MDAフレームワークのEDA分野への適用

岩政幹人　落水浩一郎

2007 年 11 月 20 日
IS-RR-2007-012

北陸先端科学技術大学院大学 情報科学研究科
〒923-1292　石川県能美市旭台1-1
E-mail:{iwamasa,ochimizu}@jaist.ac.jp

MDAフレームワークのEDA分野への適用

岩政幹人、落水浩一郎

北陸先端科学技術大学院大学 情報科学研究科
〒923-1292　石川県能美市旭台1-1

E-mail:{iwamasa,ochimizu}@jaist.ac.jp

Abstract ソフトウエア工学におけるMDAフレームワークに基づく仕様の変換・合成技術が、EDA(Electric Design Automation)システムの自動化・高品質化にどのように寄与できるかを検証した。
変換定義の言語として、オープンソースプロジェクトであるATL(Atlas Translation Language)を選択し、ATLにおいて、1）設計データDBからの目的コード生成部をATLにて記述する。また2）設計データDB生成に関してはオープンソースプロジェクトであるEMF(Eclipse Modeling Framework)を適用することによりメタモデルから自動生成を構築できることを確認する。さらに3）2におけるXMIファイル保存形式からの内部DBへの読み取り(デシリアライズ)の箇所を対象プログラムと見立てて、EMFではなくてATLにてメタモデルからのＡＰＩ生成を行うことにより、メタモデルを利用したプログラム生成がEDAにおいても有効であることを示す。またこれらの過程で、どこまでimperativeな定義を排除した仕様書が記述できるかを検証した。
結果として、EDAの分野ではメタモデルに基づくモデル変換がインダストリアルな事例においても自動化と高品質化に寄与することがわかった。また木言語理論によるモデル変換技術の形式的な取り扱いについて考察した。

目次

1 はじめに

報告者はEDA(Electric Design Automation)ツール作成のプロジェクトにおいて、EDAツールの中心となる設計データDB(CADシステムの内部DB)を構築するに当たって、メタ情報(以下メタモデルと呼ぶ)からDBおよびファイル入出力処理を自動生成する変換ツールを作成した。この変換ツールはモデル構造の変更への耐性、他の設計データDBにもメタモデルを入れ替えるだけで容易に適用可能となる汎用性、生成されたDB自体にバグがほとんどないという信頼性への寄与、またメタモデルを元にしているので、ファイル構造、DBの構造の相互トレーサビリティが向上した等のメリットがプロジェクト内で評価された。一方、同EDAツールの個別機能である、設計データDBからの目的コード生成（言語はSpecC仕様記述言語)部分においては、変換処理が手続きプログラムで直接的にコーディングされたので、モデルの変更、変換規則の変更に対して、前記、設計データDB生成の自動化で挙げたメリットを享受できなかったという反省点があった。

目的コード生成部においては、設計データのメタモデルの変更および出力対象のコード形式の変更に対する保守性が低いという課題があった。これは変換元と変換対象のメタモデルの構造が異なることにより設計データDBと出力コード形式に一対一の対応が取れず、変換処理にimperativeな処理が必要になり、結果として目的コード生成部の品質（信頼性、保守性等）が確保できなかったことが主な理由に挙げられる。一方、設計データDBの構築においても、変換ツールの作成自体が属人的な職人芸で実現され、当人以外による保守性が低いという問題があった。

モデル変換技術は変換元モデル、変換対象モデルのメタモデルをそれぞれ定義して、このメタモデルに基づいて変換仕様を定義し変換仕様から変換プログラムを自動生成する仕組みである。そこで、モデル変換技術を、上記対象に適用し課題の解決をはかった。着眼点は、いかにimperativeでない仕様記述のみで変換仕様が構成できるかである。一方設計データDBの構築も同様にモデル変換の枠組みで形式化出来ることを検証する。これらの着眼点はプログラムに不具合が発生するのは、プログラムを記述するからであり、プログラムレスすなわち、宣言的な仕様書から自動生成できれば、不具合は整合性のある仕様書を作成することに注力すれば駆逐できるという（個人的な）信念に基づいている。

モデル変換フレームワークの例として、オープンソースプロジェクトであるATL[14]を選択し、ATLにて、1）設計データDBからの目的コード生成部をATLにて記述することを行う。また2）設計データDB生成に関してはオープンソースプロジェクトであ

る EMF[4] を適用することによりメタモデルから自動生成を構築できることを確認し、さらに３） ２における XMI ファイル保存形式からの内部 DB への読み取り (デシリアライズ) の箇所を対象プログラムと見立てて、EMF ではなくて ATL にてメタモデルからの API 生成を行うことにより、メタモデルを利用したプログラム生成が EDA 分野においても有効であることを示す。またこれらの過程で、どこまで imperative な定義を排除した仕様書が記述できるかを検証した。

2 背景

EDA(Electric Design Automation) における設計データ DB とは、設計情報を保存する内部データベースである。EDA ツールとは、GUI などの CAD ツールや他の電子的なデータのインポートによって与えられた設計入力情報に対してツール内部で各種変換や合成 (synthesis あるいは refinement) 処理を行って、最終的に下流のツールが読み込むことができる目的コード（例えば、C 言語や VHDL) を生成する設計自動化のためのツールである。

ここで設計データ DB は、設計情報を格納するオブジェクト指向の内部データベースであり、入力情報からの設計データへの変換、内部データベース上での各種合成処理、および目的コード生成を行うためのデータベース API を持ち、またファイルへの保存（シリアライゼーションと呼ぶ）と読み込み（デシリアライゼーションと呼ぶ）用の API を持っている。設計データ DB は多くの場合、合成処理の実現性や、下流ツールへの目的コード生成容易性を加味したスキーマに基づいて設計される。スキーマは合成処理の正しさを保証するために特定の計算モデルに基づくこともある。

2.1 設計データ DB 事例

SOC(System On a Chip) の上流設計のための EDA ツールである MSC システム [16] は、Message Sequence Chart の図式で書かれた設計データを、内部設計データに変換し、内部設計データ DB 上で様々な仕様合成を行って、これを最終的に下流の EDA ツールの形式 (SpecC 言語で記述された仕様モデル) を生成するツールである。

MSC システムの設計データ DB は、C#言語にて記述され、C#のオブジェクト指向の特徴を利用してクラス、プロパティ、インスタンスを管理し、インスタンス間に成立する関連づけ、合成関係を記録するための独自のフレームワークを有している。ここで設計データ DB のモデルを仮に VSM(Visual Specification Model) と呼ぶことにする。

MSC システムを製造するに当たって作成した変換ツールに「DB ビルダ」がある。「DB ビルダ」は対象とする設計データのスキーマをクラス図に準拠した形式 (メタモデル相当) で与えると、これから DB コアの生成、DB 上の設計データから SpecC 形式のテキストファイルへのコード生成、DB 上の設計データの外部ファイル (XMI 形式準拠) への出力（シリアライズ）と読み込み（デシリアライズ）機能を備えている。

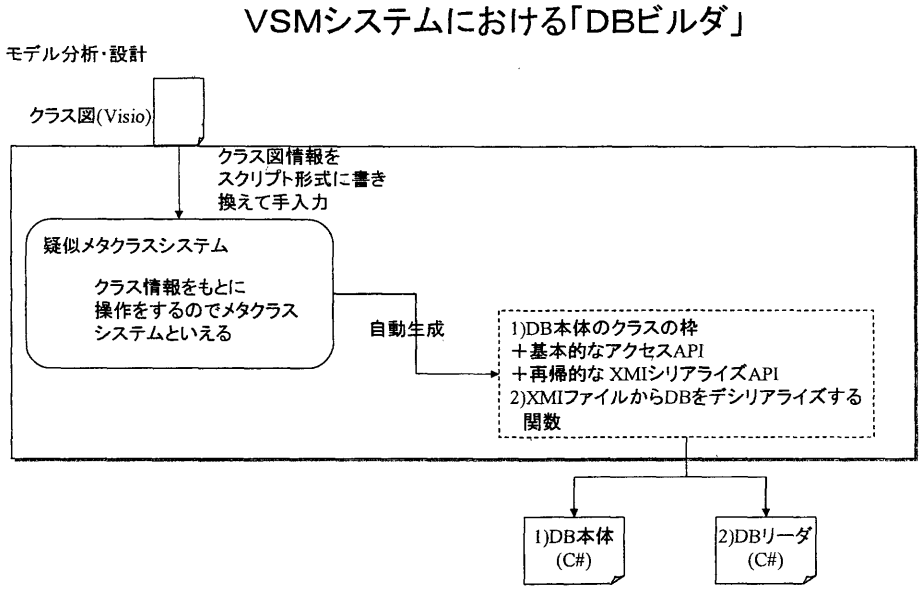

図 1: 従来のシステム

2.2 事例における課題

目的コード (SpecC 形式) 生成のためには、VSM の内部データをトラバースしてテキスト形式の目的コードを生成する処理を hand-coding されており、VSM のスキーマの変更や目的コードの記述形式（これは下流ツールの都合で度々変更がある）の変更に対して修正が発生した場合は品質を確保するためにに、コストの高いリグレッションテスト (過去の主要な出力例を全て通して目視確認する）を繰り返す必要があった

また DB ビルダー自体が hand-coding されたプログラムなので、不具合対応[1]の容易性や機能拡張性が低いという課題を抱えていた。例えばクラス定義に合成関係の再帰構造があると、これをうまく DB 用プログラムに展開できないという、モデル構造に依存するシステム運用上の制限があり、再帰構造をもつプログラム構造に DB ビルダーを変更すれば解決できることは判っていたが、上記課題により実装されることは無かった。

[1]幸い不具合は３年間無かったが

3 関連研究

ソフトウエア工学的手法の適用はDOA(Data Oriented Approach)を含め過去多数行われてきた。特に近年はMDA(Model Driven Architecture)[1]と呼ばれるモデル駆動の開発手法が着目されている。

MDAではPIM（Platform Independent Model)でプラットフォームに依存しないモデルを定義し、これから、プラットフォームの特有のモデルPSM（Platform Specific Model）を生成するという立場をとる。ここではPIMを定義するための技術がモデルでありモデルを定義するための仕組みがメタモデルという対応付けになる。モデル定義（メタモデル）を形式的に定義できれば、これからPSMである各種言語(C/C++,C#等)を生成することは機械的にできるであろうという立場をとる。

メタオブジェクトの定義から各種情報やプログラムを出力する技術としてはOMGのHUTN(Human-readable textual notation)やXMI(XML Metadata Interchange)に代表されるMOFの交換フォーマットに基づくものや、Anti-Yacc[6]などが先行研究としてある。

4 メタモデル技術の評価のアウトライン

本報告書では、メタモデルに基づく仕様の変換・合成技術を用いることにより、EDAに代表されるインダストリアルな適用対象において、どこまで自動化・高信頼化が図れるかを検証するために、MDA[1] のモデル変換技術を用いて、VSM 相当の設計データDB を構築してこれを評価する。

以下の３つの部位をターゲットとして評価を行った。

1. 設計情報のメタモデル (MMa) とコードのメタモデル (MMb) に基づく対象目的コードの生成部の自動生成
2. 設計情報のメタモデル (MMa) から設計データ DB の自動生成
3. 設計情報のメタモデル (MMa) から設計データ DB のデシリアライズ機能の自動生成

図 2 は評価部位が VSM 相当のシステムのどこに対応するかを示している。

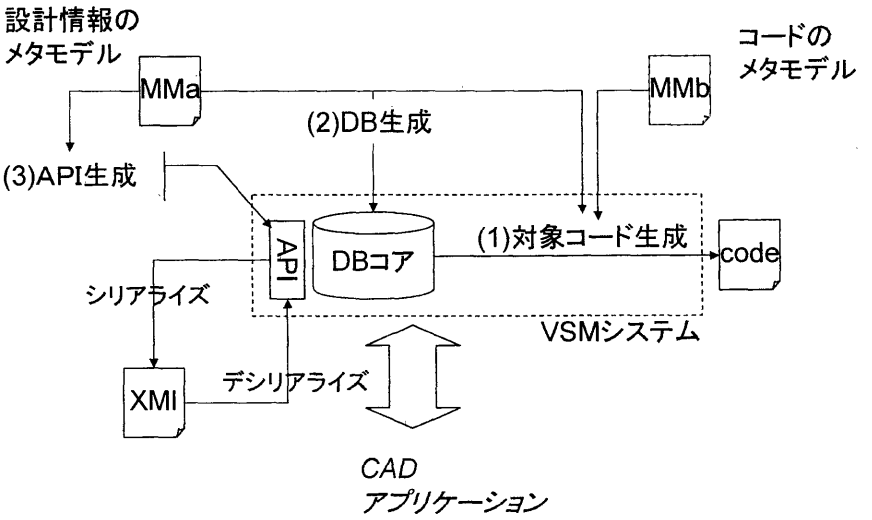

図 2: 評価の対象

5 モデル変換技術概要

オブジェクト指向の分野では、モデルを記述するためのモデルはメタオブジェクトと呼ばれ、代表的な MOF(Meta-Object Facility)[17] は OMG(Object Managing Group) によるメタオブジェクトのためのフレームワークを提供している。

OMG においては MDA は 4 つのメタレベルから構成される。MOF は M3 層の標準言語である。OMG では MOF の例として UML のためのメタモデルを提供している。

M3:メタメタモデル MOF。M2 モデルを記述するための言語の定義

M2:メタモデル クラス図を書くためのメタモデル。UML のメタモデルに相当

M1:モデル クラス図で表現されるクラス情報。UML で記述されたモデルに相当

M0:インスタンス 実世界すなわちインスタンスの世界。

UML 形式で記述されたクラス図は M1 レイヤーに対応する。M2 はクラス図を定義するための用語を定義するレイヤーである。クラス図をドメイン特化の言語や形式に変換するときのルールは M2 レベルで記述される。M0 は M1 のクラスに準拠するインスタンス情報である。

XMI は XML フォーマットで MOF を表現する OMG の標準で、M3〜1 の記述をサポートする。

また M1 を M:Model、M2 を MM:Meta-Model、M3 を MMM:Meta-Meta-Model と呼ぶことがある。

5.1 モデル変換 QVT

モデル変換とはインスタンスの間の変換で、例えば UML のクラス図で記述された対象ドメインのモデルをプログラミング言語 (例 C#) で記述されたクラス情報のモデルに変換することを行う。モデル変換には様々な技術がある [5]。

このモデル変換をメタモデル (MM) を使って組織的に行おうとする OMG の標準の一つが QVT(Queries/Views/Transformations) である。

図 3 は QVT におけるモデル変換の典型的なパターンである。MMM はメタメタモデルで, Tab は変換ルール。変換ルール自体のメタモデルが MMt である。Tab は MMa,MMb のメタモデルをつかって (basedOn) 記述されている。

この変換ルール Tab より変換プログラムが実行 (execute) され、モデル A(Ma) がモデル B(Mb) に変換される。

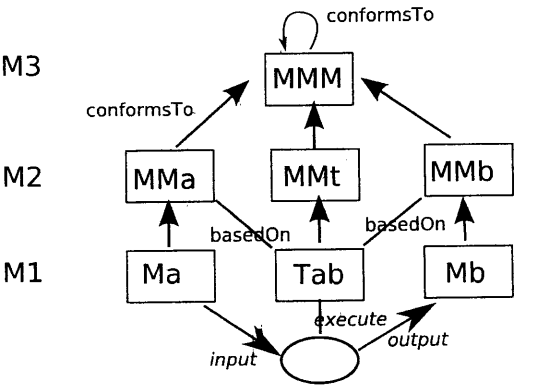

図 3: Model Transfer Pattern

メタモデルが同一のもの (MMa=MMb) の間の変換を endogeneous、そうでないもの (MMa ≠ MMb) を exogeneous と呼ぶ。

QVT に基づく代表的なモデル変換システムに ATL[15] や UMT[10] がある。

6 ATL の紹介

ATL(ATLAS Transformation Language)[14] は仏 INRIA にて開発されたモデル変換フレームワークで, OMG MOF/QVT RFP に対する一つの回答として開発されたものである。

ATL はモデル変換に対して、宣言的 (declarative) と imperative な変換処理記述を混在できることを特徴としており、基本的には宣言的に変換処理を記述し、どうしても記述できない処理に関しては imperative に記述するというスタイルを採用している。

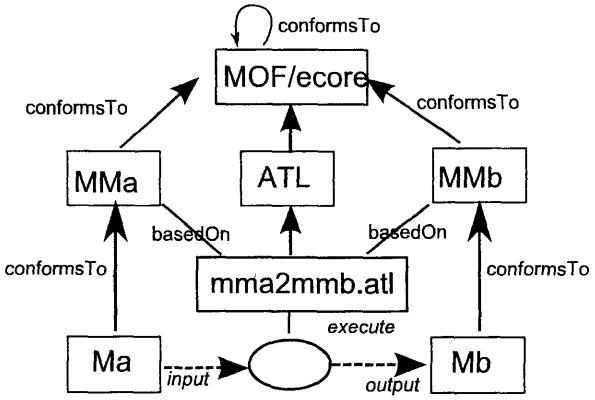

図 4: ATL 概要

図 4 に ATL の概要を示す。ATL では終端モデル a(Ma) を終端モデル b(Mb) に変換することを目的とし、そのために終端モデル a のメタモデル (MMa)、終端モデル b のメタモデル (MMb)、メタモデル間の変換ルールファイル (mma2mmb.atl) を入力として、モデル間の変換プログラムを生成する、ここでモデルとはインスタンス情報を指しており、メタモデルはクラス図に相当する。また ATL においてはメタメタモデル (MMM) として EMF プロジェクトで提供される ecore や OMG で提供される MOF(1.4) を指定することができる。メタレベルの階層に関してメタメタモデルを M3、メタモデルを M2、終端モデルを M1 と呼ぶことがある。

MOF においては M0 が実世界インスタンス、M1 がモデル（クラス図相当）、M2 がメタモデル (UML のモデル等)、M3 がメタメタモデルという分類がされる。ここで注意すべきなのは ATL における終端モデル (M1) は MOF におけるインスタンス (M0) に相当し、ATL におけるメタモデル (M2) は MOF における M1 と M2 が一緒になったものに対応し、ATL のメタメタモデル (M3) は MOF の M3 に相当していることである。これは MOF は UML を前提にしているので、M1 と M2 が未分化で M2 が UML の用語の定義 M1 が UML で定義されたクラス図なのにたいして、UML を前提としていない ATL では M2 も M1 もメタモデルである範疇では同じと見なしメタモデルを直接メタメタモデル (MOF や ECORE) を使って記述するからである。M3 は ATL も MOF も同じで、ここは自己記述性ということが条件になっているので紛れがない。

また eclipse のプラグインとして提供される ADT (ATL Development Tooling) 環境 [2] により、変換ルールの入力、変換プロジェクトの構成、ビルド、デバグを eclipse 環境によって統一的に行うことができる。

6.1 ATL 詳細

ATL では eclipse に ATL 特有の一連の機能に合わせて画面をカスタマイズする perspective が準備され、実行 configuration 画面にて入出力を指定する。

入力 (IN)	入力モデル Ma(例：Families のインスタンス) 入力メタモデル MMa(例 Families のメタモデル)
出力 (OUT)	出力モデル Mb 出力メタモデル MMB
変換	変換ルールファイル (.atl)

表 1: ATL システムの入出力

ATL のコアとなるものは変換ルールを定義するファイル (.atl) である。

変換ルールは入力メタモデル (MMa) と出力メタモデル (MMb) の間の関係をルールとして記述する。ルールは Header, Helper, Matched Rule, Called Rule で構成される。また宣言的なルール記述のほかに OCL(Object Constraint Language) に準拠した imperative な記述を可能にする特別な構文が用意されている。

6.1.1 Header

Header には 入力としてモデル (IN):メタモデル (Families)、出力のモデル (OUT): メタモデル (Persons) を定義している。ここで IN,OUT は configuration で指定されたモデルを指す。メタモデル名はメタモデル定義のトップスキームを指定する。

```
module Families2Persons;
create OUT : Persons from IN : Families;
```

6.1.2 Helper

Helper は Java のメソッドに相当し、変換処理の部品化の手段として用いる。context は helper が使用されるコンテキスト、def は helper 名、リターン値、ATL 表現 (ATL expression) で構成される。helper は引数をとることができる。下記の例は整数の Sequence 型 (リストのこと) と実数を引数にとり、リスト要素の平均が、引数で与えられた値よりも小さいかどうかを判断する関数として定義されている。

```
helper def : averageLowerThan(s : Sequence(Integer), value : Real)
                                                    : Boolean =
  let avg : Real = s->sum()/s->size() in
    avg < value;
```

6.1.3 Matched Rule

Matched Rule は入力 (from) と出力 (to) の間で関連を定義する。

ソース部 (from) は" 変換対象:ソースモデルのパターン (合致条件)"という形式をとり、ターゲット部 (to) は" 変換対象:ターゲットモデルのパターン (属性の初期化記述)"という形式をとる。

下記の例では from においては入力コンテキスト (Biblio!Journal) とマッチング条件 (not j.title.oclIsUndefined()) が合致する要素 (j) が選択され、to において変換対象 (b) が Biblio!Book であり、ソースの title 属性+vol 属性+num 属性がターゲットの title 属性になり、著者はソースの記事 (article) 集合の author 属性を集めた集合を割り当てている。このように OCL 関数 (collect,flatten,asSet()) を組み合わせて用いる。

```
rule Journal2Book {
  from
    j : Biblio!Journal(not j.title.oclIsUndefined())
  to
    b : Biblio!Book (
        title <- j.title + '_(' + j.vol + '):' + j.num,
        authors <- j.articles->collect(e | e.authors)->flatten()->
                                                    asSet()
        chapters <- j.articles,
        pagesNb <- j.articles->collect(e | e.pagesNb)->sum()
  )
}
```

6.1.4 Imperative な Rule

ルールには imperative なコードを記述することができる。imperative なコードは do という予約語で導入され、rule の to 部の記述による属性初期化の後に任意の手続き的なコードを記述できる。例えば後述する called rule の呼び出しも do 部のみにて記述できる。to 部における属性の初期化における expression 間はコンマ (,) にて区切られるのに対し、do 部の statement 間はセミコロン (;) で区切られる。これは do 部の statement は記述順に逐次的に実行されることを表している。

```
rule testCallRule(){
  from
   s:Vsm!Rule
  to
   t:AsmL!Rule
  do{
   -- 処理や、called rule の呼び出し
   t.rules <- thisModule.sampleCalledRules();
  }
}
```

6.1.5 Called Rule

called rule は from 部が無いことを除けば matched rule と同様である。また called rule は引数をとることができる。返り値は do 部の最後に評価された式である。

```
rule sampleCalledRule(){
  to
   t:AsmL!Rule
  do{
   t.location <- '/* sl2rule */';
   t;
  }
}
```

6.2 変換の例

ここではATL変換を簡単な例題で説明する。これはATLの開発チームによるユースケース「Faimlies2Persons例題」[9]より抜粋したものである。

変換概要

変換の概要を図5に示す。"Families2Persons.atl"が変換定義ファイルで、Familiesメタモデル(Families.ecore)、Personsメタモデル(Persons.ecore)をメタモデルとして変換仕様が記述される。ATLによる変換定義ファイルはコンパイルされて、モデル(インスタンス)間の変換が実現される。

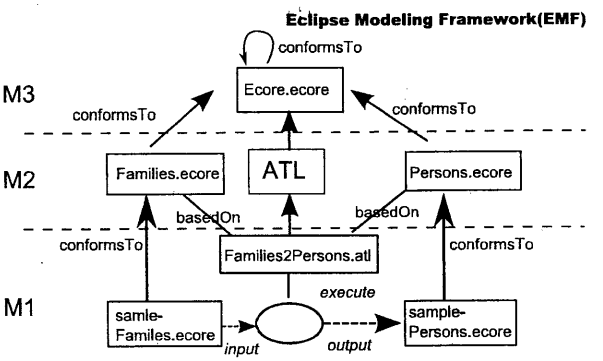

図5: ATL変換例概要

メタモデル

図6はFamilyのメタモデルを表している。Familyには lastName属性が、Memberには firstName属性がついておりそれぞれロール(father,mother,son,daughter)による合成関係でFamilyに従属する。

図7はPersonのメタモデルを表している。Personは fullName属性を持ち、サブクラスとして Male,Femaleをもつ。

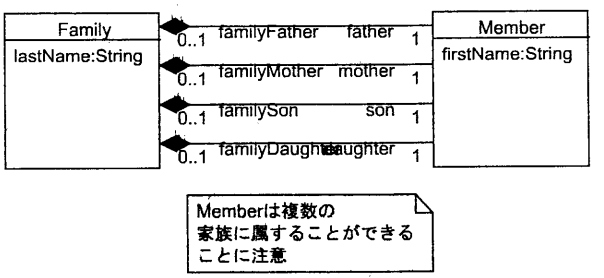

図 6: メタモデル (Families)

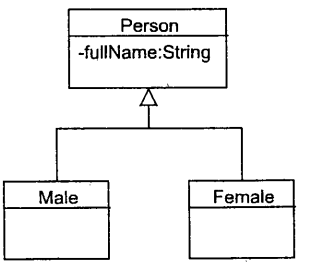

図 7: メタモデル (Persons)

ATL ファイル

helper である isFemale() は、ソースの Member に対して、それが女性かどうかを判断する関数 (返値 Boolean) を定義しており、これを用いて Member2Female,Member2Male のどちらが起動されるかが選択される。isFemale はメタモデル Families の Member 要素で有効な helper で Member が女性であるかどうかを判断する。ここで self は Member そのもの、oclIsUndefined() はモデルに該当する参照ポインタがないことを表す OCL 関数である。

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String =
--略--

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;

rule Member2Male {
  from
    s : Families!Member (not s.isFemale())
  to
    t : Persons!Male (
      fullName <- s.firstName + ' ' + s.familyName
    )
}

rule Member2Female {
  from
    s : Families!Member (s.isFemale())
  to
    t : Persons!Female (
      fullName <- s.firstName + ' ' + s.familyName
    )
}
```

変換例

図 6.2 に入力となるファイル"sample-Families.ecore" の例を示す。XMI 形式で記述されたインスタンス情報である。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns="Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <daughters firstName="Kelly"/>
  </Family>
</xmi:XMI>
```

図 8: 入力ファイルの内容

図 6.2 に出力ファイル"sample-Persons.ecore" を示す。こちらも XMI 形式で記述されたインスタンス情報である。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns="Persons">
  <Male fullName="Peter Sailor"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Jim March"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

図 9: 出力ファイルの内容

7 評価 1:ATL による目的コード生成の構築

ここでは ATL を用いて、VSM モデルから ASM(Abstract State Machine) の一実装である AsmL 形式の目的コードを生成する処理を構築する。ASM は形式的仕様記述言語の一種であり VDM[3] と同様に抽象度の高い仕様記述を行うことができる。

AsmL を選択したのは Ecore 形式のメタモデルが ATL のプロジェクトサイトより取得可能であることと、もともと ASM 言語の実行セマンティクスが形式的に定義されているので、動作イメージに対する間違いがないからである。MSC ツールの目的コードである SpecC 言語はコンパイラ・シミュレータを実装するベンダーにより動作セマンティクス異なっていた。

変換における課題

VSM は設計モデルを表しており、その大きな構造は、実行形式である AsmL と一対一対応している (例 HUnit が Class に対応する等)。しかし、VSM の状態内部の処理コードの生成に関しては、imperative な変換処理が要求される。特に VSM の構造から計算して生成される AsmL の条件判断ロジック生成の部分が、本質的にリスト構造をツリー構造に対応づけているので、単純に declarative な変換ルールで記述できる範囲を超えている。

7.1 AsmL の紹介

AsmL は Abstract State Machine の Microsoft 社による実装である。ASM 自体は Y.Gurevich が"evolving algebra"[11] という名前で導入したように、代数的な背景をもつ仕様記述言語であり、主に仕様の詳細化 (refinement) を意識して設計されたものであるが、本報告の中では VDM 相当の形式的な仕様記述言語とみなしてよい。ASM では step という最小単位で動作セマンティクスが形式的に定義されており、また並列処理を記述することができるので、hardware の動作セマンティクスや UML 等の仕様記述形式のセマンティクスを提供するものとして用いられる。

ここでは、AsmL で記述される仕様の例を、以下にしめす。

```
// Vsm Sample implemetation in AsmL V1.0
class  HUnit  //親クラスを定義します

class  HUnit1 extends HUnit    //スレッド1のクラス
 var a as Integer = 0
 var _state as String = "st1"  //state 変数
 Execute()                     //状態遷移を計算するメソッド
  if (_state = "st1") then
   WriteLine("hu1:st1")
   _state := "st2"
  else
    if (_state = "st2") then
     WriteLine("hu1:st2")
     _state := "st1"

class  HUnit2  extends HUnit    //スレッド2のクラス
 var b as Integer = 0
 var _state as String = "st21"  //state 変数
 Execute()                     //状態遷移を計算するメソッド
  if (_state = "st21") then
   WriteLine("hu2:st21")
   _state := "st21"

hu1 = new HUnit1()
hu2 = new HUnit2()

var cnt as Integer =0

Main()                         //main 関数
 step while cnt <10            //step 文を cnt<10 の間実行
  hu1.Execute()
  hu2.Execute()
  WriteLine("===>cnt" + cnt)
  cnt:=cnt+1 .
```

7.2 VSM メタモデル

図 10 に VSM のメタモデルの抜粋を示す。VSM は最上位構造である UnitComponent の下位構造に HUnit と呼ばれる動作モジュールが配置される構成をもつ。HUnit は並列動作するプロセスに相当する。HUnit は状態 (State) を下位構造にもって、内部で状態遷移を行う。状態 (State) はさらに、StateAction を下位構造に持ち、StateAction に、HUnit の外部通信路 (Chanel) との入力 (Input)、出力 (Output)、ポーリング (Condition) が下位構造として位置し、さらに任意の処理コード (Text) が配置される。

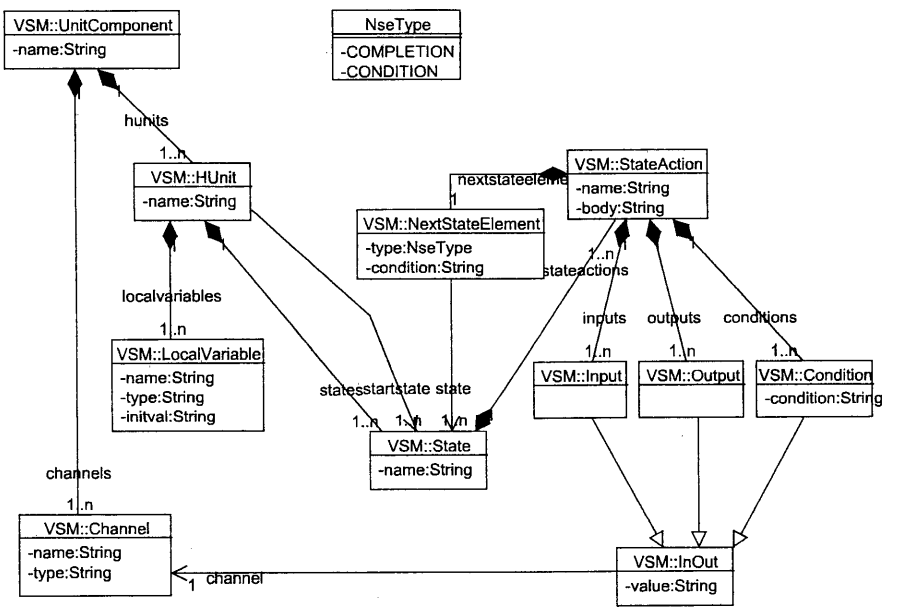

図 10: VSM のメタモデル (抜粋)

7.3 AsmL メタモデル

図 11 に AsmL のメタモデルを示す。AsmL のメタモデルは抽象構文木のモデルに相当する。

AsmlFile がルートモデルで、これに Main 関数と Main 以外のグローバルな定義が AsmLElement として所属する。AslmElement にはグローバル変数 (VarDeclaration) やクラス定義 (Class)、列挙型 (Enumeration)、構造体 (Structure) の定義が派生として挙げられる。処理の中心となるのは Body の下位構造である Rule であり、ここに step 単位での処理が記述される。

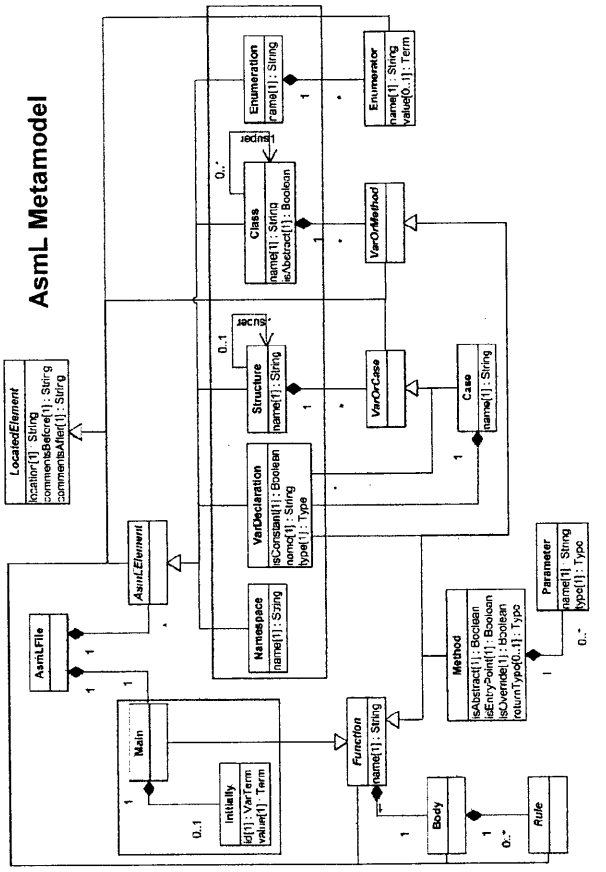

図 11: AsmL のメタモデル

図 12 に rule 部のメタモデルの詳細をしめす。

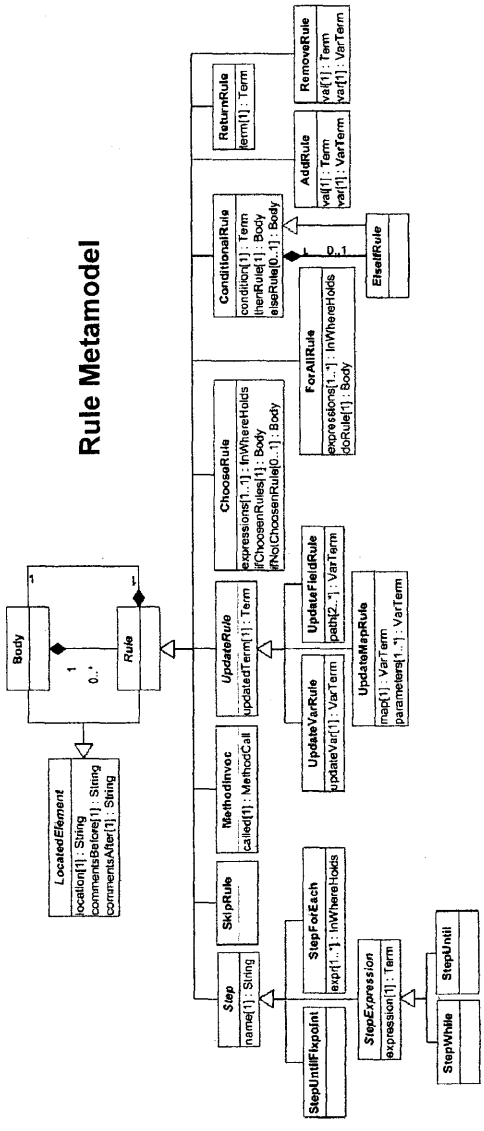

図 12: Rule 部のメタモデル詳細

7.4 単純変換ルール

一対一に宣言的にルールを記述できる部分を以下に表形式でまとめる

VSM 要素	→	AsmL 要素
RootModel	→	AsmLFile
HUnit	→	Class
		→名前は HUnit.name
		→ローカル変数定義
		→状態変数 (_state) 追加
		→ Execute メソッド (空) の追加
LocalVariable	→	VarDeclaration
		→名前は LocalVariable.name

以下に単純な変換ルールの例として LocalVariable の変換ルールを示す。Vsm!LocalVariable が AsmL!VarDeclaration に対応し、ターゲットの name 属性はそのままソースの name 属性に対応し、ターゲットの type 属性は、一端 AsmL!NamedType モデルを生成したものに対応づける。ATL の宣言的なルール記述機能をしか使っていないが、この例のようにソースの属性のリスト構造 (s.name, stype) をターゲットのモデルの木構造 (VarDeclaration) に変換が可能になっている。

```
rule LocalVariable2Var{
  from
    s:Vsm!LocalVariable
  to
    t:AsmL!VarDeclaration (
       name <-s.name,
       type  <-nt
      ),
    nt:AsmL!NamedType (
       name <-s.type
      )
}
```

図 13 は構造の違いをクラス図形式にて表したものである。VarDeclaration の下位構造に NamedType が集約配置されている。ATL の宣言的な変換ルール記述のみで属性のリ

ストが木構造に跨って展開できる理由はターゲットとなる木構造が構造的に一意にあらかじめ決まっているからである。

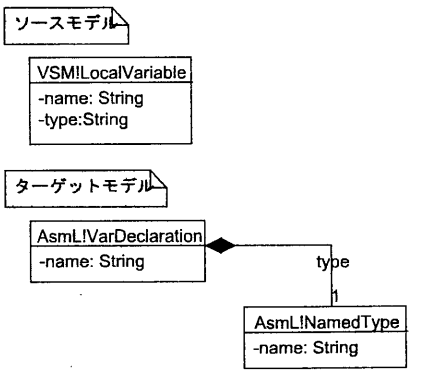

図 13: 構造変換の例

7.5 複雑変換ルール

HUnit 内部の変換に際しては以下の規則に従う

- HUnit は State をもつ、
- HUnit は startstate 参照をもつ
- State は StateAction(複数) をもつ
- StateAction は Input(複数),Output(複数)Condition(複数),Text(1 つ),NextSatateElement(複数) を持つ
- StateAction の実行動作は、
  1. 外部通信路からの入力 (Input)
  2. 条件判断 (条件=Condition)
  3. 処理本体の実行 (Text)
  4. 外部通信路への出力 (Output)
  5. 次状態の決定 (NextStateElement)

として展開される。

- State 中の StateAction には Condition を持たないものが少なくとも 1 つある。
- Condition がもつ条件は論理積で展開され、この条件が成立しないと一連の動作 (Text → Output → NextStateElement) を実行しない

対応する Asml のコード断片を以下に示す

```
Execute()
 if (_state = "st1") then  //State に対応
   let lreq = chan_req
   if (cnd1) then     //Condition 有りの StateAction に対応
    //body1           //Text に対応
    _state := "st2"  //NextStateElement に対応
   elseif (cnd2) then  // Condition 有りの StateAction に対応
    //body2
    _state := "st3"
   else     // Condition がない StateAction に対応
    //body3
    _state := "st4"
 elseif (_state = "st2") then //State に対応
     WriteLine("hu1:st2")
     _state := "st1"    //Condition 無しの StateAction しかない場合
```

この変換にはリスト構造から木構造への変換が 2 重の入れ子構造 (State の if then else 構造と StateAction の if then else 構造) になっている。

さらに前出の LocalVariable の例での宣言的なルールによる変換例ではリストの要素数が固定であったものが不定になっており、もはや宣言的なルールによる変換の範囲を超えている。

ここでは ATL の imperative なルール記述を利用することにする。また called rule は再帰的な定義を許しているので、ルールの do 部 (imperative コード記述ができる箇所) にリストを木構造に展開する called rule を再帰的に呼ぶことによりこれを実現する。

下記は State のリストを ConditionalRule の木構造に展開する自己再帰の called rule の例である。

```
rule StateList2CRuleBody(sl:Sequence(Vsm!State)){
  using{
      st:Vsm!State = sl.first();
      tail: Sequence(Vsm!State) = sl.excluding(st).asSequence();
    }
  to
    body:AsmL!Body(
      rules <- Set{t}
    ),
    t:AsmL!ConditionalRule (
       condition<-cndOpe,
       thenRule<- st
    ),
  -- 略 --
  do{
    if(sl.size()>1){
     t.elseRule <- thisModule.StateList2CRuleBody(tail); //再帰呼び
出し
    }
    body;
  }
}
```

この called rule は State のリストから図 14 のような AsmL の木構造を生成する。

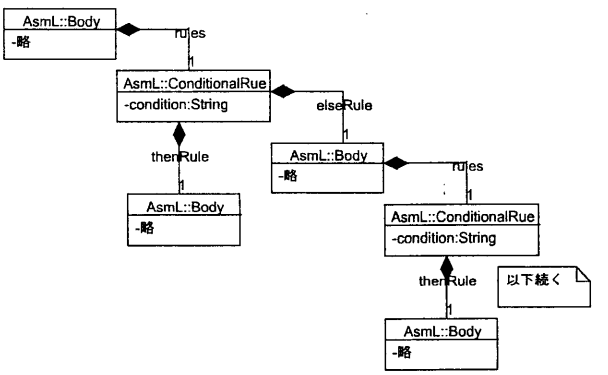

図 14: AsmL の木構造 (ConditionalRule)

8 評価 2:EMF による設計データ DB の構築

本節では、メタモデルに基づいて設計データベースを生成し、これを XMI 形式にて保存（シリアライズ）、読み込み（デシリアライズ）するシステムの例として、ATL と関連する eclipse プロジェクトである EMF(Eclipse Modeling Framework)[4] を説明し、EMF を用いて VSM データベースを構築する。

8.1 EMF の紹介

EMF ではメタモデルに基づく Java コードの自動生成を行うシステムとして開発された。メタモデルを実現するリポジトリ (データベース) としての Java コードを生成することと、XMI 形式に従ったシリアライズ・デシリアライズの機能を持つことが特徴とする。

EMF ではメタモデルを Ecore モデルと呼ぶ。OMF における MOF メタモデルに相当する。

EMF ではメタモデル情報から生成された EMF generator モデルをスタートにして、

- 設計データベースの基盤コード (Model Code) の生成
- 設計データベースを編集するための API コード (Edit Code) 生成
- 設計データベースの内容を入力編集するための木構造の簡易エディタのコード (Editor Code) 生成

を自動的に行うことができる。

EMF generator モデルは XML 形式で記述されたファイルであり、これを直接 XML として入力編集する以外に EMF においては、

- Annotated Java コードからの生成
- Rational Rose モデル (.mdl) からのインポート
- EMF のメタモデル形式 (ecore) からのインポート

. 等の機能をサポートしている。

図 15 に EMF の概要を示す。

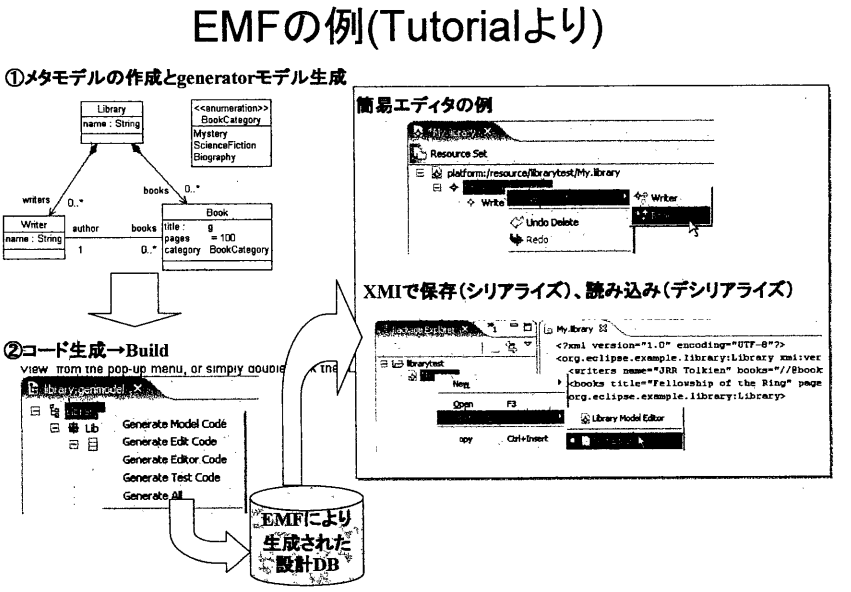

図 15: EMF の概要 (Tutorial より)

例えば図 16 のようなメタモデルがあれば、Mode Code として Book,Write のクラスおよび、Book には setAuthor(Writer)、getAother() メソッドが生成される。また Book,Write にたいする Factory クラスが生成される。

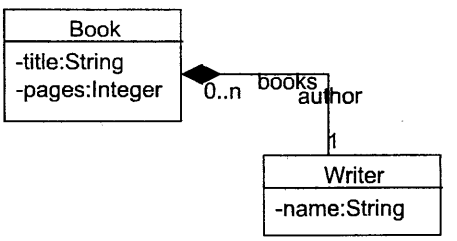

図 16: Book,Writer

生成された Model Code をコンパイルすれば、データベースが作成され、これを利用するためには所定の API を介して行う。以下にデータベース利用の例としてコード断片を以下に示す。factory に対して"create クラス名" のクラスメソッドにてインスタンスを生成し、インスタンスに属性を設定 ("setName" 等) し、参照関係に関しては book インスタンスに setAuthor(writer インスタンス) を行うことによりこれを行っている。

```
LibraryFactory factory = LibraryFactory.eINSTANCE;

Book book = factory.createBook();

Writer writer = factory.createWriter();
writer.setName("William Shakespeare");

book.setTitle("King Lear");
book.setAuthor(writer);
```

さらに、自動生成されたResourceSet機能を用いて、作成したインスタンスをリンク情報を保存したままxmiファイルに保存(シリアライズ)ができる。下のコード断片はシリアライズの例である。

```
// Get the URI of the model file.
URI fileURI = URI.createFileURI(new File("mylibrary.xmi")
                                                .getAbsolutePath());
// Create a resource for this file.
Resource resource = resourceSet.createResource(fileURI);

// Add the book and writer objects to the contents.
resource.getContents().add(book);
resource.getContents().add(writer);

// Save the contents of the resource to the file system.
try
{
  resource.save(Collections.EMPTY_MAP);
}
catch (IOException e) {}
```

逆にXMIからの読み込みも、同様にデシリアライズのためのAPIが用意されている。

8.2 VSM 設計データ DB の構築

節 7.2 で説明した VSM のメタモデル (図 10) を Ecore 入力として、Model Code 生成、Edit Code 生成、簡易エディタの Editor code 生成を順次行った。図 17 に得られた簡易エディタの画面を示す。次章以降の VSM のインスタンスデータは、この簡易エディタを利用して作成された。

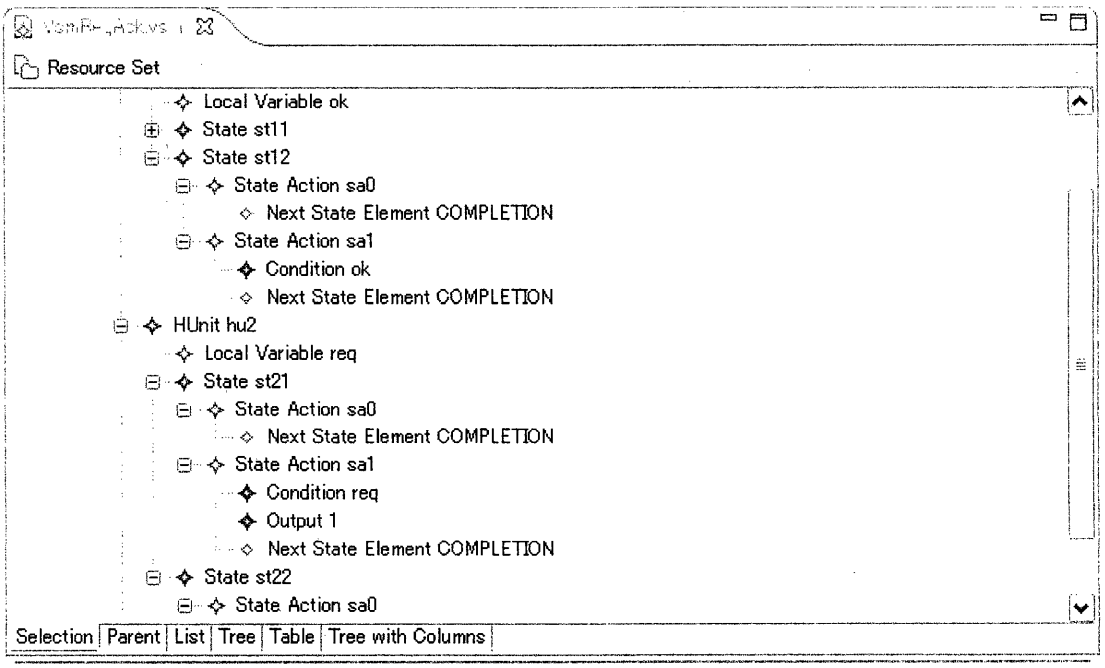

図 17: 生成された VSM 簡易エディタ

9 評価 3:ATL によるデシリアライズの生成の構築

9.1 デシリアライズ問題の説明

EMF 自体はメタモデルに基づいているといっても、コード生成を意図した一プログラムの実装に過ぎず、その点ではハンドコーディングされた設計データベース生成プログラムと同等の位置づけなので、メタモデルを使う真のメリットを享受していない。

メタモデルからのコード生成 (Model Code) 部は、比較的実装対象 (EMF の場合は Java) と構造が同等であることと、メタモデルの構成要素 (合成、参照、属性) 毎に適切な実装のパターンが定義できるので、ほぼ自明なコード生成であるといえるが、XMI ファイルとの間での（デ）シリアライズ処理は、メタモデルから imperative なコードを生成するという点で、自明ではない。

シリアライズは、もととなる設計データ DB の構造（メタモデル）が、Ecore メタモデル (MOF に相当) に従っていればオブジェクト構造に対する XMI の構造の対応は自明なので、Model Code 生成時に各クラス毎に定型のパターンのシリアライズのコードを生成すれば、全体-部品構造の最上位のインスタンスからシリアライズのメソッドを全体-部品構造に沿って再帰的に呼べば簡単に実現できる。これは VSM での最初の設計データ DB 構築の際に取った戦略である。

一方デシリアライズは、自明ではない、XMI ファイルの読み込みと Mode Code で提供される基本 API をつかったインスタンスの生成、参照の張り込みとが混在する imperative なコードを生成する必要がある。VSM のデータベースではメタモデルに基づいてデシリアライズを１つのメソッドとして生成する generate コードをハンドコーディングしており、これには以下の問題点があった。

- ハンドコーディングなのでメインテナンスが困難である
- 一メソッドで実現していたので、例えば再帰的な構造を持つモデル要素は取り扱えない

デシリアライズは、「モデル（インスタンス）をそれに対応する API 列」に変換すると捕らえればモデル変換の遡上にあげることができる。さらにこの変換方法自体は、もともとのメタモデル情報にのみ依存しているので、メタモデルからデシリアライズの全てが生成できるはずである。

9.2 ATL によるデシリアライズ変換の実現

ここでは、デシリアライズの問題を、XMI でかかれたモデルから設計データ DB をスクラッチから順に構築する API を呼ぶ順番を記述したテキストファイルを生成するモデル変換処理により実現する。

ここで API は第 8 章で説明した EMF が自動生成した Model Code の API とする。

ATL では、モデルをトラバースしておもにテキスト出力処理を行うための手段として query モードが用意されている。ここでは query モードを利用して、入力モデルをトラバースして API のテキストを生成することにした。

図 18 に ATL による変換の枠組みをしめす。メタモデルとして Ecore ベースで定義された VSM のメタモデルを利用し、VSM の（インスタンス）モデルを変換して、EMF で自動生成される API の列を生成する。

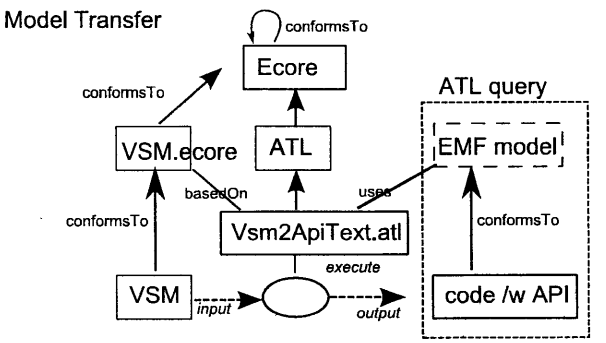

図 18: モデル変換によるデシリアライズ

query では、ソースモデルそれぞれに対して helper 関数として返値 String を持つ toString2() を定義する。また合成関係により全体部分関係があるものに関しては適切な場所で EMF が生成した API を使ってインスタンスの生成と、親オブジェクトへの登録を実現している。

```
helper context Vsm!UnitComponent def: toString2() :  String =
 let hunits : Sequence(Vsm!HUnit) = self.hunits in
  'UnitComponent uc = factory.CreateUnitComponent('+self.name+');\r\n'+
   if hunits->size() > 0 then
     hunits->iterate(e; acc : String = '' |
       acc  + e.toString2()+';\r\n'+
       'uc.addHUnit(hu_'+e.name+');\r\n'
     )
   else
     ''
   endif +
   '//endof code\r\n';
```

下記が出力結果である。所望の順番にて EMF が自動生成した API が呼ばれていることに注意。

```
UnitComponent uc = factory.CreateUnitComponent(uc1);
HUnit hu_hu1= factory.CreateHUnit(hu1);
State st_st11= factory.createState(st11); %st11 の作成
hu_hu1.addState(st11);  %親オブジェクト hu_hu1 への st11 の登録
State st_st12= factory.createState(st12);
hu_hu1.addState(st12);
hu_hu1.setStartState(st11);
uc.addHUnit(hu_hu1);
HUnit hu_hu2= factory.CreateHUnit(hu2);
State st_st21= factory.createState(st21);
hu_hu2.addState(st21);
State st_st22= factory.createState(st22);
hu_hu2.addState(st22);
hu_hu2.setStartState(st21);
uc.addHUnit(hu_hu2);
```

9.3 メタなモデル変換の導入

図 18 の変換では、メタモデルが変わる毎に ATL のルールを書き換えないといけないので、メタモデルのみをつかった自動化は実現されていない。そこでメタ度を 1 つ上げて、VSM のメタモデルを変換ソースモデルとし、Ecore をメタモデルとして上記ルール (Vsm2ApiText.atl) を生成するメタなモデル変換（モデル変換のルールをモデル変換で生成するという意味) を構築する。

Ecore は図 19 にあるようなモデルの構造をもっている。EClass がクラスに相当し、合成関係は eReferences 関係のなかで containment 属性が true のものであり、これを取得してルールを展開する。

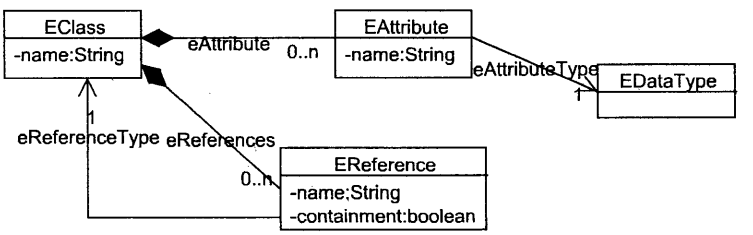

図 19: Ecore モデル (抜粋)

図 20 に ATL による変換の枠組みをしめす。メタモデルとして Ecore をそのまま使い、VSM のメタモデルを変換して、図 18 の変換ルール (Vsm2ApiText.atl) 相当を得る。

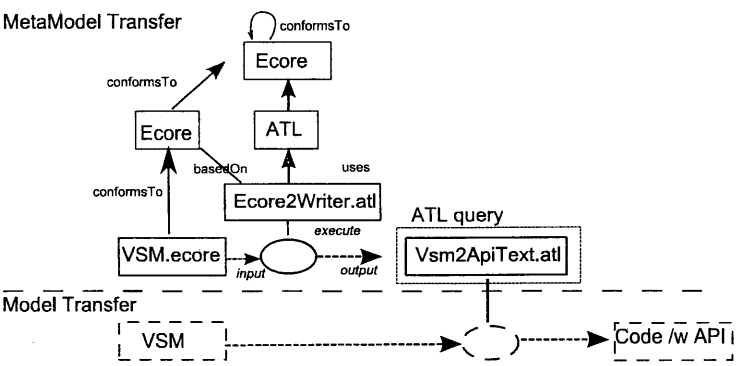

図 20: メタなモデル変換を組み合わせたデシリアライズ

```
query Ecore2Writer = ecore!EClass.allInstances()
               ->select(e|e.name='UnitComponent')
               ->first().toString2().writeTo('C:/test/example.txt');

helper context ecore!EClass def: toString2() :  String =
  let containments : Sequence(ecore!EClass) = self.eReferences
      ->select(e|e.containment=true) in
 'helper context = Vsm!'+self.name+' def:toString2(): String =\r\n'+
  if containments->size() > 0 then
     containments->iterate(e; acc:String = ''|
       acc +'  let '+e.name+': Sequence(Vsm!'+
       e.eReferenceType.name+') = self.'+e.name+' in;\r\n'
     )
  else
     ''
  endif+
   '  \''+self.name+' uc= factory.Create'+self.name+
   '(\' self.name + \');\\r\\n\'\r\n'+
   if containments->size() > 0 then
     containments->iterate(e; acc:String = ''|
       acc +'  if '+e.name+'->size() > 0 then\r\n'+
       '    --- do something with '+e.name+' \r\n'
     )
   else
     ''
   endif;
```

下記が上記ルールでの出力例である。

```
helper context = Vsm!UnitComponent def:toString2(): String =
 let hunits: Sequence(Vsm!HUnit) = self.hunits in;
 let channels: Sequence(Vsm!Channel) = self.channels in;
'UnitComponent uc= factory.CreateUnitComponent(' self.name + ');\r\n'
 if hunits->size() > 0 then
   --- do something with hunits
 if channels->size() > 0 then
   --- do something with channels
```

このようにモデル変換を用いることにより、メタモデルのみを入力とした、デシリアライズの仕組みを構築できることがわかった。

10 考察

10.1 ルールに基づくモデル変換のクラス

ルールに基づく、モデル変換においては、個々のルールは、ソースの部分構造をターゲットの部分構造に変換する仕様を表している。ここで部分構造は木構造をとる。6.2章で紹介した変換事例は、深さ1の木を深さNの木に展開するルールについての評価であった。6.2章の分析にもあるように、木の長さが固定であるか不定であるか、不定の場合も最大長があるか（有限不定）か最大長がわかっていないか（無限不定）の場合に分けることができる。

ソース木構造	ターゲット木構造	ルールの種別
固定長	固定長	宣言的な変換ルール
有限不定長	有限不定長	宣言的な変換ルール
無限不定長	無限不定長	宣言的な変換ルール+再帰ルール
その他	の組合せ	組合せ例があるかどうか不明

表2: モデル変換のクラス

10.2 メタなモデル変換が有効な対象について

メタなモデル変換（メタモデルをメタメタモデルを参照して変換する）が有効な対象は、モデル変換自体がメタモデル特有の情報に依存せず、メタメタ情報に変換ルールが依存する場合である。9章の事例はデシリアライズはメタメタモデルであるECoreに依存しており、個々のメタモデル(VSMのメタモデル等)には依存してない。

11 木言語 (Tree Language) 理論との関連

メタモデルに基づくモデル変換は、ATL の事例をみてもわかるように、中間ノードをメタモデルの情報で、リーフをインスタンス固有の情報で構成される木構造間の変換であるとみなすことができる。木構造を基盤とする言語変換の枠組みで理論的な研究がなされてきた。木構造間の変換を行う機能は Tree Transducer と呼ばれている。Tree Transducer は、入力木のノードのパターンに対して、出力木を生成する関数を呼ぶルールの集合で構成される。特に Macro Tree Transducer(以下 MTT と略す)[8, 7] と呼ばれるクラスは、入力パターンに引数をとることができることが特徴で、木を生成する再帰的な 1 階関数プログラムとみなすことができる。

また単項二階論理 (Monadic Second Order) 論理に基づく Transduce は、MTT を制限したものと同等であることが証明されており [7]、形式的な取り扱いが期待される枠組みである。本章では MTT を用いてモデル変換が構成できることを検証する。

Tree Translation の定義

Σ および Δ を rank 付きのアルファベット、T_Σ を Σ 上の木構造の集合であるとき、写像 $\tau : T_\Sigma \rightarrow T_\Delta$ を tree translation と呼ぶ。

Macro Tree Transducer(MTT) の定義

Macro Tree Transducer(MTT) は tree translation を実現する手段一つで、5 つの組 $M = (Q, \Sigma, \Delta, Q_0, R)$ にて構成され、各要素は以下の項目に従う

- Q は有限の**状態**の集合である。状態とは入力ソースのノードの状況のパターンに対応する。
- Σ は有限の rank 付入力記号、Δ は出力記号である。ここで rank とは木構造における直近の枝の数 (あるいは子ノードの数) に相当する。
- $Q_0 \subset Q$ は初期状態の集合
- R は有限のルールの集合で、以下の 2 つの形式をとる

  1. $q(a(x_1, x_2, ..., x_n), y_1, ..., y_k) \rightarrow t$
  2. $q(x_0, x_1, ..., x_k) \rightarrow t$

  ここで $q \in Q$ は k パラメータをもつ関数で、記号 $a \in \Sigma$ は rank 数=n

図 21 が 6.2 章で説明した例における Families に相当するインスタンス木構造である。属性情報、汎化情報、リスト構造はすべて木構造に置き換えて表現している。

以下MTTのルール $q(a(x_1, x_2, ..., x_n), y_1, ..., y_k) \rightarrow t$ を以下のtext形式で表現することとする。

```
<q,a>(y1,...,yk)  -> t(.. x1.. x2..)
%ここで xn は a の子ノードの n 番目をさすものとする
%<qid,a>は a がリーフ node であり唯一の子ノードであるインスタンス値を差す
```

MTTの例

ここでMacro Tree Transducerがモデル変換と同様の機能を持つことを、第6.2章で紹介した例題に準じて説明する。

最初にソースのモデルであるMarch Familyを木構造で表現する。ここでは家族の構成要因の数が不定であることを鑑み構成員(Member)をリスト構造にて構成していることに注意。

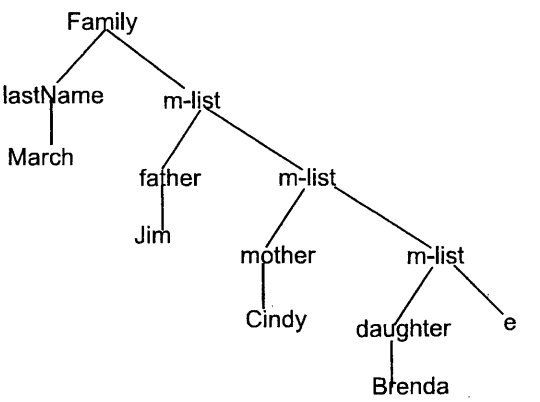

図21: 入力木(Families)

以下が、ソースモデルFamiliesをターゲットモデルPersonsに変換するMTTの例である。

```
<q0,Family>      -> <q,x2>(<q,x1>)
<q,m-list>(y)    -> o(<q,x1>(y),<q,x2>(y))
<q,father>(y)    -> Male(<qid,x1>,y)
<q,mother>(y)    -> Female(<qid,x1>,y)
<q,son>(y)       -> Male(<qid,x1>,y)
<q,daughter>(y)  -> Female(<qid,x1>,y)
<q,lastName>     -> (<qid,x1>)
<q,e>(y)         -> e
```

ソースノードが mlist であるときには二分木で展開され子ノードが mlist である場合は再帰的にルールが呼ばれる。また lastName は Family ノードの rank=1 の子ノードに格納されているのでこれを引数 (y) として他のルールを呼び、最後にリーフノードで、lastName をターゲット枝に展開している。

図 22 が変換後の Persons に相当する木構造を表している。

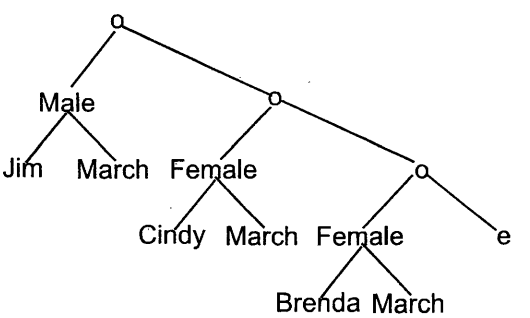

図 22: 出力木 (Persons)

11.1 ATL 事例との対応付け

第 7.4 章における VSM 事例での LocalVariable の変換は、MTT では例えば以下のように定義できる。

```
<q,LocalVariable> -> VarDeclaration(<qid,x1>, NamedType(<qid,x2>))
```

さらに第 7.5 章における Action のリスト構造から ConditonalRule の木構造への変換

は、MTT では例えば以下のように定義できる。

```
<q, sa_list>       -> ConditinalRule(<q,x1>,<q,x2>)
<q, StateAction>   -> 略%個々の StateAction の変換
<q, e>             -> e % リストの最後
```

このように、ATL で記述できる変換クラスが MTT でも同様に記述できる。

12 まとめと Future Work

メタデータに基づくモデル変換の枠組みである ATL にて、設計データ DB に関連する対象コード生成が可能であることを示した。リスト構造を木構造に変換する箇所が、宣言的なルールで書き切れない部分で、これは ATL の再帰呼び出しルールで記述が可能であった。

また設計 DB 自体の生成の形式化の試みとしてデシリアライズを、ATL の query モードを利用して行うことができた。メタモデルのみに基づいてデシリアライズを実現するには、メタモデル (MM) に対する ATL 変換ルールを、さらにメタメタモデル (MMM) に基づく ATL 変換ルールにて生成する組み合わせにより、これを実現できることがわかった。

以上のように、モデル変換技術がインダストリアルな対象にも適用可能であることが検証された。

EDA に限らず CAD 分野ではモデルの変換が、重要な機能要素なので、MDA 技術の変換ツールとしての適用だけでなくて、より本質的な研究が行われるべきで有ろう。

本質的な研究とは、モデル変換を意味の差異まで考慮した変換として実現することである。意味の違いを考慮したモデル変換は、過去人工知能のオントロジーとよばれる意味論の研究分野では、KEF(Knowledge Exchange Format) と呼ばれる異なる意味（セマンティクス）に基づくモデル間でのデータのやりとりを行うための交換フォーマとを策定したり、意味定義の共通基盤として Asm(L) や Descriptive Logic を用いる研究が行われており、モデル変換でも参考になると思われる。

またメタモデルを木構造と見なすと、モデル変換はメタモデル間の木構造の変換論理であると捕らえることができ、木オートマトン理論における Tree Transducer の枠組みで定式化できる。Tree Transducer における形式的な取り扱いとして二回述語論理 (Monadic Second Order) を用いることにより、形式的な検証や計算の複雑さに関する研究の成果を利用できる。例えば XML 同士の変換を MONA[12] ツールで実現した例がある [13, 18]。

参考文献

[1] *MDA* **モデル駆動アーキテクチャ**. エスアイビーアクセス, 2003.

[2] F. Allilaire and T. Idrissi. ADT: Eclipse Development Tools for ATL. *EWMDA-2, Kent, September*, 2004.

[3] D. Bjorner and C.B. Jones. *The Vienna Development Method: The Meta-Language.* Springer-Verlag London, UK, 1978.

[4] F. Budinsky. *Eclipse Modeling Framework.* Addison-Wesley Professional, 2003.

[5] K Czarnecki and S Helsen. Classification of Model Transformation Approaches. *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, 2003.

[6] Jim Steel David Hearnden, Kerry Raymond. Anti-Yacc: MOF-to-text. *Enterprise Distributed Object Computing Conference*, pages 200 – 211, 2002.

[7] J. Engelfriet and S. Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Information and Computation*, 154(1):34–91, 1999.

[8] J. Engelfriet and H. Vogler. Macro tree transducers. *J. COMP. SYST. SCI.*, 31(1):71–146, 1985.

[9] Fredric Jouault Freddy Allilaire. "families to persons" a simple illustration of modela model–toto–model transformationmodel transformation.

[10] R Gronmo and J Oldevik. An Empirical Study of the UML Model Transformation Tool (UMT). *NTEROP-ESA'05*, 2005.

[11] Y. GUREVICH. Evolving Algebra Lipari Guide. *Specification and Validation Methods*, page 936.

[12] J.G. Henriksen et al. *MONA: Monadic Second-Order Logic in Practice.* BRICS, Computer Science Department, University of Aarhus.

[13] Kazuhiro Inaba. "xml transformation language based on monadic second order logic". Master's thesis, Tokyo Univ., 2006.

[14] F. Jouault and I. Kurtev. Transforming Models with ATL. *Proceedings of the Model Transformations in Practice Workshop at MoDELS*, 3844:128–138, 2005.

[15] F Jouault and I Kurtev. On the Architectural Alignment of ATL and QVT. *Proceedings of ACM Symposium on Applied Computing (SAC 06)*, 2006.

[16] T.Ishii M.Iwamasa. System level specification synthesis. In *The 14th Workshop on Synthesis And System Integration of Mixed Information technologies(SASIMI)*, 2006.

[17] OMG. "Meta-Object Facility (MOF), version 1.4.

[18] Takeshi Yashiro. Typechecking k-pebble tree transducers: Practical efficiency. Master's thesis, Tokyo Univ., 2006.