特集●ソフトウェア工学

メソッドの自動生成を用いたOCLのJMLへの変換

尾鷲 方志　岡野 浩三　楠本 真二

本論文では OCL (Object Constraint Language) からの JML (Java Modelling Language) への変換手法を提案する．UML/OCL 記述から JML アノテーションつきの Java のスケルトンコードを自動生成する方法については，研究が少ない．従来の研究では OCL 記述から JML 記述を得る方法が提案されているが，データの集まりを表現するコレクションに関するいくつかの重要な機能，とりわけ，繰り返し操作である iterate について対応していない．提案手法では，iterate の操作を生成される Java スケルトンに直接対応するメソッドを記述するという方法により，この問題に対応した．また，OCL 記述からの JML 記述変換方法を具体的に示した．

The paper presents a translation method 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. 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 paper also provides a translation example from OCL into JML.

1　はじめに

OCL は UML 記述に対しさらに詳細に性質記述を行うために設計された言語で，OMG 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 [3] [8].

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 (Java Modeling Language) [9] 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．

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 [4] [6]，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 [2].

開発プロセスの観点から，OCL で記述された要求仕様を機械的に JML に変換することは，コードの検証作業の効率化，精緻化に繋がる．Hamie は文献 [5] 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 [14] 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，Avila らが文献 [1] にて型の扱いなどについて改善を示しているものの，いずれの方法も Collection の対応が不十分であり，iterate の対応が一部の演算に対応しているのみである．しかし，iterate は単純な探索問題や，データベースのモデル化等で広く使われるため，対応すべき問題だと考えられる．

そこで本論文では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 JML 記述への変換方法を具体的に提示し [13]，その変換例を用いて設計開発方法の適用可能性について考察する．

A translation method from OCL into JML by translating the iterate feature into Java methods.

Masayuki Owashi, Kozo Okano, Shinji Kusumoto, 大阪大学 大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

コンピュータソフトウェア, Vol.1, No.0 (1984), pp.1–6.
[研究論文 (レター)] 2009 年 12 月 31 日受付.

以降，2 章で背景について述べ，3 章で変換方法について述べ，4 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[11] は OMG 標準の 1 つで，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ために導入された．

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．また，Design by Contract[10] 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[9] は Java プログラムにおいて Design by Contract[10] 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，コンパイルや実行に影響がない．

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 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 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，バグ出しを行うのには実用的に有用とされる．

2.3 関連研究

UML から JML への変換については，Engels らの文献[4] や Harrison らの文献[6] 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．

表 1 既存手法との比較

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

√:supported, -: not supported

Rodion と Alessandra らは文献[14] 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[5] 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．Avila らは文献[1] において，文献[5] においてマッピングされた OCL と JML のコレクションの型の差異を吸収し，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 式への対応が不十分である．また，iterate 式の引数として与えられる OCL 式について，JML や Java においてそのような式の評価機構 (クロージャ) 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もある．例えば，$students$->$select(score > 70)$ に対し，対応する Java メソッド $select(exp)$ を用意した場合，OCL 式で与えられる引数 $score > 70$ が exp として与えられることになり，そのまま評価することができない．

また文献[5] における構文変換技法は概論のみで，精緻な変換方法は触れられていない．このような OCL の JML への変換の対応状況を表 1 にまとめた．

本稿では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し，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．

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．

3 OCL から JML への変換手法

この章では文献[5] 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く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 の型と OCL Essential の一部の型を対象とする．
2. 対象 (部分) 式に型，または型の候補情報が定義でき変換時に利用できるものとする．型として Bool, Integer, Real, String, Collection, Set, Bag, OrderedSet, Sequence を仮定し，Java(JML) のクラスや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 Java や JML で直接扱えないため対象にしない．
4. 使用範囲の狭い Message 型および，message 演算は変換の対象外とする．
5. OclVoid は Undefined 定数のみをもち，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が，3.3 節にて議論する．

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 OCL の定義書 [11] に従い対応した．

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ブ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．

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．

3.2 構文変換

ここでは主要な構文に対して，文献 [5] の構文変換を拡張した構文変換法を与える．文献 [5] にならい，OCL 式から JML 式への変換関数の表記を μ，特にコレクションのリテラルに対する変換を μ_e で与える．

表 2 μ 変換 Collection loop features

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

文献 [5] 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 Integer, Collection を持つ部分式をそれぞれ i_m, c_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 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．なお，collectNested() は各サブクラスで個別に変換定義される．

3.2.2 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 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) 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 の引数の OCL 式を iteration 込みで評価するメソッド 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，クロージャを持たない JML において変換時に作成されるメソッド m の引数として OCL 式あるいはそれに相当する Java 式 (JML 式) を持つことはできない．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ものとなる．

iterate feature は Java 1.5 よりサポートされている 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，
- ナビゲーションを非終端記号として持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．

Context[$\mu(a_1$ $->$iterate(e; $init$ | $body$))] に対し，初期化式 $init$ で T_1 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 T1 mPrivateUseForJML01() {
  μ(init);
  for (T2 e: μ(a1)){
    res = μ(body)}
return res;
}
```

- 変換構文内で

```
Context[mPrivateUseForJML01()]
```

変換例を以下に与える．

例 1 c $->count(`ocl')$ $>$ 0 を考える．このとき *Context*[] は [] $>$ 0 となる．

$\mu(c$ $->count(`ocl')$ $> 0) \Rightarrow$
$\mu((c$ $->iterate(\ e;\ acc\ :\ Integer\ =\ 0\ |\quad if\ e\ =$ $`ocl'$ $then\ acc + 1\ else\ acc\ endif))> 0)$

メソッドとして

```
private int mPrivateUseForJML01() {
  μ(acc : Integer = 0);
  for (String e : c){
    acc = μ( if e = 'ocl' then acc + 1 else acc
endif)
  }
  return acc;
}
```

→

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

μ 変換の本体内

```
  mPrivateUseForJML01()> 0
```

Java 1.4 に対しては上述の for 文の代わりに，Iterator を用いる．変換テンプレートを以下にあげる．

```
private T1 mPrivateUseForJML01() {
  μ(init);
  Iterator i=μ(a1).getIterator();
  while(i.hasNext()){
    res = μ(body);
    i.next();
  }
  return res;
}
```

3.3 変換の評価

変換できるクラスの違いは明確なため，質の比較のために本手法の一部である OCL 式レベルでの置き換えを適用することにより既存手法での変換が可能な例題をもとに変換の質について考察する．

変換する例題は，在庫管理問題[12] における倉庫をモデル化したものであり，倉庫は商品が収納されているコンテナを複数持ち，そのコンテナにはそれぞれ固有の ID を持つ．また，すべてのコンテナに含まれる全品目を管理する全品目リストを別に持つとする．

コンテナの集合を containerSet，全品目リストを allItemList，商品を item と表すと，倉庫に関する上記の制約は以下のように OCL 式で表すことができる．

```
Context Storage
inv: containerSet->isUnique(containerID)
inv: allItemList->collect(item) = containerSet
  ->collect(container)->collect(item)
```

上記の一つ目の不変式を表す OCL 式を提案手法で以下の様に μ 変換を適用し，

```
/*@ invariant μ(containerSet->collect(containerID
  )->forAll(x,y| x<>y implies x.value <> y.value))
@*/
```

iterate 式で置き換えていくと，

```
/*@ invariant μ(containerSet
  ->iterate(containerID;res:Bag(containerID)
    =Bag{}|res->includeing(containerID))
      ->forAll(x,y| x<>y implies x.value <>
y.value)) @*/
```

となり，iterate に対応するメソッドを生成し最終的な JML 記述は以下のようになる．

```
/*@ invariant (\forall ContainerID x, y;
  iterateMethod01().has(x) &&
      iterateMethod01().has(y);
    !x.equals(y) ==> !x.value.equals(y.value));
@*/
```

既存研究において，文献[5] では，直接 isUnique の変換に関する言及がなく，また，本研究のように OCL での置き換えを適用し，collect と forall の組み合わせで表記しなおした場合に関しても，collect の変換については JML に\collect 命令を追加することを提案するにとどまっているために対応できない．また，文献[14] にて，collect への変換には対応していて，それを用いた表記は以下のように表される．

```
/*@ invariant (\forall ContainerID x, y;
  JMLTools.collect(
      containerSet, "containerID").has(x)
    && JMLTools.collect(
        containerSet, "containerID").has(y);
      !x.equals(y) ==> !x.value.equals(y.value));
@*/
```

可読性の観点では，文献[14] の手法の変換は本手法の変換結果に比べ優れているが，collect の引数が，containerID->toLower() のように動的な式評価を要する場合，速度面で遅いとされるリフレクション API を用いて文字列として与えられた引数を評価するよりは，構文木をそのまま用いることができる本手法の方が速度面において優位であると推測できる．この点は実装してから今後評価したい．

また，メソッドの生成により処理と関係ないコードが追加されることによる処理時間への影響については，コンパイル時にコメント化するなどのオプションを用意することにより必要なコードとの切り分けが可能になるため回避できる．

全体的な変換の正しさに関して考察する．基本方針で述べたように，OCL 式レベルでの等価変換は仕様定義書で述べられている意味定義に従い，Collection の演算の内，size(), isEmpty(), notEmpty(), union(), intersection(), including(), Set と OrderedSet の excluding() , as 演算，OrderedSet 固有演算の全てについては直接 Java のクラスファイル (CF) のメソッドに対応付けている．この正しさについては OCL 仕様定義書には OCL で事前，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[7] 事前，事後条件とを対応させること，また，変換したものを検証器に通した際の警告等を利用することで検証器が保証する程度の正しさの確認が可能である．

コレクション演算については，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 式を使っているのは sum,includes のみで，この変換は直接対応付けているため妥当であ

る．Java の CF を使っているのは前述のとおりで，上述の方法で確かめる必要がある．

残りの loop feature は OCL の仕様定義にしたがっているため iterate 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，メソッド生成部に関する制約を適切に記述し，検証器を通すことで構文変換の正しさを確認できる．

3.1 で触れた OclVoid について，null とは違う点として，True or Undefined のような一部の論理演算においては OclVoid は式そのものを未定義としては扱わず，正当な評価 (上記の場合は True) を返すというものがある．正確に OclVoid を扱うためにはツール側で対応するオブジェクトを用意し，差異を適切に吸収する必要がある．

また，コレクションの各サブクラス特有の順序関係等の性質を保持するためには文献[1] のように対応するクラスをライブラリの形で定義する必要がある．

4 まとめと今後の課題

本研究では OCL の JML への変換方法を提案し，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広いクラスに対し具体的に示した．

可読性の点においては，一般形に直さずに直接変換できるような形については個別に用意することで対応し，文献[1] で指摘された問題点については，文献[1] と同様の手法を用いて解決したいと考えている．

これらの問題を解決し，本研究において設計したツールを実装し，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できるか，などについて評価した上で，OCL 入力支援手法や，ESC/Java2 などによるソフトウェアモデル検査の併用など，本手法の効果的な適用について考察したい．

謝辞 本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，一部日本学術振興会科学研究費補助金基盤研究 (C) (課題番号: 20500033, 21500036) の助成を受けている．

参 考 文 献

[1] Avila, C., Flores, Jr., G., and Cheon, Y.: A library-based approach to translating OCL constraints to JML Assersions for Runtime Checking, *International Conference on Softw. Eng. Research and Practice*, 2008, pp. 403–408.

[2] Eclipse Foundation: Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.

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

[4] Engels, G., Hücking, R., Sauer, S., and Wagner, A.: UML collaboration diagrams and their transformation to Java, *UML1999 -Beyond the Standard, Second International Conference*, 1999, pp. 473–488.

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

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

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

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

[9] Leavens, G., Baker, A., and Ruby, C.: JML: A Notation for Detailed Design, *Behavioral Specifications of Businesses and Systems*, (1999), pp. 175–188.

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

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

[12] 尾鷲方志, 岡野浩三, 楠本真二: 在庫管理プログラムの設計に対する JML 記述と ESC/Java2 を用いた検証の事例報告, 電子情報通信学会論文誌, Vol. J91D,No. 11(2008), pp. 2719–2720.

[13] 尾鷲方志, 岡野浩三, 楠本真二: メソッドの自動生成を用いた OCL の JML への変換ツールの設計, ソフトウェア工学の基礎 *XVI,* 日本ソフトウェア科学会 *(FOSE 2009)*, 近代科学社, 2009, pp. 191–198.

[14] Rodion, M. and Alessandra, R.: 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, 2006, pp. 13–17.