Javaプログラムからのオブジェクトフローグラフの生成

堀田　吉彦[†]　　大久保　弘崇[††]　　粕谷　英人[††]　　山本　晋一郎[††]　　齋藤　邦彦[†††]

[†] 愛知県立大学大学院情報科学研究科　[††] 愛知県立大学情報科学部情報システム学科
[†††] 滋賀大学経済学部情報管理学科

A Method for Generating Object Flow Graphs from Java Programs

Yoshihiko HOTTA[†]　Hirotaka OHKUBO[††]　Hideto KASUYA[††]
Shinichiro YAMAMOTO[††]　Kunihiko SAITO[†††]

[†]Graduate School of Information Science and Technology, Aichi Prefectural University
[††]Faculty of Information Science and Technology, Aichi Prefectural University
[†††]Faculty of Economics, Shiga University

1　はじめに

オブジェクト指向プログラムの開発・保守・テストにおいて，自然言語で記述されたドキュメントやソースプログラムからそのプログラムの意味を読み取り理解することは非常に困難である．そこでプログラムの理解を支援するための方法の一つとしてプログラムの内部構造や振る舞いを可視化するという方法がある．可視化することにより構造や振る舞いを直感的に把握することが可能となり，プログラムの理解が容易になる．現在，ソースプログラムを元にしたプログラムの可視化に関するさまざまな研究がなされている．

近年のソフトウェア開発では，オブジェクト指向プログラムを設計する段階で使用するために開発された UML(Unified Modeling Language)[1] が積極的に採用されている．UML はオブジェクト指向プログラムの内部構造や振る舞いを図式で記述するグラフィカルなモデリング言語である．UML はそもそもオブジェクト指向言語との馴染みがよく，広くサポートされているので，本来の目的であるプログラム設計だけではなく，保守過程においても，UML を使った視覚的なプログラムの理解支援ツールは有効である．Tonella らはオブジェクト指向言語のソースプログラムに対してリバースエンジニアリングをおこない，UML 各図 (クラス図・パッケージ図・オブジェクト図・コラボレーション図・シーケンス図・ステートチャート図) を作成する手法を提案している [2]．ここでは UML 図を生成する前のプログラムの中間処理として，オブジェクトフローグラフ (OFG) を用いる．しかし，C++言語での実装しか存在しておらず，なおその実装も公開されていない．

我々はリバースエンジニアリングの手法をオブジェクト指向言語の一つである Java 言語に適用し，その実装を与え，UML 図を生成するシステムの構築することを目標としている．本システムで用いる OFG は，細粒度リポジトリの持つソースプログラムの解析情報を保持していることが特徴である．それによって，この OFG から生成する UML 図をソースプログラムやソフトウェアに付属するドキュメントなどと連携させることができ，UML 図を用いたプログラム理解の為の CASE ツールの制作者を支援することが可能となる．本稿では，UML 図を生成するシステムの一部である Java ソースプログラムから OFG を生成する手法について述べる．

2　オブジェクトフローグラフと UML

この章では UML 図を生成するために利用する OFG と生成される UML 図について述べる．2.1 節で OFG についての説明を行う．また，OFG をソースプログラムから容易に生成するために Java 言語ソースプログラムの抽象化を行う．そのために抽象化した Java 言語とその生成方法について述べる．2.2 節では OFG から生成される UML 図について説明し，UML の一例としてクラス図の紹介を行う．

2.1　オブジェクトフローグラフ

OFG は静的解析によるオブジェクト指向プログラムのモデルの一種である．OFG は，プログラムの実行中にオブジェクトが生成されてから，変数に代入され，クラス属性に保管されたりメソッドの呼び出しに使用されるようなオブジェクトについての流れの情報を表す．OFG はプログラムのデータフローのみを表現し，コントロールフローは反映していない．コントロールフローを反映させないことによって，抽象化した Java 言語プログラムから容易に OFG へ変換することができる．

2.1.1　抽象化 Java 言語プログラム

Java 言語で書かれたプログラムを抽象化 Java 言語のプログラムに変換する．抽象化 Java 言語では全てのオブジェクトのデータフローに関する Java 言語の命令を単純化して記述する．一方，データフローに影響しない全ての命令 (条件文・ループ・オブジェクト型でない変数やパラメータ等) は記述しない．ま

た，抽象化 Java 言語では名前解決を単純にするため，全ての識別子は接頭辞にパッケージ名，クラス名を付加する．この方法を用いることで名前の衝突を防ぐことができる．抽象化 Java 言語の構文では，Java プログラム P は 0 個以上の宣言 D と 0 個以上の文 S から構成されている．抽象化 Java 言語の構文規則を図 1 に示す．

宣言　宣言は，フィールドの宣言，メソッドの宣言，コンストラクタの宣言の三種類に分類できる．以下に Java プログラムの断片とそれを抽象化 Java 言語で記述した例を示す．

ソースプログラム 1

```
public class Sample {
    Object obj;
    public Sample() {
        ...
    }
    public boolean method(Object o) {
        ...
    }
}
```

抽象化 Java 言語プログラム

```
Sample.obj                      フィールド
Sample.Sample()                 コンストラクタ
Sample.method(Sample.method.o)    メソッド
```

文　文は，アロケーション文，代入文，メソッド呼び出しの三種類に分類できる．以下に Java プログラム断片とそれを記述する抽象化 Java 言語のプログラムを示す．

ソースプログラム 2

```
public class Sample2 {
    public boolean method(Object o) {
        ...
        Object obj = new Object();
        method2(fp);
        ...
    }
}
```

抽象化 Java 言語プログラム

```
Sample2.method.obj = new Object();
Sample2.method.this.
    Sample2.method2(Sample2.method.fp);
```

この際，暗黙の対象オブジェクト (this) は明示され，他のプログラムロケーションと同じルールに従って接頭辞が付加される．

抽象化された Java 言語ではプログラムの全ての要素が一意の名前を持つ．同じクラスに属しオーバーロードされている (同じメソッド名だが引数の数や型，返り値の型が違う) メソッドの名前には 1 つづつ増やした整数の接尾辞を加えることで一意にする．

2.1.2　オブジェクトフローグラフ

OFG はノードの集合 N とエッジの集合 E から構成される有向グラフである．OFG のノードは，Java 言語のプログラムロケーションである．プログラムロケーションとはフィールド・引数・メソッドのメンバ変数等のオブジェクトを参照する変数である．

抽象化 Java 言語の構文	エッジ生成のルール	
$P ::= D^*S^*$		(1)
$D ::= a$		(2)
$\mid m(f_1,...,f_k)$		(3)
$\mid cs(f_1,...,f_k)$		(4)
$S ::= x = \text{new } c(a_1,...,a_k);$	$\{(a_1,f_1)\in E,...,$ $(a_k,f_k)\in E,$ $(cs.this,x)\in E\}$	(5)
$\mid x = y;$	$\{(y,x)\in E\}$	(6)
$\mid [x=]y.m(a_1,...,a_k);$	$\{(y,m.this)\in E,$ $(a_1,f_1)\in E,...,$ $(a_k,f_k)\in E,$ $(m.return,x)\in E\}$	(7)

Class scoped identifiers :
a: class attribute name
m: method name
f_1,...,f_k: formal parameters
x, y: program location
a_1,...,a_k: actual parameters
cs: class constructor
c: class name

図 1: 抽象化 Java 言語の構文と OFG エッジ生成のルール

OFG のエッジは，ノード間に図 1 の右側に示すルールに基づいて作成される．エッジは解析対象のプログラムで起こるプログラムロケーション間のデータフロー (代入・引数渡し・呼び出し) を表現している．

プログラムの実行時に，コンストラクタやメソッドの呼び出し (図 1(5),(7) 参照) が起こる場合，実引数 a_i からそれぞれの仮引数 f_i へのエッジが 作成される．またコンストラクタ呼び出しの場合は，$cs.this$(cs は $new\ c(...)$ で呼ばれるコンストラクタ) が参照する新しく生成されたオブジェクトを表すノードから左辺の x へのエッジが作成される．メソッド呼び出しの場合は，呼ばれたメソッド内で $m.this$ となる対象のオブジェクト y から $m.this$ へのエッジが生成される．メソッド m によって返された値がもしあれば左辺の x へのエッジが生成される．

2.2　UML 図

OFG を用いて UML 図を生成する．UML 図には利用目的や視点の異なる 10 種の図がある．そのうちオブジェクト指向プログラムの静的な構造を表すクラス図・オブジェクト図・パッケージ図を生成する．また，動的な振る舞いを表す図として，ステートチャート図，相互作用図 (シーケンス図・コラボレーション図) を生成する． 表 1 は UML 図の分類と名称，また本研究での生成対象であるかどうかを示している．本稿ではソースプログラムから生成した OFG を UML 図に応用する一例として，静的な構造を表す構造図の中で最も代表的な図であるクラス図の生成を採り上げる．

2.2.1　クラス図

クラス図は UML の中で最も重要で，オブジェクト指向システムの記述に最も広く使われている．クラス図はシステムを構

表 1: UML の図名と本研究での生成対象

UML 各図の名称			対象
構造図	クラス図		○
	パッケージ図		○
	オブジェクト図		○
	コンポーネント図		
	配置図		
振る舞い図	ユースケース図		
	相互作用図	コラボレーション図	○
		シーケンス図	○
	ステートチャート図		○
	アクティビティ図		

築するのに使われる中心的なクラスの静的な構造を表す．それぞれのクラスに最も関連のある特徴 (属性とメソッド) は，いくつかのプロパティ (可視性や型など) と共にクラス図に記述される．さらに，クラス図はシステムのクラス間の関係も表している．クラス図はクラス間の通信や相互作用を可能とする構造的な接続を静的に表現する．したがって，クラス図はとても有益な，システム構成に関する多くの設計決定の要約である．

ソースプログラムからクラス図を生成するのは難しい作業である．ある要素を表すか省略するかについての決定はクラス図の使い勝手に大きな影響を及ぼす．そのうえ，クラス間の関係は (ソースプログラムの解析からでは推測できない) 領域知識と合理的設計に従った意味的な情報を持っている．

クラス図を再現する基本的なアルゴリズムでは，クラス間の関係を定義すれば，純粋なソースプログラムの構文解析によってクラス図を得られる．例えば，クラスのフィールドが別のクラス型のオブジェクトへの参照を格納する場合はそのクラス間の関係を推測することができる．

クラス図を再現する基本的なアルゴリズムが持つ 1 つの問題は，宣言された型が本来，スーパークラスやインタフェースに帰すものであり，プログラム内で実際に継承や実装しているクラスのオブジェクト型ではないことがある．一方，OFG に基づいたクラス図を再現するアルゴリズムはソースプログラムから抜き出されたクラス図をサブクラスの存在やインタフェースの実装を表現するように定義することが可能である．

クラス図を再現する基本的なアルゴリズムのもう 1 つの問題は型付けの弱いコンテナの使用である．コンテナの宣言された型から決定されたクラス間の関連は，コンテナに格納されたオブジェクトの型を記述していないのでコンテナに格納されたオブジェクトとの関連が決定されない．OFG に基づいたアルゴリズムは，OFG で定義されたフロー解析を利用することによって，コンテナに格納されたオブジェクトの型の情報を回収し正確な関係を決定することを可能とする．

3 JX-model

本研究では，抽象度の高いモデルである OFG を作成するために JX-model[5] を使用する．JX-model とは Java(Java2 ver.1.4) のソースプログラムの細粒度な XML(eXtensible Markup Language) リポジトリのモデルである．JX-model では 20 個の終端要素と 7 個の非終端要素によって Java ソースプログラムを表現している．ここで，終端要素とは字句情報であり，子ノードにテキストノードのみを持つ．非終端記号は構文情報であり，子ノードにモデルの要素を持ち，テキストノードを持たない．

JX-model は多様な CASE ツールの要求に応えるために，ソースプログラムに直接タグを付加することでインデントや改行などのスタイルも含めて解析対象のソースプログラムを完全な形で保持している．また XML リポジトリからは XSLT・DOM・SAX などの XML 文書を取り扱う技術を用いて情報を取り出すことが可能である．

JX-model に基づいた XML リポジトリである XSDML(eXtensible Software Document Markup Language) は，CASE ツールプラットフォーム Sapid[3] の一部で Java 言語を対象として解析をおこなう Japid[4] が用いるフォーマットである．また JX-model は 1 つの Java ソースプログラムを対象にした解析をおこなうが，ソースプログラムのファイル間に跨がった参照の情報を扱うために Sapid Jtool[6] を利用することができる．

本研究では細粒度の XML リポジトリの要素それぞれに振られた一意な ID を保持しながら，OFG で用いるモデルとしての抽象化を行う．細粒度リポジトリの持つ情報を利用することでソースプログラムとの密接な連携が可能な UML の各図を生成することができる．

4 OFG-model

本章では 2 章で述べた OFG の実装をおこなうとともに，細粒度リポジトリの持つ情報を組み合わせたスキーマ OFG-model を提案する．OFG-model は Java 言語を対象とし Java ソースプログラムの OFG を XML を用いて表現する．OFG-model は 2 章の抽象化 Java 言語にあたる表現を OFG-model Layer1 として，OFG のノードとエッジを持つグラフ構造を OFG-model Layer2 として設計した．

OFG-model Layer1 は 8 個の非終端要素と 5 個の終端要素から構成されている．前節の JX-model と同様に，終端要素は字句情報を扱い，子ノードにはテキストノードのみを持つ．また，非終端記号は子ノードに OFG-model の要素を持ちテキストノードは持たない．テキストノードは Java プログラムの識別子や演算子に相当する．OFG-model Layer2 では OFG-model Layer1 の要素に加えてノード・エッジの要素が加わる．4.1，4.2 節では OFG-model Layer1・OFG-model Layer2 それぞれについての説明を行い，4.3 節では JX-model から OFG-model を作成する過程を述べる．

表 2: OFG-model Layer1 の要素

非終端要素	Java の要素	終端要素	Java の要素
Package	パッケージ	pid	パッケージ識別子
Class	クラス	cid	クラス識別子
Attr	フィールド	vid	変数識別子
Method	メソッド	mid	メソッド識別子
Param	パラメータ	op	演算子
Constr	コンストラクタ		
Locvar	メンバ変数		
Progloc	オブジェクト型の変数		

4.1 OFG-model Layer1

OFG-model Layer1 は，Java 言語のオブジェクトのデータフローの要素のみに特化した，抽象化された Java 言語プログラムを記述するためのモデルである．表 2 は OFG-model Layer1 の要素とそれに対応する Java 言語の要素を表している．例として, 図 2 の簡単な Java ソースプログラムを用いると，解析対象のソースプログラムの 4 行目は，抽象化 Java 言語プログラムでは図 3 の様に記述する．それを OFG-model Layer1 を用いた XML で記述すると図 4 のようになる．また，図 4 の中で Class・Attr・Constr・Parame などが持っている属性 id の値は，JX-model において対象の要素に一意に割り振られた id から取得しており，OFG-model Layer1 においてもその一意性を維持する．

```
 1:public class A{
 2:    Object b;
 3:    public A(Object c)  {
 4:        b = c;
 5:    }
 6:    public boolean method(Object p) {
 7:        A d = new A(p);
 8:        return true;
 9:    }
10:}
```

図 2: 解析対象ソースプログラム

```
A.b = A.A.c
```

図 3: 抽象化 Java 言語プログラム (4 行目)

```
<Progloc>
  <Class id = "s5888802817">
    <cid defid = "s5888802817">A</cid>
    <op>.</op>
    <Attr id = "s5905580036">
      <vid defid = "s5905580036">b</vid>
    <Attr>
  </Class>
</Progloc>
<op sort="assignment">=</op>
<Progloc>
  <Class id = "s5888802817">
    <cid defid = "s5888802817">A</cid>
    <op>.</op>
    <Constr id = "s5913968642">
       <cid defid ="s5913968642">A</cid>
       <op>.</op>
       <Param id = "s5905580035">
          <vid defid="s5905580035">c</vid>
       </Param>
    </Constr>
  </Class>
</Progloc>
```

図 4: OFG-model Layer1(4 行目)

OFG-model Layer1 は Java プログラムの持つブロック・条件文・ループなどの構造は一切取り除き，データフローに関わる以下の三要素のみを記述している．

- オブジェクト型の変数への代入
- メソッド・コンストラクタの呼び出し
- 引数の参照渡し

また，特殊な “Progloc” として，疑変数 “this” と “return” を扱っている．“this” はカレントオブジェクトへのポインタであり，“return” はメソッドの返り値である．

OFG-model Layer1 は Java ソースプログラム 1 つに対して 1 つの XML 文書を生成する．OFG-model Layer1 で記述する抽象化 Java 言語プログラムに図 1 で示したエッジ生成のルールを適用することで，OFG を生成することができる．生成された OFG は OFG-model Layer2 で表現される．

4.2 OFG-model Layer2

OFG-model Layer2 は本稿の目的である細粒度リポジトリの情報を保持した OFG を記述するモデルのスキーマである．表 3 は OFG-model Layer2 の要素とそれに対応する OFG の要素・Java 言語の要素を示している．OFG-model Layer2 は 10 個の非終端要素と 5 個の終端要素から構成されている．例として，図 2 の簡単な Java ソースプログラムを用いる．Java ソースプログラムから生成された OFG-model Layer1 の XML 文書 (図 4) を通じて，図 5 の OFG-model Layer2 の XML 文書に変換を行う．OFG-model Layer2 はノードの集合とエッジの集合から構成される有向グラフの構造を持っている．図 5 では 2 つのノード (“A.A.C”，“A.b”) とそのノード間のエッジを表している．ノードおよびエッジの持つ id 属性は生成時に一意になるように割り振る．図 6 で図 5 の OFG-model Layer2 で記述されているの OFG のイメージ図を示す．

表 3: OFG-model Layer2 の要素

非終端要素	Java の要素	終端要素	Java の要素
Package	パッケージ	pid	パッケージ識別子
Class	クラス	cid	クラス識別子
Attr	フィールド	vid	変数識別子
Method	メソッド	mid	メソッド識別子
Param	パラメータ	op	演算子
Constr	コンストラクタ	非終端要素	OFG の要素
Locvar	メンバ変数	Graph	OFG
		Node	OFG のノード
		Edge	OFG のエッジ

4.3 OFG-model の生成

最初に，ソースプログラムを解析しタグ付けを行った JX-model に基づく XSDML 文書群を生成する．次に，ソースプログラムのファイル単位の XSDML 文書群から OFG-model Layer1 に基づく XML 文書群への変換の処理を行う．最後に OFG-model Layer1 に基づく XML 文書群の統合と OFG ノード，エッジの生成等の変換を行い，1 つの OFG-model Layer2 に基づく XML 文書を生成する．図 7 にデータの流れを示す．

```
<Node id="s5905580036">
  <Class id = "s5888802817">
    <cid defid = "s5888802817">A</cid>
    <op>.</op>
    <Attr id = "s5905580036">
      <vid defid = "s5905580036">b</vid>
    <Attr>
  </Class>
</Node>
<Node id = "s5905580035">
  <Class id = "s5888802817">
    <cid defid = "s5888802817">A</cid>
    <op>.</op>
    <Constr id = "s5913968642">
      <cid defid = "s5913968642">A</cid>
      <op>.</op>
      <Param id = "s5905580035">
        <vid defid = "s5905580035">c</vid>
      </Param>
    </Constr>
  </Class>
</Node>
<Edge id = "e0000000001"
 source = "s5905580035"
 target = "s5905580036"
>
  <op sort = "assignment">=</op>
</Edge>
```

図 5: OFG-model Layer2(4 行目)

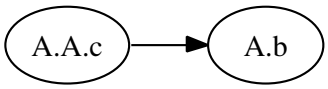

図 6: OFG のイメージ (4 行目)

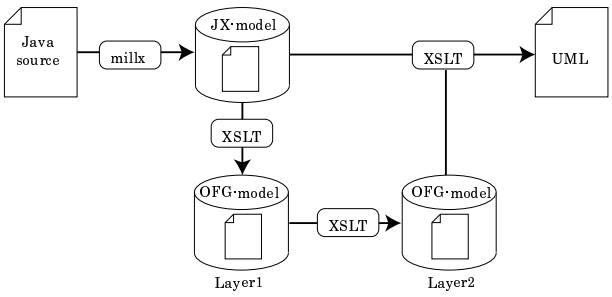

図 7: UML 図生成の流れ

4.3.1 JX-model の XML リポジトリ生成

この処理は，対象となる Java ソースプログラムを解析しタグ付けをおこなった JX-model に基づく XSDML 文書の生成を行う．

JX-model は一つの Java ソースプログラムを対象としておりソースプログラムのファイル間の参照を表現しないので，Jtool を利用してファイル間参照を解決する．XSDML 文書の生成および，ファイル間参照解決の Jtool のコマンドは Sapid に実装されており，それを用いる．

例として，図 2 のプログラムから変換を行うと，ブラウザのドキュメントツリー表示で見て 139 行の XSDML 文書が生成される．

4.3.2 OFG-model Layer1 への変換

この処理は，XSDML 文書から，XSLT を用いて構造を変換することで抽象化された Java 言語を記述する OFG-model Layer1 に基づく XML 文書の生成を行う．解析対象の Java ソースプログラム・XSDML 文書・OFG-model Layer1 に基づく XML 文書はそれぞれ 1 対 1 の対応関係にある．

例として，図 2 のプログラムから生成した XSDML 文書を OFG-model Layer1 に基づく XML 文書に変換すると，図 8 の抽象化 Java 言語プログラムにタグ付けを行った，ブラウザのドキュメントツリー表示で見て 96 行の XML 文書が生成される．

```
A.A(A.A.C)
A.b=A.A.C;
A.method(A.method.p)
A.method.d = new A.A(A.method.p);
```

図 8: 抽象化 Java 言語プログラム (図 2)

4.3.3 OFG-model Layer2 への変換

この処理は，最初に OFG-model Layer1 に基づく XML 文書群を XSLT を用いて統合し，次に OFG の構造を持つ OFG-model Layer2 に基づく XML 文書への変換を行う．統合の部分では，解析対象の Java ソースプログラムの数だけある OFG-model Layer1 に基づく XML 文書を 1 つの XML 文書に統合する．変換の部分では，OFG-model Layer1 のプログラムロケーションの要素を OFG-model Layer2 のノード要素に変換し，ソースプログラム間に跨がった部分も含めてプログラムロケーション同士の関係を，図 1 のエッジ生成の構文のルールに基づき OFG-model Layer2 のエッジ要素の生成を行う．

例として，図 2 のプログラムから生成した OFG-model Layer1 に基づく XML 文書から OFG-model Layer2 に基づく XML 文書への変換を行うと, ブラウザのドキュメントツリー表示で見て 111 行の XML 文書が生成される．図 9 は生成された OFG-model Layer2 に基づく XML 文書で記述されている OFG のイメージ図である．

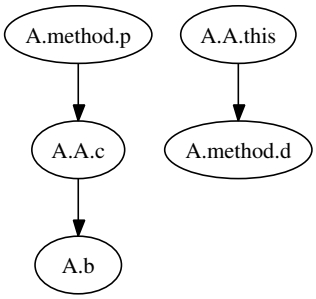

図 9: OFG のイメージ (図 2)

5 OFG-model の応用

本章ではオブジェクト指向プログラムから UML 図を生成する時に，OFG-model を応用する方法について述べる．ここでは例として UML 図の中で代表的な静的構造図であるクラス図の生成について考える．クラス図はシステム内にあるクラス群とそのクラス間の関係から構成されている．オブジェクト指向プログラムからクラス図を生成する手順もその２つの部分に分かれる．まず，クラス図内のクラス群の記述に対しては OFG を用いなくとも静的な構文解析のみで可能である．5.1 節でクラス図内のクラス群の生成について説明する．次にクラス図内のクラス間の関係を記述する．これには静的な構文解析のみを用いる方法と OFG を用いる方法がある．5.2 節で両者を比較する．

5.1 クラス群の記述

クラス図は用途や視点によって様々な抽象度で表現される．一般的には三段の矩形の中にクラス・フィールド・メソッドを記述するが，可視性やフィールドの型，メソッドの返り値や引数の型なども加えて記述されることがある．また，矩形内にクラス名だけを記述した抽象度の高い表記法もある．例として図 10 では図 3 のソースプログラムについて，それぞれの抽象度で記述している．

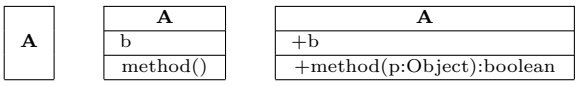

図 10: 抽象度の異なるクラス図 (図 2)

JX-model に基づく XSDML 文書を用いて XSLT で必要な情報を抽出し，ユーザの与えるオプションで抽象度を変更可能なクラス図のクラス群のリストを出力する．

5.2 クラス間の関係の記述

ここではクラス間の関係の記述について説明する．UML で定義されているクラス図におけるクラス間の関係は数多くあるが，本研究で生成するクラス図では，クラス間にはアソシエーション／集約・ディペンデンシィ・汎化・実現の 4 種類の関係が存在する．

アソシエーション／集約 インスタンス内に別のインスタンスへのコネクションを持つ関係である．アソシエーションと集約は UML の仕様には厳密な定義があるが，Java ソースコードから UML へのマッピングを行う場合は実質的な違いはないので同じ関係として扱う．Java 言語ではクラスが他のクラスの型のフィールドを持つ場合にこの関係にあたる．

ディペンデンシィ アソシエーションや集約よりも弱い関係である．Java 言語ではクラスに属するメソッドの引数やフィールドが他のクラスの型である場合にこの関係にあたる．

汎化 汎用的なクラスとより特殊なクラスの関係である．Java 言語ではあるクラスが別のクラスを継承している場合この関係にあたる．

実現 リアライゼーションは仕様のクラスと実装のクラスの間にある関係である．Java 言語ではあるクラスがインタフェースの実装を行っている場合この関係にあたる．

汎化と実現はそれぞれソースプログラムに対し静的な構文解析を行い，クラス宣言部で予約語 “extends”，“implements” の後に来るクラス (インタフェース) の識別子を探すことで関係を取得することができる．しかし，アソシエーション／集約とディペンデンシィの関係を静的な構文解析で決定しようとするとオブジェクト型変数の宣言された型と実際の型の差の問題が起こる可能性がある．

例として，図 11 のプログラムから OFG を用いない方法でクラス図を生成するとする．この時クラス A から，ユーザ定義でないライブラリのインタフェース Lib への関係が生成されてしまう．クラス B を見るとわかるように，クラス A のインスタンス生成時にクラス C のインスタンスが渡っている．クラス A のフィールド b には，実際にはコンストラクタから渡されたクラス C のインスタンスが格納される．ユーザの定義したシステムの図という視点でクラス図を見たい時には，クラス A とインタフェース Lib の実装であるクラス C との関係がある方が望ましい．図 13 の OFG を用いたクラス図の生成を行うと，クラス A のフィールド b の実際の型を判定し，クラス C とクラス A の間にアソシエーション／集約の関係を生成することができる．

```
public class A{
    Lib b;
    public A(Lib x){b = x;}
}
public class B{
    public m(){C c = new C(); A a = new A(c);}
}
public class C implements Lib{...}
```

図 11: 宣言の型と実際の型

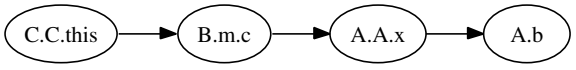

図 12: クラス C のインスタンスを注目した OFG

クラス図におけるクラス間の関係は JX-model に基づく XSDML 文書群と OFG-model Layer2 に基づく XML 文書を XSLT を用いて変換することで．クラス間の関係のリストを出力する．

最後にクラス群のリストとクラス間の関係のリストを統合し，クラス図を表す XMI[7] 文書を生成する．XMI とは UML モデルを含むメタデータを交換するためのストリームフォーマットである．

6 おわりに

本稿では，OFG を応用して生成されるオブジェクト指向プログラムの UML 図から細粒度リポジトリの持つ情報を利用するためのスキーマを提案し，Java 言語を対象とした OFG-model について述べた．OFG-model を応用した UML 図を用いれば，オブジェクト指向プログラムを直感的に理解するために効果的な可視化表現が可能になる．それだけではなく，ソフトウェアに付属するドキュメントやソースコードブラウザとの連携を行うような各種のプログラム理解支援のために CASE ツールから利用することができる．また，プロトタイピング型の開発過程で，ソースプログラムからリバースエンジニアリングした UML 図を使って，プログラム変更するための再設計を行い，それに基づいて実際のソースプログラムの変更を行うサイクルを支援することも可能となる．

今後の課題としては，OFG を応用する UML 図を生成する部分の実装を行う必要がある．UML 図は，XMI で記述するが，XMI にどのように細粒度リポジトリの情報を保持させるかを工夫する必要がある．さらに，UML 図とソースプログラムの要素がハイパーリンクで結合したソフトウェアブラウザを作成し，実際にオブジェクト指向プログラムの理解支援やリファクタリングを行えるツールを作成したいと考えている．

謝辞 ご指導して頂いた愛知県立大学情報科学部の稲垣康善教授および山本研究室の皆様に感謝します．また，研究のベースとなった Sapid プロジェクトの皆様に感謝します．

参考文献

[1] Unified Modeling Language (UML)1.5 specification. OMG, March 2003.

[2] P. Tonella, A. Potrich: Reverse Engineering of Object Oriented Code. Springer, 2005.

[3] 福安 直樹, 山本 晋一郎, 阿草 清滋: 細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid, 情報処理学会論文誌, Vol.39, No.6, pp.1990–1998, June 1998.

[4] Y. Hachisu, S. Yamamoto and K. Agusa: A CASE Tool Platform for an Object Oriented Language , IEICE Trans. on Information and Systems, Vol.E82-D, No.5, pp.977–984, May 1999.

[5] 吉田 一, 山本 晋一郎, 阿草 清滋: XML を用いた汎用的な細粒度ソフトウェアリポジトリの実装 , 情報処理学会 OO2002 シンポジウム, pp.83–90, August 2002.

[6] K. Maruyama, S. Yamamoto: A CASE Tool Platform Using an XML Representation of Java Source Code, Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM04), pp.158-167, Sep. 2004.

[7] OMG:XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm