B-04

業務フロー図に含まれる仕様を形式表現する手法の提案について

Methodology of the Formal Description for Flow Diagrams

濱野 義満†　　銀林 純†
Yoshimitsu Hamano　　Jun Ginbayashi

1. はじめに

エンタープライズ系業務アプリケーションを設計する際，業務の流れについてはフロー図などで表記する．その表記には様々な方法があり一定していない．

業務フロー図は，要件定義をはじめ，ユーザーインタフェース設計といった外部設計にも利用されている．フローチャート図もフロー図に含めるとすると，詳細なプログラム設計にまで利用されている．

これらのフロー図を記述する標準的なツールは少ないため，Excel 等で自由に描かれることが多い．実際，UML ツールや BPM ツールなどを用いて，精緻な設計が行われるのは，利用範囲が限られている．一方，フロー図以外の表形式で記述された設計書から，設計情報を XML 形式などで取出すのは比較的容易である．取り出した XML 形式の設計情報を，設計書間の整合性の保持に利用し，設計品質を高め，生産性を高める取り組みがある．[1][2]

しかし，業務フローやフローチャートを一度 Excel などで描くと，そこから本質的な仕様の情報だけを XML 形式などで取り出すことは難しい．その結果として，フロー図のようなグラフィカルな設計書のレビューや，整合性の確認は，人の目で行うことになる．特に，大規模なエンタープライズ系のアプリケーション開発の場合，レビューの作業量は膨大となり，レビューの難易度も高まる．従って，設計品質の確保に大きなコストがかかる．

グラフィカルな設計情報を標準化するため，これまで，いくつかの取り組みがある．Eclipse Modeling Framework[3]では，設計情報を XML 形式で表記する方法が提案されている．また，ビジネスプロセスの厳密な表記法である BPMN[4]を具体化した，実行可能なビジネスプロセスモデリング言語である WS-BPEL[5]でも，XML 形式での表記法を規定している．その他，これまでに，仕様記述の方法として，XML 形式を用いた取り組み[6][7]はあるが，実際の適用範囲は限られている．

その他，XML 形式以外の方法で仕様表現を拡張する取組みも見られる．Executable UML[8]は，UML を拡張し，アクション言語を用いて実行可能な仕様を記述している．また，自然言語で書かれた仕様書の単語を VDM++のようなフォーマルメソッドを用いて，形式表現することで，仕様を精緻に記述するような取り組みも報告されている．[9][10]

本論文では，論理的な意味を持つシンプルな言語としてオブジェクト指向言語を用い，業務フローに含まれる本質的な設計情報を形式表現する手法を提案する．

2．業務フローの課題と分析

2.1　グラフィカルな設計書の課題

業務フローやフローチャートのようなグラフィカルな設計書は，基本的には明確な記述ルールがあり，その記述ルールに従う限り，曖昧さはなく，設計者が意図したことを，読み手が正確に読み取ることができる．しかし，自由度が高いため，記述ルールに厳密に従うことは結果的に難しい．

具体的には， Excel のようなツールを用いて，お絵かき感覚で自由に表記されることが多い．実際の例として，設計に関する補足情報を記載するために，少々基本ルールを逸脱するようなことが発生する．また，設計者も人間であり，記述漏れや，自明と思われる部分を省略するなど，完全に記述ルールに従うことは難しい．特に，複数人で大量の設計書を短期間で作るような場合，記述に不統一が発生し，曖昧になり易いという課題がある．その結果品質の低下を招き，手戻りが発生しコスト増につながる．

これまでの研究で，グラフィカルな設計書についての評価結果がある．一般的に，フロー図や UML 図のようなグラフィカルな図の記述ルールを，ドメイン固有言語（DSL）の１つと見なして，グラフィック型 DSL と呼ぶ．それに対し，UML の拡張である OCL（オブジェクト制約言語）や VDM++等の仕様記述言語は，テキストで記述されるため，テキスト型 DSL と呼ぶ．グラフィック型 DSL はテキスト型 DSL に比べ，仕様規模が大きくなると仕様の理解と維持管理が難しいという結果が報告されている．[11]

2.2　設計支援ツールの限界

Excel のような汎用的なツールで，グラフィカルな設計書を作成した場合，その設計書から本質的な仕様情報だけを取り出し，グラフィカルな座標などの付随情報を分離することは容易でない．付随情報とは，仕様に直接関係のない，線の長さやフォント，文字や図形の位置などである．その結果，グラフィカルな設計書は，人が理解するための概要図の役割を持つだけで，設計書に含まれる仕様の情報を直接活用することは難しい．

また，UML をサポートした設計ツールのほか，BPMN あるいは BPEL などをサポートしたツールがある．業務フローの例として，UML のアクティビティ図や BPMN のビジネスプロセス図等がある．

但し，これらのツールを活用しても，やはりグラフィカルな設計書から本質的な仕様の情報を取り出すことは難しい．UML ツールにしても，設計情報の保存形式はツールに依存しており，ツール間でデータのやり取りは難しい．また，設計情報として取り出せるデータもすべての設計情報ではなく，一部の設計情報しか取り出せないことがほとんどである．グラフィカルな設計書から本質的な設計情報を過不足なく XML 形式等で取り出すには，一般的には独自に作りこむことが必要になる．

†富士通株式会社，FUJITSU LIMITED

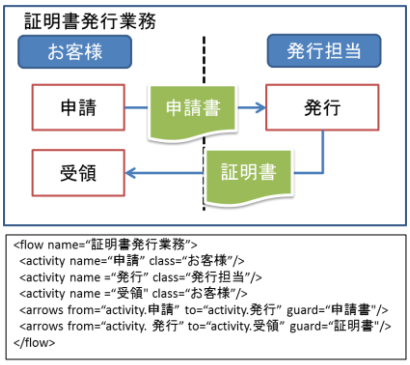

図1. 業務フローとXML表現の例

2.3 XML 形式の課題

グラフィカルな設計書に含まれる仕様を表現するための XML 表現には様々な方法がある．代表的な XML 表現として，図をノードとエッジから構成されると見なし，それぞれを XML タグとして表すことが多い．例えば図 1 は，業務フローとそれに含まれる仕様の情報を XML 形式に表現した例である．これは，付随情報以外の本質的な仕様の情報の形式表現の一例にもなっている．

しかし， XML 形式で設計情報を表記することにも課題がある． 例えば，業務フローの流れの中で，矢印の向きが 1 つだけ反対になったとする．XML の書式としては問題ないが，そのフローは論理的に意味をなさない．従って，論理的に意味のある XML データであることを保証するためには，さらに中身のチェックが必要にある．チェックするには，XML スキーマにより XML の形式を定義し，制限する方法も考えられるが，作成が難しく，可読性が低いことも課題となる．

設計書の数が増えると別の課題が発生する．設計情報の一部を変更する際，再度全体への影響範囲を，人が思考し，判断する必要がある．設計支援ツールを作り，複数の XML ファイルの整合性をチェックすることも可能であるが，XML ファイル間の整合性に関するルールをやはり人が作成する必要がある．

XML データとしての活用範囲も限定される．実際，XML の形式が設計ツール毎に異なるため，ツール間のデータ交換をはじめ，XML データから Java 等のソースコードの形式で自動生成する機能も，限られた範囲にとどまる．

可読性の問題もある．XML 形式の情報から，業務の論理的な中身を直接読み取るのは難しい．特にノードとエッジの数が多くなると，図表に変換しないと人が容易に理解できない．

このような状況から，筆者らは別の方法がないかを模索した．

2.4 形式表現についての仮説

ソフトウェアの設計とは何かということを考えた場合，設計者は，「設計を考える行為」と「設計をした結果を設計書に記載する行為」の 2 つの異なる作業を並行的に行っているといえる．

このことを業務フローの設計作業に当てはめると，設計者は，業務の流れや関連を考え，その結果を業務フローという設計図に描いているといえる．すなわち，本質的な仕様の情報を頭の中に描き，その結果を業務フローという「ビュー（投射図）」に描いている．

このことから，設計書は仕様の情報をある角度から見たビューに過ぎないのではないか，と考えた．例えば，設計書というビューは仕様の一面しか表記できず，すべての仕様が見えるようにするには，複数のビューが必要になる場合もあるのではないか．

また，仕様の表現方法として，XML 形式を利用すると，2.3 節で議論したように，論理的な表現力に限界がある．XML に代わるものとして，テキスト型の仕様記述言語であれば，論理的意味を表記できると考えた．

上記から，以下 2 つの仮説を立てることで，本質的な仕様の表現方法を模索した．

(1) 業務フローに含まれる本質的な仕様の情報は，過不足なく仕様記述言語で表記できる．
(2) 業務フロー図は，仕様の情報をある角度から見たビューである．ビューとは，人が理解したり他者に仕様を伝えたりするための補助ツールである．ビューは仕様の情報を全て表現できなくても良い．

3. Java による形式表現

3.1 オブジェクト指向言語の利用

業務フローの中に含まれる情報として，論理的な意味の最小単位を考えた場合，動作主（主語）と，その動作（述語）がある．さらに動作には入力と出力が存在する．これは，オブジェクト指向言語に対応させると，クラスとメソッドに対応させることで，表記できそうである．（図 2 参照）

また，業務フローの動作の順序は，ノードとノードを結ぶエッジの向きで表現されている．その動作の順序を，外部の動作主（クラス）からの呼び出しで表記できそうである．図 1 の業務フローを Java 言語で形式表現した結果は，図 3 の通りである．本論文では，今後この表記法を **Java 形式表現**と呼ぶ．

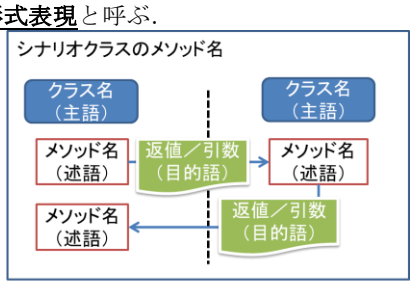

図2. 業務フローと形式表現の対応

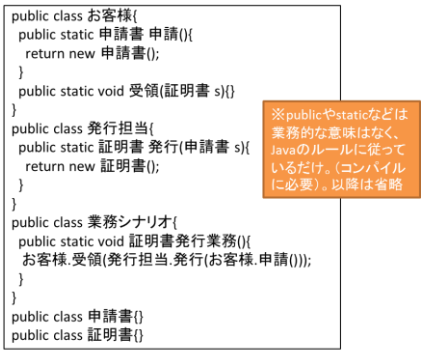

図3. 業務フローとJava形式表現の例

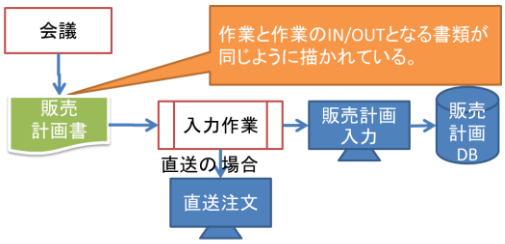

図4.業務フローの改善例(改善前)

```
class 発注依頼担当 {
 void 入力作業(販売計画書 計画書){
  if (計画書.is直送()) {
   在庫管理システム. 直送注文 (計画書);
  } else {
   在庫管理システム.販売計画入力(計画書);
  }
 }
}
```

※コンパイルに必要なpublicやstaticなどは省略

```
発注依頼担当の作業
①入力作業
販売計画書を入力に、それが直送対象であれば、
在庫管理システムの直送注文画面に入力する。
それ以外の場合、計画書の情報を販売計画入力
画面に入力する。
```

図5.Java形式表現とその解釈

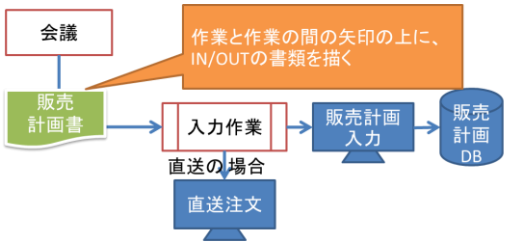

図6.業務フローの改善例(改善後)

3.2 Java 形式表現の手順

業務フローから Java 形式表現に変換するための手順を以下のように定義する.

(1) 処理や操作を「述語」とみなす.
(2) 「述語」の「主語」を明確にする.
(3) 「主語」はクラスに,「述語」はそのメソッドに対応付ける.述語の目的語を「入力」と見なし,述語処理の結果を「出力」とする.「入力」と「出力」を,それぞれメソッドの「引数」と「返値」に対応付ける.
(4) 「業務シナリオクラス」を 1 つ定義する.
(5) 「述語」をつなぐ業務の流れを,「業務シナリオクラス」のメソッドとして記述する.このメソッドの中で,業務の順序と入力と出力の受け渡しを規定する.

このように,業務フローに含まれる本質的な仕様の情報を,オブジェクト指向言語の最も基本的な文法である,クラスによる抽象化,メソッド定義,if 文等だけで表記することができた.

3.3 Java 形式表現と業務フローの関係

業務フローから Java 形式表現に変換するための手順について説明した.しかし本来の設計の手順は,業務フローと同等の仕様の情報を Java 形式表現で直接記述し,ビューを通して,その設計内容を確認する,という順序であるべきである.

ビューは,ある視点,ある角度から見たものであり,あくまで写像にすぎない.従って,あるビューに全ての仕様の情報を表現できないこともあるし,あえて一部の情報しか見せないビューがあってもよい.例えば,複雑な業務の流れを把握するため,主語(クラス)や述語(メソッド)に重要度をつけて,指定した重要度までの情報を表示するビューを作成することなどが考えられる.

4. 評価と検証

Java 形式表現を複数の業務フローに適用し,評価した結果について説明する.

4.1 業務フローの改善例

業務フローを自由な形式で書いた場合,形式の不統一や曖昧さの問題が発生しやすい.結果として,仕様の誤解や手戻りなどの様々な問題が発生する.例として,図 4 のような記述は良く見られるが,業務フローの記述ルールから,厳密には少し逸脱している部分がある.Java 形式表現で記載した場合の一部は,図 5 のようになる.また,論理的な意味は,図に示したように,手順を日本語で解釈可能である.さらにこの仕様の情報を正しい業務フローに戻すと,図 6 のように改善できる.

4.2 複数業務フローの例

業務フローが複数になった場合に,XML 表現をした場合と Java 形式表現した場合を比較する.

例として,図 7 の業務フローを XML 形式で表現すると,業務フローの数だけ XML ファイルが存在する(図 8).

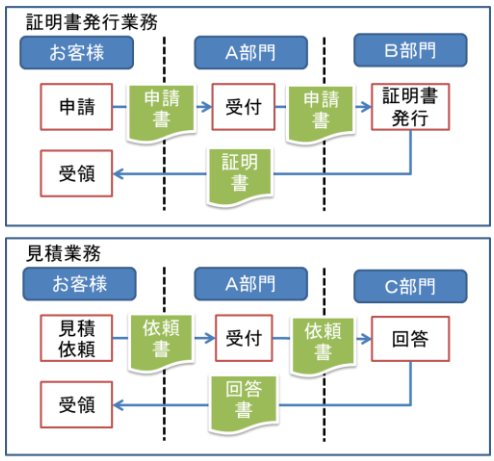

図7.複数業務フローの例

```
<flow name="証明書発行業務">
 <activity name="申請" class="お客様"/>
 <activity name ="受付" class="A部門"/>
 <activity name ="証明書発行" class="B部門"/>
 <activity name ="受領" class="お客様"/>
 <arrows from="activity.申請" to="activity.受付" guard="申請書"/>
 <arrows from="activity.受付" to="activity.証明書発行" guard="申請書"/>
 <arrows from="activity.証明書発行" to="activity.受領" guard="証明書"/>
</flow>
```

```
<flow name="見積業務">
 <activity name="見積依頼" class="お客様"/>
 <activity name ="受付" class="A部門"/>
 <activity name ="回答" class="C部門"/>
 <activity name ="受領" class="お客様"/>
 <arrows from="activity.見積依頼" to="activity.受付" guard="依頼書"/>
 <arrows from="activity.受付" to="activity.回答" guard="依頼書"/>
 <arrows from="activity.回答" to="activity.受領" guard="回答書"/>
</flow>
```

図8.XMLで表現した例

```
class お客様{
 申請書 申請(){return new 申請書();}
 依頼書 見積依頼(){return new 依頼書();}
 void 受領(証明書 s){}
 void 受領(回答書 s){}
}
class A部門{
 申請書 受付(申請書 s){return s;}
 依頼書 受付(依頼書 s){return s;}
}
class B部門{
 証明書 証明書発行(申請書 s){return new 証明書();}
}
class C部門{
 回答書 回答(依頼書 s){return new 回答書();}
}
class 業務シナリオ{
 void 証明書発行業務(){
 お客様.受領(B部門.証明書発行(A部門.受付(お客様.申請())));
 }
 void 見積業務(){
 お客様.受領(C部門.回答(A部門.受付(お客様.申請())));
}
class 申請書{}
class 証明書{
class 依頼書{}
class 回答書{}
```

※コンパイルに必要なpublicやstaticなどは省略

図9. Java形式表現の例

一方，Java 形式表現の場合（図 9），業務フローが 2 つになっても，新しいクラスやメソッドは追加されるが，同じ名前のクラスや同じ名前で同じ引数のメソッドは再利用される．XML 形式を見ても，業務フロー間の関係性は，直観的には分からないが，Java 形式表現の場合，動作主とその操作という形で，論理的に理解できる形に整理される．

4.3　業務フローの詳細化例

業務フローを詳細化すると，概要フローおよび詳細フローと 2 段階で書かれることが多い．業務フローとして図 10 のような例を取り上げる．この例では，「受付」業務は少々複雑で，確認システムへの確認と受付台帳の更新を含んでいる．「受付」業務について，図 10 の概要フローでは処理順までは表現できていないが，「受付」業務を詳細化した詳細フローは図 11 で表現される．これら 2 つのフロー図は，Java 形式表現にすると図 12 のように，1 つの形式表現でシームレスに表現できる．また，同様の

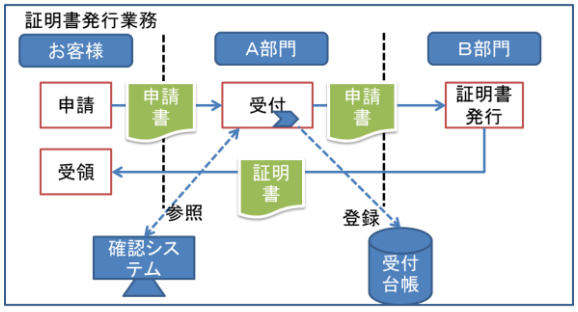

図10.業務フローの詳細化例（概要フロー）

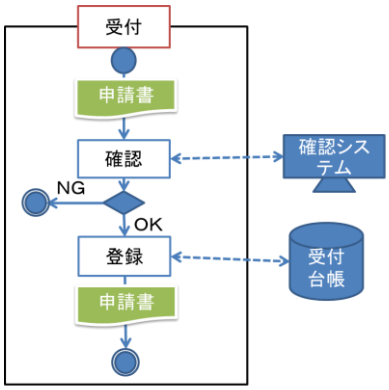

図11.業務フローの詳細化例（詳細フロー）

```
class A部門{
 申請書 受付(申請書 s){
  if (確認システム.確認(s) == false) {
   return null;
  } else {
   受付台帳. 登録 (s);
   return s;
  }
 }
}
```

※コンパイルに必要なpublicやstaticなどは省略

図12.詳細化された形式表現の例

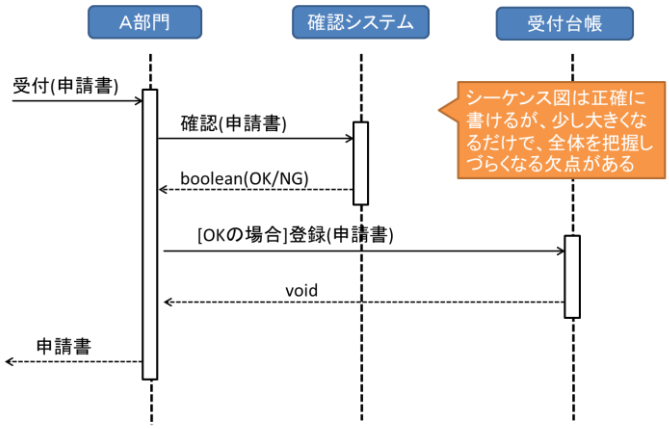

図13.業務フローの詳細化例（シーケンス図）

詳細仕様を，シーケンス図（図 13）の形式で見ることも可能である．簡単な処理であればシーケンス図で見ると分かり易いが，クラスが増え，処理が複雑になると，図が大きくなりすぎ，理解が難しくなる．しかし，Java 形式表現で表記すれば，どちらか便利なビューを採用すればよい．Java 形式表現を主役にすることで得られるメリットである．

4.4　その他の拡張方針について

4.3 節までは，Java 形式表現の基本的な例を示した．形式表現のバリエーションは，Java やその他の言語の表現力を活用することで，拡張することができる．

例えば，並列処理の業務フロー（図 14）は，Java の文法を利用することで，図 15 のように表記できる．さらに

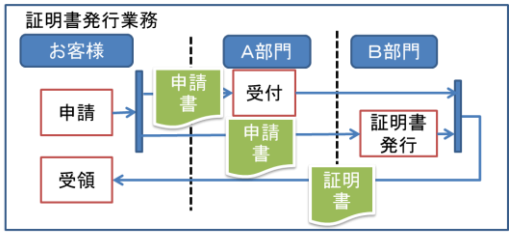

図14. 並列処理（フォーク・マージ）の例

```
class 業務シナリオ{
 void 証明書発行業務() {
 申請書 a = お客様.申請();
 Thread t = new Thread(){void run(){A部門.受付 (a)}};
 t.start();
 証明書 b = B部門.証明書発行 (a);
 t.join();
 お客様.受領(b);
}
```

「受付」を子スレッドとし、「証明書発行」を親スレッドとし、待ち合わせ。

※public、static、Exceptionのthrows文などは省略

図15.並列処理のJava形式表現の例

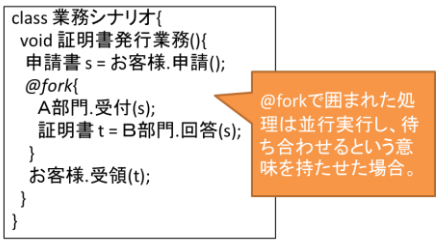

図16.並列処理のJava形式表現の例(文法拡張)

文法を独自に拡張することで，シンプルに記載することができる（図 16）．

その他，複雑なフローに関しても Java をはじめ，その他言語の表現力を活用することで，様々なバリエーションに対応できる．

5. 効果

4 章では Java 形式表現の具体例について示したが，本章では Java 形式表現の効果について説明する．

5.1 Java 形式表現のメリットについて

オブジェクト指向言語である Java を用いて，業務フローに含まれる仕様の情報を形式表現するメリットについて述べる．

本論文では，オブジェクト指向言語の代表として Java を選択したが，利用した文法は，クラスによる抽象化，メソッド定義など，基本的なもののみであり，他のオブジェクト指向言語でも同等の記述ができる．

(1) 主語，述語，入力／出力の明確化

業務フローをグラフィカルなツールで記載する際，作業名は記載しても，その主語は省略可能のため，曖昧になり易い．Java 形式表現では，主語（クラス名）を省略できないため，必然的に明確化される．さらに，述語（メソッド）とその入力（引数）と出力（返値）が省略できないため，必然的に明記される．さらに，主語（クラス）が行いうる述語（メソッド）のリストが明確になる．

その他，ある作業の入力でも出力でもないが，外部からデータを取得し，外部のデータに書き込むような処理も形式表現され，入力と出力とも明確に区別できる．

(2) 同じ設計の粒度で精緻に表記可能

業務フローの記述レベルに関して，あまり規定がないことが多いため，設計の粒度が不揃いになり易い．例えば，ある作業の出力が「販売計画書」だとする．「販売計画書」の中には，さらに「仕入計画」と「チャネル別販売計画」が含まれていたとする．他のフローでは，これらの計画書が必要に応じ参照されることとなる．業務フローでは，それらを区分できるが，これらの関係は別の方法で説明する必要がある．

一方，Java 形式表現を用いると，主語と述語の組み合わせが明確になることで，必然的に粒度がそろう．仮に，主語（クラス）に階層構造があった場合，クラスとサブクラス，あるいはクラスと属性の関係などで階層化することで，設計要素間の関係が明記され，粒度の不揃いの問題はなくなる．

(3) 形式的な整合性をコンパイルによりチェック可能

形式表現の言語として，プログラミング言語を用いたことにより，大きなメリットが発生する． Java 形式表現で書かれた仕様のコードは，形式的にコンパイル可能である．コンパイルを実行することで，クラス名，メソッド名，引数，返値に関し，名称の整合性チェックを自動で行うことができる．この論文で例示したコードもすべてコンパイル可能である．なお，Java バイトコードは利用しないが，コンパイルの機能のうち，語彙解析，構文解析の部分を活用している．

(4) 業務フローの階層化が可能

Java 形式表現により， メソッドのインタフェースレベルの概要フローと，メソッドの処理内容レベルの詳細フローの 2 階層で表記できる．さらに，それらの関係が，メソッドの定義と，その処理の中身という関係でシームレスに表記できる．

5.2 Java 形式表現と XML 表現の比較

Java 形式表現について，XML 表現と比較した結果は以下の通りである．

(1) 可読性が高い

XML 形式は，図の情報としてはシンプルであるが，業務全体を直感的には捉えられない．ビューを通してはじめて理解できる．一方，Java 形式表現は論理的な処理が記述されており，意味を直接理解できる．実際，Java の文法は，主語と述語，目的語が明確であり，そのまま日本語のような自然言語で解釈することが可能である．

(2) 再利用性が高い

複数の業務フローを作成した際，同じ主語（クラス）と述語（メソッド）は再利用できる．一方 XML 表現では毎回書く必要がある．

また，同じ述語（メソッド）でも，入力（引数）が異なれば多相性（ポリモーフィズム）により別の述語として定義できるメリットもある．

(3) 仕様変更と影響範囲の調査が容易

Java 形式表現で書かれた仕様の情報は，コンパイルにより，クラス名やメソッド名，引数と返値などの設計要素の整合性チェックが簡単に実行できる．これは，仕様の情報として，形式的に完結しており，内部矛盾がない状態が実現できているといえる．従って，仕様変更の際の影響範囲の検索等，仕様変更なども確実に実施できる．但し，業務的な意味で正しいかどうかは，人の目でチェックする必要がある．

XML 形式の場合も，整合性のチェックは可能であるが，独自に XML タグ間の依存関係についてのルールを定義して，それに従っているかどうかチェックするツールを作成する必要がある．

(4) 仕様情報がグラフィカルなビューから完全に独立

代表的な XML による形式表現はノードとエッジをそれぞれ定義し，関連させる形式である．従って，グラフィカルな情報としての本質的な要素を持っていると解釈できる．実際，XML 形式で書かれた仕様の情報は，基本的に同じビューで見ることしかできない．

一方，Java 形式表現は，オブジェクト指向言語の文法に基づいて記述され，論理的な意味を持つ．また，本質的な仕様の情報と，そのビューとは完全に分離されている．

従って，Java 形式表現のビューは，様々な角度からビューを作成可能である．例えば UML のクラス図を用いれば，主語に対する述語のリスト，シーケンス図を用いれば，詳細業務フローと同等の情報を見ることができる．

5.3 今後の応用について

オブジェクト指向言語の基本的な文法のみを用いて，業務の流れを，シンプルに形式表現することができた．これまで，フロー図を中心に議論してきたが，Java 形式表現がシンプルなオブジェクト指向言語による表現であることから，業務フロー以外への応用について述べる

(1) 様々なビューを作成可能

Java 形式表現は本質的な仕様であり，グラフィカルなビューから完全に独立している．これまで見てきた業務フローは，Java 形式表現の１つのビューであり，目的に合わせて，様々なビューを作成することができる．

例えば，大規模で複雑な業務のシナリオを Java 形式表現で記述した場合，通常の業務フローでは，図が大きすぎ，全体を把握することが難しくなる．

その場合の解決策として，例えば優先度の高いクラスだけを表示するビューを作成すればよい．あるいは，あるクラスのサブクラスだけを表示する・表示しない，等の制限をつけることで，注目したいクラスだけを表示するビューを作成することが可能である．

このようなビューを作成するための情報は，本質的な仕様の情報ではないが，設計作業を進める上で重要な，設計のための補足情報ともいえる． Java 形式表現の中で，これらの補足情報を管理するためには，注釈文やアノテーションの表記を用いることで仕様の情報とは区別して，統合的に表記できる可能性がある．

(2) 多相性の応用

オブジェクト指向言語の特徴である多相性（ポリモーフィズム）により，類似処理の表現が容易となり，処理記述の再利用などに活かせる．

例えば，類似のクラス（主語）がある場合，共通のメソッド（述語）をスーパークラスのメソッドとして定義し，異なるメソッド（述語）をサブクラスに定義することで，仕様の情報の共通化ができる．さらに必要に応じ，オーバライドを用いれば，同じメソッド（述語）でも異なる処理を定義できる．

(3) 「システム仕様」と「操作」の分離・統一的表記

Java 形式表現で書かれた仕様の情報を，良く見ると，クラス（主語）には，「人や組織」と「システム」の 2 種類に大きく分かれることが分かる．

クラスがシステムの場合は，そのクラスを抽象的な「システム仕様」と解釈する．この仕様を具体化すると，プログラムコードに対応すると考えられる

クラスが人や組織の場合は，そのクラスを抽象的な「操作」と解釈する．この操作を具体化するとテストコードに対応すると考えられる．

抽象的ではあるが，「システム仕様」と人の「操作」を同じ形式表現で統一的に表記できているとみなすことができる．

(4) 仕様アニメーションによる仕様のテスト

(3)の議論をさらに一歩進めれば，人の「操作」のクラスに，変数の値などにバリエーションを持たせ，「システム仕様」のクラスを呼び出すことで，仕様レベルのテストを行うことができる．

これまでの議論では，コンパイルは文法チェックの目的で利用しただけだが，コンパイルでできた Java バイトコードを実行することで，仕様アニメーションが可能になると考えられる．

6. おわりに

Java を形式表現の言語として用いることで，業務フローの仕様情報を形式表現することができた．Java による形式表現は，論理的な意味を直接表現する形式であり，XML による表記よりも様々な点で有力であることを示した．

本論文で提案した立場は，これまでの業務フロー作成の立場と根本的に異なる．業務フローというグラフィカルな図を主役にするのではなく， Java 形式表現で書かれた形式表現を主役とする．また Java 形式表現で書かれた情報が仕様のすべてであり，グラフィカルなビューとは完全に切り離される．一方で，設計者の知りたい自由な角度からビューを通して見ることが理論的には可能である．

但し，これまでの設計作業を大きく変えてしまうことによる問題もある．設計者は，これまでグラフィカルな図を描いた代わりに，直接的に Java 形式表現で仕様を記述することになるため，最初は抵抗を感じるはずである．また，Java 言語の基本的な文法を新たに覚える必要がある．さらに， Java 形式表現を主役として設計を進めるためには，設計支援ツールなどを利用して，効果的なビューを作成し，確認する仕組みを実現することが必要である．特にエンタープライズ系の開発では，開発規模が大きいことが多いため，ビューを通さなければ設計の全体像を把握することが難しい．

現在，Java のプログラミングコードからクラス図やシーケンス図を出すことは様々な UML ツールで実現されているので，既存の技術を利用することで，実現は難しくないことが予想される．

今後，筆者らは，実現に向けた検討を継続する予定である．

参考文献

[1] 上野真由美, 大森麻理：MDA におけるモデル間の整合性保持のアプローチ, 情報処理学会研究報告, Vol.2007-SE-156 (7)
[2] 古家直樹, 村上正敏他：メタモデル上でのルール定義に基づく多ドメイン展開可能な依存関係抽出技術, 情報処理学会研究報告, Vol.2014-SE-184 No.9
[3] フランク バティンスキー他, Eclipse Modeling Framework, 翔泳社, 2005
[4] Business Process Model and Notation (BPMN), http://www.omg.org/spec/BPMN/, Object Management Group
[5] Web Services Business Process Execution Language (WS-BPEL), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, OASIS
[6] 阿部睦：統一的表現に向けた仕様記述方法, 情報処理学会研究報告, Vol.2011-SE-171 No.24
[7] 阿部睦：統一的表現に向けた形式表現と UML との連携, 情報処理学会研究報告, Vol.2011-SE-174 No.8
[8] スティーブ J.メラー他：Executable UML MDA モデル駆動型アーキテクチャの基礎, 翔泳社, 2003
[9] 大森洋一, 荒木啓二郎：自然言語による仕様記述の形式モデルへの変換を利用した品質向上に向けて, 情報処理学会論文誌 プログラミング, Vol.3, No.5, pp.18-28(2010)
[10] 大森洋一, 荒木啓二郎：Refinement Calculus に基づく用語辞書からのフォーマルなモデル構成, 情報処理学会研究報告, Vol.2014-SE-184 No.10
[11] 田中明, 高橋修：ビューポイント DSL を用いたシステム仕様記述に関する考察, 情報処理学会研究報告, Vol.2010-SE-168 No.13