メソッドの自動生成を用いたOCLのJMLへの変換ツールの設計

Design of a translation tool from OCL into JML by translating the iterate feature into Java methods

尾鷲 方志*　岡野 浩三†　楠本 真二‡

Summary. The paper presents design of a translation library from OCL (Object Constraint Language) into JML (Java Modelling Language). Several approaches have proposed automatic generation methods of Java skeleton files from UML class diagrams. Some of them are publicized as plug-ins for Eclipse. On the other hand, less papers are found for automatic generation of JML from OCL. They deal with not all of the standard OCL library. Especially, some features of collections including iterate feature are not implemented. We resolve the problem by translating the iterate feature into Java methods. This paper also provides a concrete translation algorithm. The advantages of the proposed approach are as follows: 1. it helps to develop in Model Driven Development (MDD) with formal approach; 2. it conforms to test first which is often used in agile development, because designers can generate test cases using Java skeleton files with JML using JML standard tools; and 3. it helps to verify correctness of the program using model checking tools such as ESC/Java2.

1　まえがき

近年，モデル駆動型開発がソフトウェア開発の方法論として注目を浴びている．

UML 記述からのモデル駆動開発のアプローチとして MDA が OMG により提唱されており，より厳密なモデル定義の手段として OCL が定義されている．OCL は UML 記述に対しさらに詳細に性質記述を行うことを目標に設計された言語であり，OMG によって標準化されている．また，UML 記述レベルで設計無矛盾性検出を行う研究は世界的に多くされてきている [10,11].

モデル駆動型開発において，設計プロセスをできるだけ自動化することにより設計の後戻りに容易に対応することが可能となり，アジャイル開発への親和性も高いと考えられる．モデル駆動型開発の利点を推し進めると OCL を UML 記述の段階のみに適用するだけでなく，次のプログラム実装の際にも活用できることが望まれる．Java プログラムに対しては OCL に似た役割りを果たす制約記述言語として JML (Java Modeling Language) [7,8] が提案されている．JML，OCL ともに DbC (Design by Contract) の概念に基づきクラスやメソッドの仕様を与えることができる．

本研究ではそれらの点に加え，アジャイル開発における軽量フォーマルアプローチ (light-weight formal approach) の適用を考慮に入れ，OCL の JML への変換ライブラリの設計を行った．

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 [19,20]，自動変換ツールも EMF フレームワークを用いた Eclipse プラグインなどの形で公開されている [12]．一方 OCL から JML への変換については，Moiseev と Russo らが文献 [5] において OCL から JML への変換法とツールの実装を示している．また，Hamie は文献 [6] 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が，いずれの方法

*Masayuki Owashi, 大阪大学

†Kozo Okano, 大阪大学

‡Shinji Kusumoto, 大阪大学

も Collection の対応が不十分でありとりわけ iterate の対応がされていない．

そこで本研究では Eclipse プラグインに実装する前提でこれらの論文で示されている変換法より OCL 記述のクラスに対して OCL 記述からの JML 記述変換方法を具体的に提示し，その変換方法を用いた設計開発方法の適用可能性について考察する．

以降，2 章で提案する手法の背景について述べ，3 章で変換方法とライブラリ設計について述べる．4 章で背景を振り返ったのち，変換ライブラリのアジャイルソフトウェア開発への応用の可能性について述べ，5 章でまとめる．

2 準備

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 OCL

OCL [2] は OMG 標準の 1 つであり，制約式を述語を用いて記述する．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た．

OCL は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，詳細に表現する．

Design by Constract [9] 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，不変条件等を記述することができる．

2.2 JML

JML [4, 7, 8] は Java プログラムにおいて Design by Constract [9] 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，不変条件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，初心者でも記述しやすい特徴を持つ．また，記述は Java コメント中に記述できるため，プログラムの実装，コンパイル時に影響はなく，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 型を持つ任意の式で与えられる．また，JML は事前条件，事後条件，不変条件などを記述するために，それぞれ ensure, requrire, invariat 等のキーワードを持つ．さらに，より実装よりの条件を詳細に記述するために signal, pure, assignable, assert など豊富なキーワードを用意している．

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 ESC/Java2 [1] を用いて静的検査 (モデル検査) をメソッド単位で行うことができる．ESC/Java2 は完全性，健全性を保証しているわけではないが，軽量モデル検査の概念に基づき，バグだしを行うのには実用的に有用とされる．ESC/Java2 はプログラムの正しさを述語論理に変換し，その充足不能性を判定することによって行う．判定エンジンとして Simplify [3] を用いている．

2.3 関連研究

UML から JML への変換については，Engels らの文献 [19] や Harrison らの文献 [20] 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．Moiseev と Russo らは文献 [5] において OCL から JML への変換法とツールの実装を示している．Hamie は文献 [6] 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 の対応が不十分でありとりわけ iterate の対応がされていない．また文献 [6] 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 (表 1)．

本論文では Collection に対応する Java プログラムを用いることでこの問題を解決する．また，変換に際し型情報を用いた，より厳密な変換プロセスを提案する．

UML 記述レベルで設計無矛盾性検出を OCL 記述を利用して行う研究は多いが，本研究は OCL を設計対象ソフトウェアの仕様記述に用いることを仮定している．

表 1　既存手法との比較

Feature	提案手法	Moiseev と Russo	Hamie
基本演算	√	√	√
Collection	√	√	-
Iterate (forall etc)	√	√	√
Iterate (collect)	√	√	-
Iterate (iterate)	√	-	-
Structures (Set etc)	√	√	√
Structures (tuple)	√	√	-
OCL Spec.	√ but partially	√	√
Message	-	-	-

3　OCL から JML への変換手法

この章では文献 [6] の構文変換を元に JML への変換方法を与える．

3.1　構文変換の仮定と基本方針

OCL の基本式だけに限っても，標準ライブラリには多くの基本型と演算子を用意している．すべてを実際の記述に用いるわけではないのでそれら全ての変換を行うのは現実的でないため，構文変換にあたり，以下の仮定を置きクラス制限を行う．

1. OCL Basic で定義されている型に Collection を表すために OCL Essential の一部の型を追加した型を表す式を対象とする．
2. 対象 (部分) 式に型 (あるいは型の候補情報) が定義でき，構文変換時にその型情報が利用できるものとする．型として，Bool, Integer, Real, String, Collection, Set, OrderedSet, Sequence, Bag を仮定する．これらは，そのまま Java(JML) のクラス (インタフェース) に対応する．
3. いくつかの OCL 構文からは型的に不整合な表現式が導出され得るがそのような型不整合な式は対象にしない．
4. Message 型および，message 演算は変換の対象外とする．
5. OclVoid は Undefined 定数のみをもつクラスであり，未定義値を意味する．Java(JML) では null に相当するのでここでは触れない．

議論の正確さのため，本論文では構文変換を与えるにあたり，いくつかの OCL 演算子のうち，他の OCL 演算子の組み合わせで表現できるものはそのように対応した．この対応においては OCL の定義書にある定義に従った．

この方法の利点としては多くの演算子の変換の妥当性を OCL の定義書に委ねることができることが挙げられる．一方，欠点としては変換された JML 式の読解性低下が挙げられる．この問題の解決としては変換ライプラリ実装時に両方の変換に対応できるようにすることが考えられる．

本論文では前者の立場で議論を進める．この立場で構文変換を行うと Collection に関する OCL 標準ライブラリの多くの演算子 (feature call) が iterate 演算子とその他の演算の組み合わせに集約することができる．また，OCL の Collection と Java の標準 Collection Framework は多くの類似点を持っている．

なお，いくつかの変換において，Java 1.5 以降の構文を利用できる場合は利用した．ただし ESC/Java2 は 1.4 にしか対応していないため，1.4 に対応する方法も併記している．

3.2　構文変換

ここでは主要な構文に対して，文献 [6] の構文変換を拡張した構文変換法を与える．文献 [6] にならい，OCL 式から JML 式への変換関数の表記を μ で与える．

表 2　μ 変換 Collection loop features

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

表 3　μ 変換 Collection loop features Set

$\mu(st_1{-}{>}\text{select}(a_1 \mid a_2))$	=	$\mu(st_1{-}{>}\text{iterate}(\ a_1; res : \text{Set}(T) = \text{Set}\ \{\} \mid$ if a_2 then res $-{>}$includeing (a_1) else res endif))
$\mu(st_1{-}{>}\text{reject}(a_1 \mid a_2))$	=	$\mu(st_1{-}{>}\text{select}(\ a_1 \mid \text{not } a_2))$
$\mu(st_1{-}{>}\text{collectNested}(a_1 \mid a_2))$	=	$\mu(st_1{-}{>}\text{iterate}(\ a_1; res : \text{Bag}(a_2.type) = \text{Bag}\{\} \mid res\ {-}{>}\text{including}(a_2)))$
$\mu(st_1{-}{>}\text{sortedBy}(a_1 \mid a_2))$	=	$\mu(st_1{-}{>}\text{iterate}(\ a_1; res : \text{OrderedSet}(T) = \text{OrderedSet}\{\} \mid$ if $res{-}{>}\text{isEmpty}()$ then $res.\text{append}(a_1)$ else $res.\text{insertAt}(res{-}{>}\text{indexOf}($ $res{-}{>}\text{select}(i \mid a_2(i) > a_2(a_1){-}{>}\text{first}())),a_1)$ endif))

文献 [6] では μ を基本データと演算，および，コレクションの一部の演算についてのみ，与えている．本研究では対応クラスのほぼすべてについて μ の定義を新たに与えていくが，ここでは本論文において新たに定義された，コレクションループ演算，Iterate 演算について述べる．

以下では型 Bool, Integer, Real, String, Collection, Set, OrederedSet, Sequence, Bag，ユーザー定義クラスを持つ部分式をそれぞれ $b_m, i_m, r_m, s_m, c_m, st_m, os_m, sq_m, bg_m, u_m$ で表す $(m = 1, 2, 3, \ldots)$．また，任意の型を持つ部分式を a_m で表す．

3.2.1　コレクション演算 ループ演算

コレクションのループ演算を表 2 ，Set に対するループ演算を表 3 に与える．

その他のサブクラスにおいても同様の変換を定義すると，これまでにて変換が未定義である feature は iterate であることがわかる．これについては後の節で述べる．なお，collectNested() は各サブクラスで個別に変換定義される．

3.2.2　その他の構文変換

OCL 構文の featureCall はオプショナルにナビゲーション式 [exp] と時間修飾子がつく．ナビゲーション式はアソシエーションの方向を規定するためのものでありUML 図におけるコンテキスト解釈の際に必要となるが，ここでは省略する．時間修飾子は OCL 2.0 ではキーワード@pre のみが定義されている．JML では同等のキーワード \old が用意されている．従って，@pre を \old を用いて置き換えればよい．

OCL 構文の primaryExp についてみる．primaryExp は literalCollection か literal, (expression), ifExp, letExp, tupleExp の何れかである．

if 式 ifExp は OCL では関数型言語のように任意の型を持つ式として定義されているが，JML や Java ではあくまで制御フロー式となる．そのかわり　Java では cond?exp1:exp2 構文を用いることができる．対応にはこれを用いる．

したがって literalCollection 以外の構文の変換則は表 4 となる．

let 式は基本的には複雑な式を分解し，見やすくしているだけであり，変換は式中の定義に従い，順次変数代入を行うだけでよいのでここでは変換則は省略する．

OCL 構文の literalCollection は Collection の定数初期宣言に相当する．変換則を

表 4　μ 変換 Primary Expression

$\mu(a_1)$	=	a_1
μ(if b_1 then a_1 else a_2 endif)	=	$(\mu(b_1)?\mu(a_1):\mu(a_2))$
$\mu((a_1))$	=	$(\mu(a_1))$

表 5　μ 変換 Primary Expression

μ(Collection $\{a_1\}$)	=	(new ArrayList($\{\mu_e(a_1)\}$))
$\mu_e(a_1, a_2)$	=	$\mu(a_1), \mu_e(a_2)$
$\mu_e(a_1)$	=	$\mu(a_1)$
$\mu_e(i_1..i_2)$	=	seq($\mu(i_1)$, $\mu(i_2)$)

表 6　μ 変換 enumeration 1.5 版

μ(Enum $\{a_1\}$)	=	Enum$n$

表 5 に与える．seq(i,j) は i から j の系列である．なお，Set，OrderedSet，Bag，Sequence については Collection と似た形式で変換できるためにここでは省略する．

OCL 構文の enumerataType については次のように変換する．

enum 型については Java 1.5 で enum 型が用意されている．これを用いて変換する．

実際の変換においては，enum の宣言部と使用部にわける必要がある．

例えば次の OCL における enum 宣言式　Enum{ kind, name } に対して，以下の Java 記述を生成する．

```
enum Enum01 { kind, name }
```

一方，変換中の enum 宣言式に対しては Enum$n$ を型として用いる (表 6)．

1.4 に対応するためには単純に String の Set として変換する．

その他，OCL の構文上現れるパス名の結合子 "::" は "." に変換する．

いくつかのキーワードは対応する JML キーワード等に変換可能である．単純なキーワードの置換なのでここでは詳細は省略する．

OclType 型は JML の Class クラスのメソッドを対応させることで変換できる．

3.3　tuple 型

tuple 型については対応する Java クラスのソースコードを変換時に生成するアプローチをとる．OCL における tuple の操作は tuple の要素に対するアクセスのみであり，これは Java における field のアクセスとしてそのままシームレスに対応づけできる．クラス生成時において重複をさけるようにクラス名に気をつける必要がある．

例えば次の OCL における tuple 宣言式に対して，以下の Java ファイルを生成する．

Tuple{name(: String), id= "i023454" }

```
public final class Tuple_01 {
  public static String name;
  public static String id = "i023454"'
}
```

3.4　iterate の変換

iterate の変換は次の理由により，構文変換による naive な変換では対応できない．

- iterate feature の引数はほぼ任意の OCL 式であり，単純な構文変換では変換先の言語 L において，L の任意式の評価機構 (クロージャ) [17] が必要となる．
- JML や Java はクロージャを直接的にはサポートしていない．

幸い，本研究では OCL 式のインタプリタを実装するわけではなく，言語変換を

行うので，次のステップを踏むことにより，間接的にこの問題に対応できる．

- iterate feature の引数である OCL 式を iteration 込みで評価するためのメソッド m を変換時に作成する．
- μ 変換ではそのメソッドの結果を参照する．

一般に，iterate feature の引数は任意の OCL 式であるためにメソッド m の引数として OCL 式あるいはそれに相当する Java 式 (JML 式) を持つことはきない．そこでメソッド m は無引数とし，評価式はそのままメソッド m の body 内で展開する．変数などのスコープを乱さないため，メソッド m は JML 式がアノテートされる Java ソースプログラム内に private メソッドとして実装される．このメソッドはプログラム本来の実装には無関係なのものとなる．

Java 1.5 より for-each 文がサポートされている．iterate feature は for-each 文に変換することにする．ここでの変換はやや複雑であり，変換対象 JML 式の内部の変換とメソッド m の変換の 2 つからなる．

一般に，これまでの μ 変換は OCL 構文木の根から葉に向かって再帰降下的に行なわれてきた．ここで，もともとの対象となった OCL 構文木のうち，対象となる iterate feature をもつ部分木 (s) を含む部分木で，かつ，navigation を非終端記号としもつノードを根とする部分木 (t) であり，かつ，t から s までのナビゲーションがすべて $->$ である部分木のうち極大なものを改めて変換の対象とする．この部分木の表す OCL 部分式を t とする．

部分式 t にいたるまでに変換された JML の式文脈を Context[] で表す．

iterate feature の引数内で初期化式 $init$ で変数 res が初期化され，for-each 文による評価結果は変数 res に入る場合を場合 1．それ以外を場合 2 で表す．

場合 1

Context[$\mu(a_1$ ->iterate(e; $init$ | $body$))] に対し，初期化式 $init$ で T_2 型の変数 res が初期化され，iterate 文による評価結果が res に入る場合, 次の JML 式を生成する．

- メソッドとして

  private T_2 mPrivateUseForJML01() {
  $\mu(init)$;
  for (T e: $\mu(a_1)$){
  $res = \mu(body)$}
  return res;
  }
- 変換構文内で

  Context[mPrivateUseForJML01()]

場合 2

Context[$\mu(a_1$ ->iterate(e;| $body$))] に対し，a_1 の型を T_2 とし，次の JML 式を生成する．

- メソッドとして

  private T_2 mPrivateUseForJML01() {
  $\mu(init)$;
  for (T e: $\mu(a_1)$){
  $\mu(body)$}
  return $\mu(a_1)$;
  }
- 変換構文内で

  Context[mPrivateUseForJML01()]

変換例については文献 [21] で与える．

Java 1.4 に対しては上述の 変換例における for 文の代わりに，Iterator を用いる．

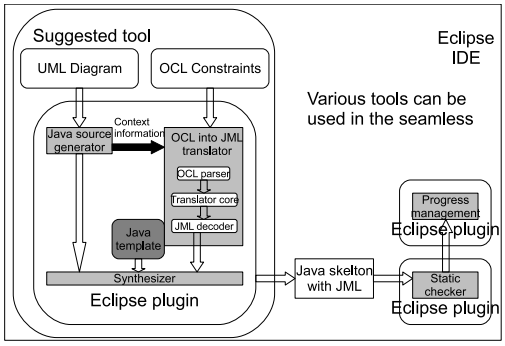

図 1　Tool implementation as Eclipse plugin

3.5　変換の等価性について

基本方針で述べたように，OCL 式レベルでの等価変換は OCL 仕様定義書で述べられている意味定義にしたがっている．Collection の一部の演算については直接 Java の CF のメソッドに対応付けさせている．この変換の正しさについては OCL 仕様定義書には OCL で事前条件，事後条件を提示しており，これと Java の CF に JML で記述された [16] 事前条件，事後条件とを対応し，検証することで確認することが可能である．

コレクションに関する演算については，一部 OCL の定義を使わず，Java の CF および，JML の式を使っている．このうち JML の式を使っているのは sum,includes のみである．この変換は見ての通り妥当である．Java の CF を使っているのは size(), isEmpty(), notEmpty(), union(), intersection(), including(), excluding() (一部のみ), as 演算，OrderedSet 固有の演算のすべてになる．これらについては，上述の方法で確かめる必要がある．

残りの loop feature については OCL の仕様定義にしたがっている．したがって，interate の変換の正しさのみを確かめればよい．この変換の基本は Iteration の Java による実装であり，直観的には実装の正しさは比較的容易に確認することができる．

3.6　変換ツールライブラリ設計

この節では変換ツールの設計方針 (主に構文変換系) についてやや詳しく述べる．

図 1 にツールの概念図を表す．

3.6.1　構文変換系

OCL の構文解析については既存のライブラリを活用する．具体的には Octopus tool [13], EMF の OCL パーザが挙げられる．

ここでは OCL の構文解析がおわり，OCL 木に対して，抽象構文木 (ABST; Abstract Syntax Tree) が得られていると仮定する．

簡単のため，ABST の各ノードは以下の情報を持つとする．

- NTS: OCL 構文に定義された非終端記号の列
- Type: 該当ノードの部分木に対する型
- NodeList: 子ノードリスト
- token: 演算子,feature 名, 終端 token 値等
- tokenType: token の型

OCL は SML と同様に，多相型を許す型付の言語である．例えば $1 + a$ という式から式全体が Integer 型，したがって a も Integer 型であると，いう推論ができる一方，if $a > 0$ then $a + b$ else a endif という OCL 式については式全体も変数 a, b もいずれも型は Integer 型か Real 型かいずれかであるということくらいまでしか推論

表 7　Signatures of TranslationCore

```
public static JMLNode transExpression(OCLNode n);
static JMLNode transLogicalExp(OCLNode n);
static JMLNode transRelationalExp(OCLNode n);
static JMLNode transPostfixExp(OCLNode n);
static JMLNode transPrimaryExp(OCLNode n);
static JMLNode transFeatureCallExp(OCLNode n);
```

できないこともある．いずれにせよ，与えられた OCL 式に対して，可能な限り型推論を行うことは SML の実行系に使われている型推論の機構 [15] と同様のアルゴリズムを適用することにより可能である．

与えられた抽象構文木に対して，そのような型推論アルゴリズムを適用し，各部分ノードの型を割り当てられていることを仮定しておく．

構文変換メソッドは各生成部法に従った変換サブメソッド群からなる．これらのメソッドは抽象構文木のノードを引数とし，変換された JML の部分式 (のノード) を返すメソッドとして signature 定義を行うことができる．

OCL 式の抽象構文木および JML 式の抽象構文木の各ノードのクラスは，それぞれ，OCLNode, JMLNode(のサブクラス) とする．

μ 変換規則では触れなかったが，実際に構文変換する際に，UML ダイアグラムのどの部分の制約を OCL 式が表しているかを意味するコンテキスト情報が実際の変換で必要である．また，名前空間上のクラスパスやアソシエーション情報の獲得のためにも UML ダイアグラムの情報が必要である．

表 7 に主要な変換メソッドの signature 定義を示す．

3.6.2　UML 変換系

UML クラス図から Java ソースコードのテンプレート導出のための変換系は EclipseUML などの利用を考えている．

構文変換系と UML 変換系は独立して構築できるわけではなく，前述のように構文変換系は UML 変換からコンテキスト情報やクラスのアソシエーション情報を得る必要がある．したがって，OCL 変換系と UML 変換系は最終的にはまとめて 1 つのライブラリとするのが望ましい．

また，このライブラリは Eclpse のプラグインとして実装したいと考えている．プラグインの形で統合開発環境に組み込むことで，定義されたモデルからコードの生成，検証，進捗管理を一つにまとめたフレームワークを作ることが考えられる．このような，検証に必要な枠組みを一つにまとめ，基幹部分を共用する試みは IVE [18] と呼ばれ，JML4 [18] で一部実現されている．

このフレームワークに対する入力として，OCL の付加された各種 UML ダイアグラムが与えられる．このダイアグラムと OCL は統合開発環境上で定義される．ここで OCL の抽象構文木が構文変換系に与えられ，各種ダイアグラムは UML 変換系によりコンテキスト情報を取得，構文変換系の入力として出力する．同時に，Java のスケルトンコードを生成する．

構文変換系により出力された OCL 抽象構文木をコンテキスト情報により Java スケルトンコードの適切な位置に出力し，これにより OCL 付きの UML ダイアグラムから JML 付きの Java スケルトンコードへの即座の変換を可能にする．同時に，生成した JML 付きの Java スケルトンコードを入力とし，即座に JML 向けの各種検証ツールを利用可能にすることで，設計段階で定義したモデルの正当性，変換後のコードの正当性の検証が可能になる．

また，ESC/Java2 等の検証結果を入力とし，要求仕様を満たす実装を行ったクラスやメソッドを取得することで進捗管理をすることで，使用頻度の低いメソッドに対する実装漏れ等の人為的なミスを削減することが可能になる．

これらを実現するフレームワークを Eclipse 上で実現することで，成果物の信頼性が向上すると考えられる．

4 今後の展望

近年，ソフトウェア開発においてアジャイル開発と呼ばれる，設計開発プロセスに柔軟性を持った開発法が注目を浴びている．アジャイル開発の特徴として，仕様の変更が起こりうることを前提としていること，テスト仕様やテストコードの生成を実装の前に行うテストファーストの考え方を取り入れていること，モデル駆動型開発を取り入れていることが挙げられる．

一方，従来より，ソフトウェアの信頼性を向上させるための技術としてフォーマルアプローチによる設計開発法が研究され，近年では，上位設計から下位設計にかけて行われるモデル駆動型開発方法が注目を浴びている．

従来，フォーマルアプローチによる設計開発では，設計仕様は大きく変化しないという前提で仕様に対する実装の正しさの検証を数学的，論理的に保証する技術を扱ってきたが，近年ではウォータフォールモデル以外の開発プロセスが用いられることが多く，また，モデル検査技術についてはスケーラビリティの弱さが克服されているとはいいがたい状況より，スケーラビリティに優れる軽量フォーマルアプローチの技術が注目を浴びるようになった．

軽量フォーマルアプローチの立場では「モデル検査ツールを正しさの保証をするために使うことよりも，見つけうるバグを徹底的になくすことを主体に使う」ことを目指している．そのため，上位設計において軽量フォーマルアプローチを適用することは現時点では実用性・適用可能性と高信頼性保証のトータルバランスから望ましいと考えられる．

本ライブラリツールを利用することによりモデル駆動型開発に対応できること，テストファーストに対応できること，といった利点が発生すると考えられる．

最初の点は，JML 入りの Java ソースファイルスケルトン生成を自動で行なっているため，ある程度対応できていると考えられる．実用性のためには，後戻りが発生したときに，差分情報を用いて生成しなおす部分とそうしなくても良い部分の切り分けが必要と思われる．

また，アジャイル開発では，仕様変更より先にコードレベルの変更を行うことがある．理想論からいえば，UML 記述の高位レベルからモデル駆動型開発にしたがって開発すべきであるが，上述のようにすでに開発済みの部分との切り分けが可能であればある程度この立場に立つことができる．

テストファーストへの対応については，生成された JML 付きの Java スケルトンに対し，実装以前において JML の標準ツールを用いたテストコード自動生成が可能であることにおいて対応できると考えられる．また，プログラム実装を行なったあとでも上述のテストコード自動生成の機能を用いたテスト環境が容易に利用できるほか，ESC/Java2 などを用いたソフトウェアモデル検査も実行できる．

上記のように，本手法をテストを用いた動的検査とモデル検査，静的検査と組み合わせることで，高信頼性ソフトウェアの短期開発に有用であると予想できる．

一方，仕様が変わりうるアジャイル開発を主眼に考えるならば，JML やスケルトンコードから OCL への変換機構も検討の余地がおおいにあると考えられる．

また，3.1 章で定義した仮定の下で，実用的な仕様記述に対し無理なく変換が行えるか様々な観点から評価する必要があると考えられる．

5 あとがき

上位設計において軽量フォーマルアプローチを適用することは現時点では実用性・適用可能性と高信頼性保証のトータルバランスから望ましいと考えられる．

本研究ではアジャイル開発における軽量フォーマルアプローチの適用を視野にい

れ，OCL の JML(Java Modeling Language) への変換ライブラリの設計を行った．
OCL 記述からの JML 記述変換方法を従来提案されていたクラスより，より広いクラスに対し具体的に示した．また，その変換方法を用いたソフトウェア設計開発方法への適用可能性について考察し，本手法はアジャイル開発とフォーマルアプローチの融合として有効性がある可能性があると判断した．
今後プラグインの実装を行い，実用例に適用し，有用性を評価したい．

参考文献

[1] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll: “An overview of JML tools and applications,” In T. Arts and W. Fokkink, editors, Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Electronic Notes in Theoretical Computer Science, Vol.80, pp.73–89, 2003.

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

[3] D. Detlefs: “Simplify : A Theorem Prover for Program Checking,” JACM, Vol.52 No.3, pp.365–473, 2005.

[4] D. R. Cok: “Specifying java iterators with JML and Esc/Java2,” In Proceedings of the 2006 conference on Specification and verification of component-based systems pp.71–74, 2006.

[5] R. Moiseev and A. Russo: “Implementing an OCL to JML Translation Tool,” 電子情報通信学会技術研究報告, Vol.106, No.426, pp.13–17, 2006

[6] A. Hamie: “Translating the Object Constraint Language into the Java Modelling Language,”In Proceedings of the 2004 ACM symposium on Applied computing, pp.1531–1535, 2004.

[7] G. Leavens, A. Baker, and C. Ruby: “JML: A notation for detailed design,” In Behavioral Specifications of Businesses and Systems (H. Kilov, B. Rumpe, and I.Simmonds, editors), pp.175–188, Kluwer Academic Publishers, Boston, 1999.

[8] G. Leavens, A. Baker, and C. Ruby: “Preliminary Design of JML: A Behavioral Interface Specification Language for Java,” TR98-06, revised version 2003.

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

[10] M. Elaasar, and L.C. Briand: “An Overview of UML Consistency Management,” Technical Report SCE-04-18, Carleton University, 2004.

[11] C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers, and H.M. Dortmans: “An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs,” In Proceedings of Workshop on Consistency Problems in UML-based Software Development II, pp.26–34, 2003.

[12] Eclipse Foundation: “Eclipse Modeling Framework,” `http://www.eclipse.org/modeling/emf/`.

[13] K. Objecten: “Octopus (OCL Tool for Precise UML Specifications),” `http://www.klasse.nl/octopus/`.

[14] J. Warner, and A. Kleppe: “The Object Constraint Language – Getting Your Models Ready for MDA,” 2nd Ed. Pearson Education Inc., 2003.

[15] B.C. Pierce: “Types and Programming Languages,” MIT press, 2002.

[16] Samples of JML specifications `http://www.eecs.ucf.edu/ leavens/JML/examples.shtml`

[17] M. Fowler: “MF Bliki: Closure” `http://martinfowler.com/bliki/Closure.html`

[18] P. Chalin, P. R. James, and G. Karabotsos: “JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML,” Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE),2008

[19] G. Engels, R. Hücking, S. Sauer, A. Wagner: “UML Collaboration Diagrams and Their Transformation to Java,” UML1999 -Beyond the Standard, Second International Conference, pp.473–488, 1999.

[20] W. Harrison, C. Barton, M. Raghavachari: “Mapping UML designs to Java,” Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp.178–187, 2000.

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