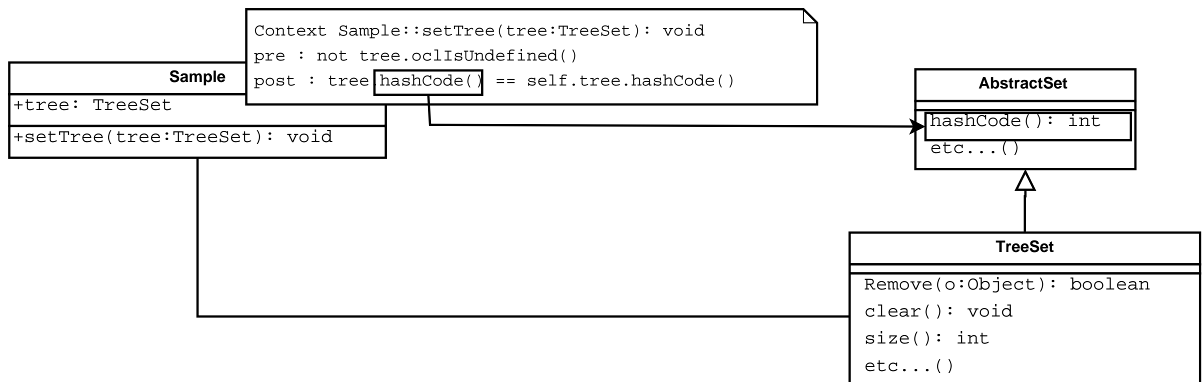

図 9: 外部ライブラリのスーパークラスのメソッドを用いた OCL

に説明する．

- 使用する外部ライブラリのクラスの情報が記述されている UML
- 使用する外部ライブラリのクラスが継承しているクラスの情報が記述されている UML

ここで使用するクラスだけでなく，そのクラスが継承しているクラスの情報まで UML に持たせる理由を具体的な例（図 9）を用いて説明する．

クラス Sample は TreeSet を保持しているが AbstractSet は保持していない．しかし，図 9 の事後条件のようにスーパークラスで定義されているメソッド hashCode() を用いた事後条件を定義することは可能である．このようにスーパークラスのメソッドを用いて OCL を記述することは文法的に可能なので，使用する外部ライブラリのクラスが継承しているクラスの情報まで必要になる．

3.2.2 実現方法

本研究では，外部ライブラリの情報を生成するために既存の Java から UML への変換ツールである TOPCASED[22] の Java2UML[23] を利用した．まずこの Java2UML の仕様と UML の規格を以下に示す．

Java2UML の入力仕様

変換対象として，Java プロジェクトを 1 つ選択する．

Java2UML の出力仕様

選択されたプロジェクトに含まれる Java ファイル全ての情報を持った UML を出力する．この UML は UML2 に準拠している．そのため Java2UML によって生成された UML クラス図は Papyrus UML で編集することが可能であり，OCLtoJML の入力に用いることも可能である．

これらの Java2UML の仕様の問題点を以下に示す．

Java2UML の入力仕様の問題点

クラス単位やパッケージ単位でのリバースエンジニアリングは行えない．

Java2UML の出力仕様の問題点

出力された UML には，元の Java ファイルの関連や継承のほとんどが正しく反映されているが，現在の仕様ではインナークラスや列挙型には対応できていないため，インナークラスの型の変数を保持するクラスとインナークラスの間の関連や，列挙型の変数を保持するクラスと列挙型の間の関連は正しく反映されない．

出力仕様の問題点に関しては，本研究においては，ただしく反映されない部分は Papyrus UML を用いて手動で修正を行った．

入力仕様に関しては，現在の仕様のままツールを利用する場合，入力がプロジェクト単位であるため，新たなプロジェクトを作成し，手動で必要最低限の外部ライブラリのクラスを作成したプロジェクトにコピーし，そのプロジェクトを変換対象としてツールに入力するという流れで変換を行うことになる．しかし，外部ライブラリの入力要件を満たすためには外部ライブラリのクラスの継承関係を確認して，ファイルを選択する必要がある．また毎回ワークスペースに手動で必要なファイルを 1 つずつコピーしなければならない．これらの作業を全て手動で行うのは煩雑である．そこで，Java2UML を拡張することでこのような作業を簡潔にする．以下に拡張点を挙げる．

- 変換対象の拡張
  変換対象を 1 つのプロジェクトではなく，クラス，パッケージ，プロジェクトから複数選択できるように拡張した．この拡張により，毎回ワークスペースにプロジェクトを作成してファイルをコピーするといった作業が必要なくなり，実行時にファイル選択を行うだけで変換できるよう改良された．

- 継承関係の自動確認機能
  選択したファイルが外部ライブラリの要件である継承関係まで正しく満たして選択されていることをツールが確認し，不足している場合は不足しているクラスを全て出力して，ユーザーに引き続きファイル選択を行わせる仕様に変更した．ここで上で挙げた変換対象の拡張を行っていなかった場合，追加でファイルの選択を行う際に必要最低限のファイルだけでなくプロジェ

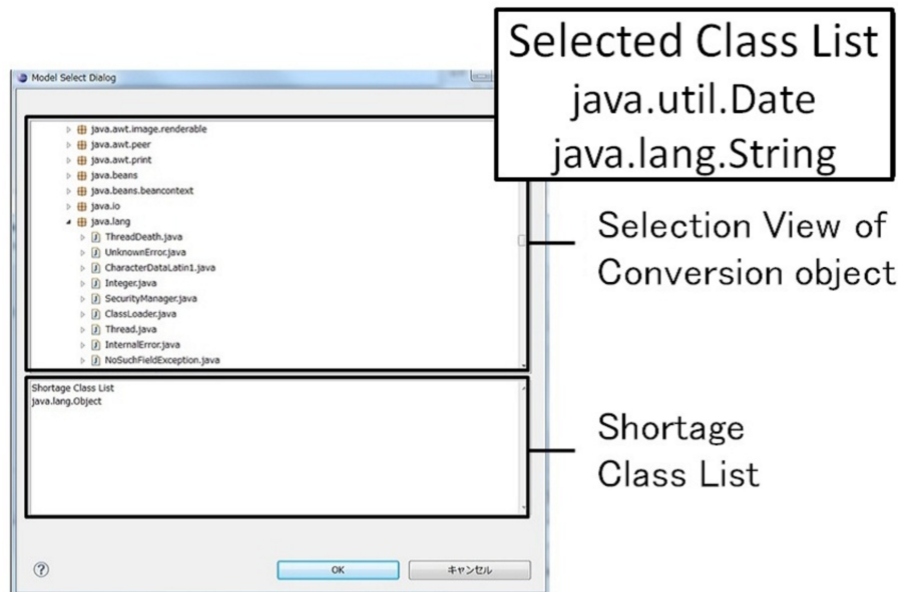

図 10: Java2UML の変換対象選択ダイアログ

クト単位での選択しか行えず，不必要なクラスを多く生成し得るため，変換対象の拡張は継承関係の確認にも活用されていると考えられる．この変更により，ユーザーが手動で継承関係を確認する必要がなくなり開発コストの削減に繋がる．また手動での確認よりも正確に継承関係の情報を取得できるので，生成される UML の品質も向上すると考えられる．

外部ライブラリの多くはオープンソースなので，Java から UML への変換を行うことが可能なツールを用いれば外部ライブラリの UML の情報が記述された XML ファイルは容易に取得することが可能である．オープンソースでない場合でも，逆コンパイルによってソースを取得し，UML の生成を容易に行える．

また，Java2UML は Eclipse プラグインとして実装されているため，OCLtoJML や Papyrus UML と親和性が高いと考えた．

3.2.3 適用例

図 10 に適用例を示す．

ここではまず使用したい外部ライブラリとして java.lang.String と java.util.Date の 2 つを選択して OK ボタンを押す．このクラスは両方とも java.lang.Object を継承しているので，不足クラスと java.lang.Object が出力される．ユーザーはこの不足クラスを解消するために引き続きダイアログからファイルの選択を行う．今回の例では，java.lang.Object を選択すれば，java.lang.Object はクラスを継承していないので，全ての継承関係を満たしたことになり UML への変換できる．

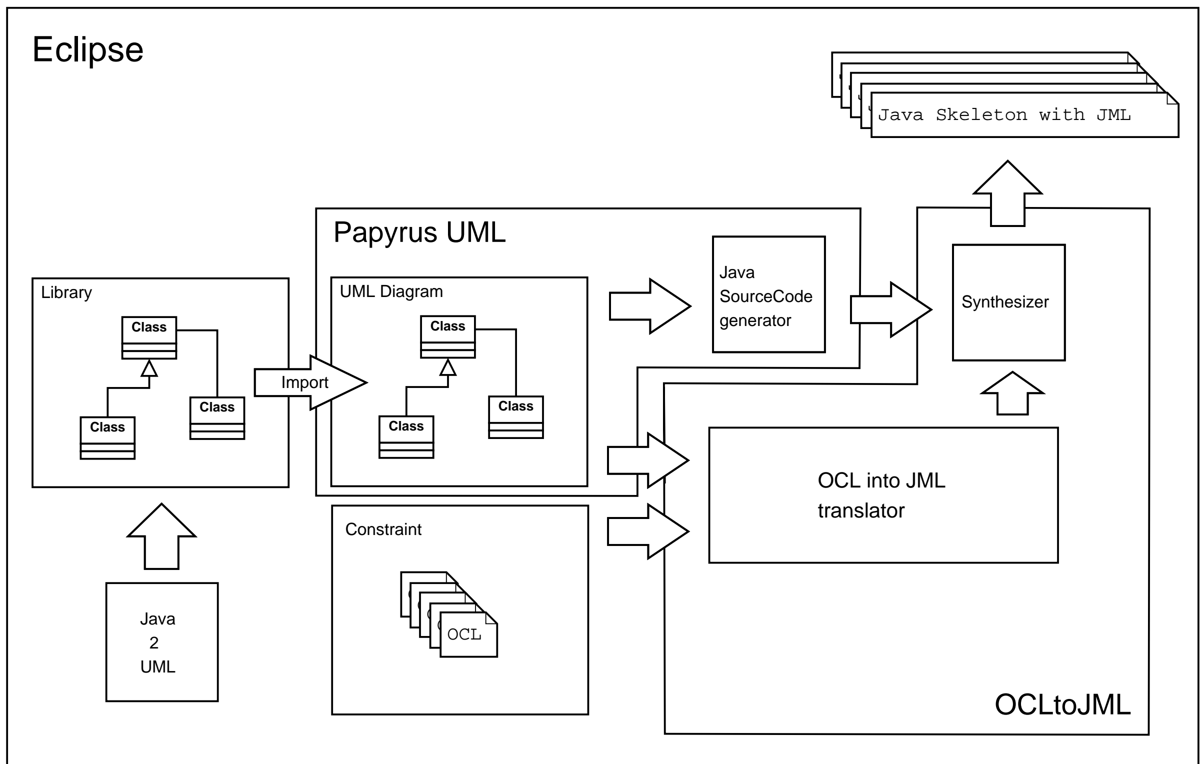

図 11: 外部ライブラリ作成も含めた開発手順の概要

3.2.4 開発手順の概要

外部ライブラリの作成も含めた一連の開発手順は以下のとおりである.

1. Papyrus UML で UML クラス図を作成.
2. 必要な外部ライブラリの UML クラス図を Java2UML で作成.
3. 外部ライブラリの UML クラス図を Papyrus UML で作成している UML にインポート.
4. 1 から 3 を UML の作成が完了するまで繰り返し行う.
5. Java スケルトンコードを UML クラス図より Papyrus UML のコード生成機能を用いて生成.
6. UML に付加する OCL 文のテキストを作成.
7. UML，OCL，Java スケルトンコードを格納したディレクトリの 3 つをそれぞれ入力する.
8. 正しく 3 か所に入力が行われれば，実行ボタンが有効になるので実行ボタンを押して実行を開始する.

9. OCL と UML から OCL の構文解析を行い，OCL を JML へ変換し，その JML を Java スケルトンコードに付加する．

図 11 に外部ライブラリの作成も含めた開発手順全体の概要を示す．図 11 と一連の開発手順からわかるように，外部ライブラリの作成も含め，JML の生成までの処理は全て Eclipse 上で行うことが可能である．

4　評価実験

4.1　実験概要

本研究では OCLtoJML を大規模な実プロジェクトに適用し，性能評価，および生成される JML の品質の確認のための評価実験を行った．

文献 [13] では 3 クラス 20 メソッドという小規模なプロジェクトに対して行われたのみである．これは OCLtoJML の適用範囲に制限が存在していたため，大規模な実プロジェクトに適用できなかったためである．本研究では適用範囲の拡張を行い，大規模な実プロジェクトに対して OCLtoJML を適用することを可能にした．

性能評価の指標として，以下の 2 点を計測した．

- 変換時間
  OCLtoJML により OCL が JML に変換されるまでに要する時間を計測する．変換時間は，OCLtoJML に対して変換対象の UML と OCL を入力した時点から，JML の抽象構文木が出力されるまでの時間とする．Java 標準ライブラリを用いて各時点の時刻（ms）を取得し，出力時刻と入力時刻の差分を計算することで計測を行う．

- 変換率
  入力した OCL のうち JML へ変換できた割合

変換率の計算式

$$\frac{OCLtoJML \text{ によって } OCL \text{ から } JML \text{ に変換できた式数}}{OCLtoJML \text{ に入力する } OCL \text{ の全事前・事後条件式数}} \quad (1)$$

また，JML の品質の確認の指標として再現率を計測した．

- 再現率
  OCLtoJML により生成された JML と手書きの JML が一致する割合

再現率の計算式

$$\frac{\text{生成された } JML \text{ と手書きの } JML \text{ で完全一致した式の数}}{OCLtoJML \text{ によって } OCL \text{ から } JML \text{ に変換できた式数}} \quad (2)$$

式（2）における手書きの JML に関しては 4.3.2 節において説明している．また，完全一致した式とは，括弧の数以外の記述が 1 字 1 句同じものを指し，記述方法は異なっているが式の意味が同じものを，意味的一致した式とする．

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．

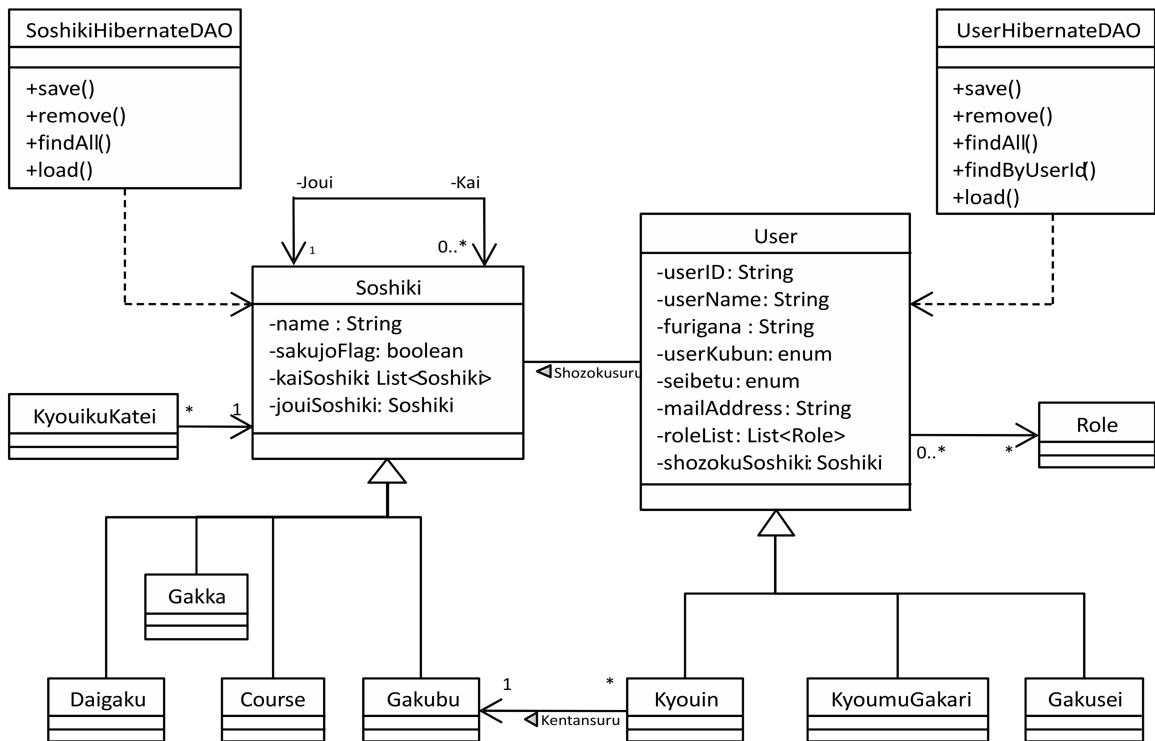

図 12: 教務システムのクラス図 1

4.2 実験対象

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）[24] の教材として作成された，和歌山大学の教務システムを実験対象とした．このシステムは 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である．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 173 クラスである．本研究における評価実験では，その内の 60 クラス 384 メソッドを JML を付加する対象として選択した．選択した 60 クラスはシステムの中でもデータベース操作と関連の強いシステム全体の機能の中心となる部分である．

図 12 と図 13 に教務システムの一部のクラス図を示す．

4.3 実験準備

対象のプロジェクトにはメソッド毎のテストケースクラスと，システムを構成する Java のソースコードは存在したが，JML，OCL，UML および全メソッドを通過するようなテストケースクラスは存在しなかった．また JML を付加するにあたり，既存の Java のソースコードのままでは問題が生じてしまうので，修正しなければならなかった．以下，それらの実験に必要なデータの準備の詳細を述べる．

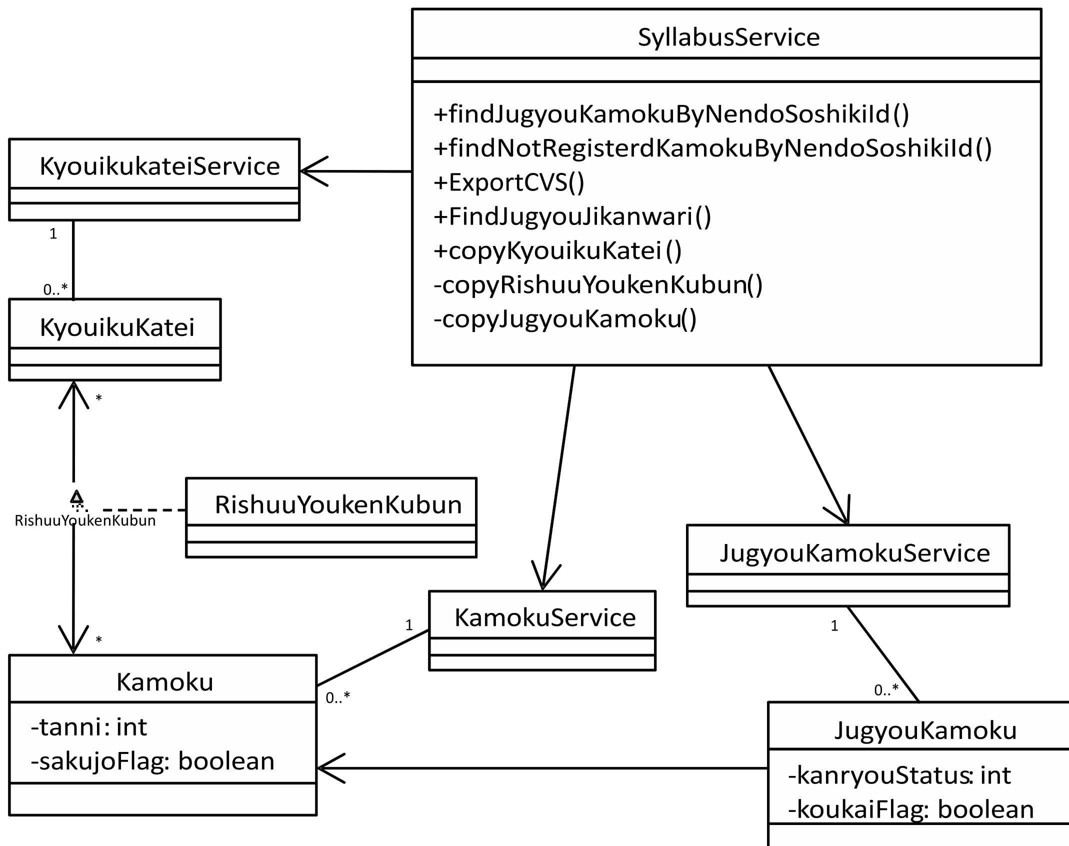

図 13: 教務システムのクラス図 2

4.3.1　対象プロジェクトの Java コード

実験対象のプロジェクトの Java コードで実験を行う際に問題となった点とその修正方法を説明する．

このプロジェクトではデータベースとのマッピングにはアノテーションが用いられていた．また，JML の記述にもアノテーションが用いられている．例として，あるメソッドに対して hibernate アノテーションと JML の記述が混在した例を図 14 に示す．

jml4c が hibernate のマッピング用アノテーションと JML のアノテーションを区別できず，hibernate のマッピング用アノテーションを JML として構文解析を行ってしまうため，hibernate 用のアノテーションがコード中に存在する場合はコンパイルができない問題が生じた．この問題を解決するために，元のプロジェクトのコード中に存在する hibernate 用のアノテーションと意味的に等価なマッピングファイルを作成し，コード中の JML 記述以外のアノテーションを削除した．

```
/*@
    ensures \result.equals(this.name);
@*/
@Column(name = "NAME", length = 40)
public String getName() {
  return this.name;
}
```

図 14: JML と hibernate のアノテーションが混在した例

4.3.2　JML

評価実験のため 2 通りの JML を作成した．1 つは，システムを構成する Java のソースコードから考えられる仕様を，手書きで JML として作成したものである．もう 1 つは，OCLtoJML を用いて生成したものである．

手書きで作成した JML

システムを構成するソースコードと，もともと用意されていたテストケースクラスから考えられる仕様に適した JML をソースコードに付加した．この JML を付加したソースコードから jml4c を用いて生成した実行ファイルがもともとプロジェクトに用意されていたすべてのテストクラスを通過したことから，手書きによって作成した JML は妥当なものだと考えられる．

また，JML の作成は複数人で行った．これは，1 人で作成するよりも様々なパターンの JML 式を作成できると考えたためである．同じようなパターンの JML 式を用いてテストを行うよりも，様々なパターンの JML を用いてテストを行う方が，より高品質な評価が行えると考えられる．ここで作成された事前・事後条件は 602 個である．

OCLtoJML を用いて生成した JML

作成した OCL（4.3.3 節）と UML から，OCLtoJML を用いて JML を生成した．

4.3.3　OCL

作成した JML と意味的に等価な OCL を手動で作成した．また，OCL の作成も複数人で行った．OCL の作成では，自分以外の人が作成した JML を元に OCL の作成を行った．これは，自分で作成したものを元に作成を行うより，他人が作成したものを元に作成を行うことで，様々なパターンが生成されるのではないかと考えたためである．

表 3: OCLtoJML による OCL から JML への変換率

全クラス数	60
手書きで作成した JML の全事前・事後条件式数	602
手書きで作成した OCL の全事前・事後条件式数	541
OCLtoJML によって OCL から JML に変換できた式数	541

全 602 個の JML 式のうち，OCL で表現できた式は 541 個である．残りの 61 個の JML と等価な OCL は記述できなかった．記述できなかった OCL の詳細は 4.4.2 節で説明する．

4.3.4 UML

教務システムの 173 クラスに加えて，JML で使用する外部ライブラリのクラスを 1 つのパッケージにまとめ，Java2UML によりリバースエンジニアリングを行うことで UML を作成した．Java2UML では対応できないインナークラス，列挙型とクラス間の関連については手動で修正した．

4.3.5 テストケースクラス

各メソッドをテストする JUnit[25] のテストクラスはプロジェクトに用意されていたが，全てのメソッドの動作を確認するためのテストクラスが存在していなかったので，各メソッドをテストするクラスを順次呼び出して実行するような全メソッド用テストクラス作成した．

4.4 計測結果

計測内容は変換時間と変換率および再現率の 3 点である．

4.4.1 変換時間

541 個の事前条件・事後条件の OCL から JML への変換に要した時間は 7 秒であった．これは十分に実用的な実行時間であると考えられる．

また，以前の評価実験では 2 つの事前・事後条件式の変換に要した時間は 21ms だった．2 つの式の変換に 21ms という数値を元に，541 個の式の変換に要する時間を計算すると約 5.7 秒になる．このことから変換に要する時間は，おおよそ事前・事後条件式数の線形時間で収まると考えられる．

4.4.2 変換率

結果を表 3 に示す．変換率の計算は式（1）に従う．表の値からもわかるように，入力した OCL 式は全て JML 式に変換されたので変換率は 100%である．

また OCL で記述できなかった条件式の主な内容を表 4 に示す．

表 4: OCL で記述できなかった条件式の内容

内容	条件式数
クラスリテラル	25
配列	18
その他	14

```
/*@
  requires this.getDAO(RishuuDAO.class) != null;
  ensures  \result == null || \result.getId() == id;
  signals (ServiceException) true;
@*/
public Rishuu load(final int id) throws ServiceException {
  return getDAO(RishuuDAO.class).load(id);
}
```

図 15: 変換できない JML の記述例（クラスリテラル）

表 4 からわかるように，クラスリテラル，配列に関するものが，OCL で記述できなかった内容の大部分を占めている．以下にそれぞれを具体例とあわせて説明する．

- クラスリテラル

  対応できなかった JML の例を図 15 に示す.

  RishuuDAO.class に対応するような文法が OCL には存在しないため，事前条件の this.getDAO(RishuuDAO.class)!=null は OCL で記述できなかった．このようなクラスリテラルの記述は Java 特有の表現であり，OCL での記述は難しい.

- 配列

  対応できなかった JML の例を図 16 に示す.

  事後条件の JML に対応するような OCL 文が記述できない．OCL では forall のような Collection 演算は Set や Sequence などに対応するものが存在する．通常，配列は OCL では Sequence を用いて記述される．そのため本来，UML や OCL を設計データとして実装を行った場合，Sequence を実装に移す際には List を用いると考えられる．しかし今回の評価実験の対象には OCL や JML はもともと用意されていないため，メソッドの戻り値や引数の型に配列を用いている個所がい

```
/*@
  requires rishuuKanouGakunenList != null;
  ensures ( \forall int i; i >= 0 &&
    i < \result.length; \result[i] != null);
@*/
public String[] getRishuuKanouGakunen() {
  return StringUtils
  .split(rishuuKanouGakunenList, Constants.DB_DELIMITER);
}
```

図 16: 変換できない JML の記述例（配列）

```
context RoleHibernateDAO::save(role : Role)
pre : not role.oclIsUndefined()
post : self.getSession().contains(role)
```

図 17: 適用範囲の拡張により変換可能になった OCL

くつか存在する．また OCL 自体が特定の言語仕様に依存しないものであるので，具体的な配列の型は変換の対象として考慮しないものとする．

また，適用範囲の拡張を行ったことにより変換を行うことが可能になった OCL と生成された JML の例をそれぞれ図 17 と図 18 に示す.

拡張前であれば Session 型の持つ contains メソッドが認識できず意味解析を行うことができなかったが，適用範囲の拡張を行ったことで contains メソッドの情報を得ることが可能になっているので，このような事後条件式も変換可能になった.

4.4.3 再現率

表 5 に手書きの JML を OCLtoJML がどの程度再現できたかを示す.

再現率は式（2）に従って計算すると 86.0% となる.

意味的に一致した 76 クラスのうち 57 クラスは，OCLtoJML が iterate 演算に対応するためにメソッドを生成したものである．iterate 演算の変換例を示す．図 19 に手書きで作成した JML を，図 20 に手書きの JML を元に作成した OCL を，図 21 に OCLtoJML で生成された JML を示す.

```
/*@
  requires role != null;
  ensures this.getSession().contains(role);
@*/
```

図 18: 適用範囲の拡張により変換可能になった OCL から生成された JML

表 5: OCLtoJML による JML の再現率

OCLtoJML によって OCL から JML に変換できた式数	541
手書きと完全に一致した式数	465
手書きと意味的に一致した式数	76

OCLtoJML では OCL の iterate 演算に対応するために，まずループ部分と意味的に等価なメソッドを生成する．そして生成したループ用のメソッドを JML 側から呼び出すことによって iterate 演算に対応している．図 19 と図 21 からわかるように，同一のメソッドに対する JML の記述方法は異なっているが意味的には等価である．

また，手書きのものと意味的には一致したもののうち iterate 演算以外の残り 19 クラスのうちの一部の例として図 22 に手書きで作成した JML を，図 23 に手書きの JML を元に作成した OCL を，図 24 に OCLtoJML で生成されたの JML を示す.

図 22，図 24 からわかるように，元の JML では instanceof 演算子を用いているのに対し，生成された JML は isAssignableFrom メソッドを用いている．こちらも記述方法は異なっているが意味的には等価であり，実行するにあたって問題なく，実行時間にもほぼ影響を与えない.

また全体で変換できたもののうち 86.0%が手書きのものと完全に一致している．この計測結果から OCLtoJML で生成される JML はほぼ手書きの JML と変わらない品質を持つと考えられる．

```
/*@
 ensures kinouList != null ? ( \forall Kinou kk; kinouList.contains(kk);
 this.kinouList.contains(kk)) &&
 kinouList.size() == this.kinouList.size() :
 this.kinouList == null;
@*/
public void setKinouList(final List<Kinou> kinouList) {
  this.kinouList = kinouList;
}
```

図 19: iterate 演算を含む元の JML

```
post:if not kinouList.oclIsUndefined()
     then kinouList->forAll(kk : Kinou | self.kinouList->includes(kk))
          and  kinouList->size() = self.kinouList->size()
     else self.kinouList.oclIsUndefined()
     endif
```

図 20: iterate 演算を含む OCL

```
/*@
  ensures (kinouList != null ? mPrivateUseForJML25(kinouList) &&
   kinouList.size() == this.kinouList.size() : this.kinouList == null);
@*/
private boolean mPrivateUseForJML25(Sequence<Kinou> kinouList){
    boolean res = true;
    for(Undefined kk : kinouList){
        res = res && this.kinouList.contains(kk);
    }
    return res;
}
```

図 21: iterate 演算を含む OCL から変換された JML

```
/*@
  ensures  \result == null || \result instanceof List;
@*/
public List<Kamoku> findAllIncludeDeleted() throws ServiceException {
  return getDAO(KamokuDAO.class).findAllIncludeDeleted();
}
```

図 22: 意味的に一致するパターンの元の JML

```
context KamokuServiceImpl::findAllIncludeDeleted():Sequence(Kamoku)
post : result.oclIsUndefined() or result.oclIsKindOf(List)
```

図 23: 意味的に一致するパターンの OCL

```
/*@
  ensures \result == null ||
    List.class.isAssignableFrom(\result.getClass());
@*/
public List<Kamoku> findAllIncludeDeleted() throws ServiceException {
  return getDAO(KamokuDAO.class).findAllIncludeDeleted();
}
```

図 24: 意味的に一致するパターンの OCL から変換された JML

4.5　考察

本研究における評価実験を通して，OCLtoJML による OCL から JML への変換に要する時間は十分に実用的な範囲であることを確認した．また，OCL から JML への変換自体は 100%行うことが可能という結果から OCLtoJML の変換性能も十分に実用的なものであることを確認した．さらに，変換されたものは手書きのものとほぼ完全に一致しているものが 86.0%，意味的な観点からすれば 100%一致しているという点から，十分な品質を持っていることを確認した．OCL で記述できなかった部分は主に以下の 2 点であった．

- クラスリテラル
- 配列

まずクラスリテラルは Java 特有の言語仕様である．OCL は特定の言語仕様に依存しないため，クラスリテラルと意味的に等価な文法を保持していない．また，設計段階において特定の言語仕様に依存し過ぎることは不適切であると考えられる．

また配列は OCL では Sequence を用いて表される．Sequence は要素の順序を保持することが可能で，かつ要素の重複を許すリストである．しかし，Sequence は配列というよりはむしろ Java でいう ArrayList に近い．OCLtoJML による変換でも Sequence はリストとして出力される．また，通常は設計の段階で Sequence が配列であろうがリストであろうが，どのように実装されるかは考慮する必要がない．したがって，設計段階である OCL 記述において，配列を具体的に考慮した式を記述する必要性は低いと考えられる．

5　あとがき

本研究では，OCL を付加した UML クラス図を，JML を付加した Java スケルトンコードに変換するツールである OCLtoJML の Eclipse プラグイン化，ツールの適用範囲の拡張，およびツールの教務システムに対する適用実験を行った．

評価実験では，変換の対象となった 541 個の OCL 式はすべて JML 式に変換できた．また OCLtoJML によって生成された JML は単体テストを行うにあたり，十分な品質を持っていることを確認した．変換に要した時間も 541 個の式に対して 7 秒と十分に実用的な範囲だった．これらの点から OCLtoJML の実用性は高いと言える．

今後の研究方針としては，直接 OCL から JML への変換を行うのではなく，ATL などを用いたメタモデルを利用したモデル変換などが考えられる．メタモデルの変換を用いたマッピングによる変換での実装を行っていた場合，言語仕様が変更されたとしても，設計資産としてメタモデルを有効に活用できるなどの利点がある．

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，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.

本研究に用いたツールの大部分を設計, 実装してくださり，また本研究に関して多大なるご助言,ご助力を頂きました，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博士前期課程 2 年の 宮澤 清介 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，実験等でお世話になり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] 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] Object Management Group. OMG Modeling Specifications. `http://www.omg.org/`.

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

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

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

[11] C. Avila, 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.

[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. *IEICE technical report*, Vol. 110, No. 169, pp. 53–58, 2010.

[13] 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.

[14] Eclipse Foundation. Papyrus UML. `http://www.papyrusuml.org`.

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

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

[17] Freddy Allilaire, Ivan Kurtev, and Jean Bezivin. ATL: A model transformation tool. *Science of Computer Programming*, Vol. 72, No. 1-2, pp. 31–39, 2008.

[18] 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.

[19] 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.

[20] Amritam Sarcar and Yoonsik Cheon. A new eclipse-based jml compiler built using ast merging. *World Congress on Software Engineering*, Vol. 2, pp. 287–292, 2010.

[21] Object Management Group. OMG Unified Modeling LanguageTM (OMG UML),Infrastructure, 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[22] TOPCASED. TOPCASED The Open-Source Toolkit for Critical Systems. http://www.topcased.org/.

[23] Atos Origin TOPCASED Team. *How to generate UML From Java*. http://gforge.enseeiht.fr/docman/view.php/7/279/TPC_2.4_Java_Reverse_tutorial.pdf.

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

[25] JUnit. http://www.junit.org/.