モデル駆動型開発環境のプラグイン開発を容易にするモデル設計

木　村　功　作[†1]

開発環境のユーザビリティにとって，プラグイン開発を容易にすることは重要である．モデル駆動型開発環境では，モデルの設計次第でプラグイン開発においてモデル変換ルール等のモデルへの操作の記述が煩雑になってしまう恐れがある．本稿では，モデルの適合性を重視した設計とモデル操作の記述容易性を重視した設計を紹介し，モデルの設計とプラグイン開発の容易性の関係について述べる．

Model Design for Simplifying Plugin Development on Model-driven Development Environment

Kosaku Kimura[†1]

Simplifying plugin development is important for usability of development environment. Descriptions of model manipulations such as model transformation rule can involuntarily be complicated depending on model design. This paper describes relationship between model design and ease of plugin development contrasting two model designs: one prioritizes conformance of model, and the other prioritizes simplicity of model manipulations.

1. はじめに

開発環境のユーザビリティにとって，プラグイン開発を容易にすることは重要である．モデル駆動型開発環境は，Eclipse Modeling Framework (EMF)[⋆1]のようなモデリングフレームワークとそれを基盤としたモデル変換等のツールを利用して構築される．EMF は，Ecore という Meta Object Facility (MOF)[⋆2]準拠のメタモデルによって，モデルとそれをインスタンス化したオブジェクトの 2 階層モデルを定義する仕組みを提供する．しかし，Ecore でなく独自のメタモデルを開発環境で定義したい場合は，メタモデルを含む 3 階層モデルを，モデルとオブジェクトの 2 階層だけで設計するような工夫が必要となる．

本稿では，3 階層モデルの例として我々が EMF を基盤として開発を行っている Madras[1)] のデータフローモデルを用い，特徴の異なる 2 種類の設計を紹介することでモデルの設計とプラグイン開発の容易性の関係について述べる．

2. データフローモデル

Madras のデータフローモデルは，データと処理（プロセス）の依存関係からなるデータフローによって，アプリケーションの処理内容・手順を定義するために用いられる．データにはファイルやメッセージキュー，プロセスにはフィルタや集計のように様々な種類が存在する．プロセスは入出力ポート，プロパティで構成され，各々のプロセスの入出力ポートは各々異なる名前や多重度，結線可能なデータの種類を持つ．データ，プロセスおよび入出力ポートは各々独自のプロパティを持つ．データフローモデルには，以下の 3 つの階層が存在する．

M0　各種オブジェクト（データ，プロセス等）
M1　各々の種類のデータ，プロセスのモデル
M2　データ，プロセス等そのものを定義するメタモデル

3. データフローモデルの設計

本節ではデータフローモデルについて，モデルの適合性を重視した設計と，モデル操作の記述容易性を重視した設計の 2 種類のモデルを紹介する．

3.1 モデルの適合性を重視した設計

モデルの適合性を重視した設計におけるモデルの階層関係を図 1 (a) に示す．本設計では，Ecore をインスタンス化したモデル（Ecore モデル）としてメタモデル **M2** を定義し，その Ecore モデルのインスタンス化によりモデル **M1** を定義する．また，モデル **M1**

†1 株式会社 富士通研究所
Fujitsu Laboratories Limited

⋆1 http://eclipse.org/modeling/emf/
⋆2 http://www.omg.org/mof/

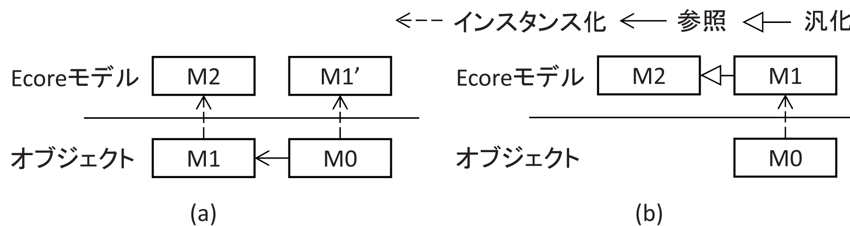

図 1 データフローモデルの階層関係

```
[template public generate(aProcess : Process) overrides generate
 ? (definition.name='AddTimestamp')]
insert into
 [getOutputPort()->select(definition.name='out')->first().data.name/]
 select
 [for (getInputPort()->select(definition.name='in')->first()
.data.dataTypeReference.oclAsType(Table).field) separator(',')]
 [name/]
[/for]
 , UDF.timestamp(
[property->select(definition.name='time')->first().value/],
[property->select(definition.name='millis')->first().value/]
) as
[property->select(definition.name='storedIn')->first().value/]
 from
[getInputPort()->select(definition.name='in')->first().data.name/]
[/template]
```

図 2 モデルの適合性を重視した設計におけるモデル変換ルールの例

を定義するオブジェクトへの参照を持つモデル **M1'** を Ecore モデルとして定義し，モデル **M1'** のインスタンス化によりオブジェクト **M0** を定義する．オブジェクト **M0** が持つモデル **M1** への参照により，オブジェクト **M0** がどのモデル **M1** に適合するかを表現する．

本設計の長所は，メタモデル **M2** の制約が Ecore モデルで定義できるため，モデル **M1** の適合性の検証が容易であることや，Ecore モデルとしてのメタモデル **M2** とモデル **M1'** が静的であるため，Java コード自動生成等を活用してモデル・オブジェクトの管理，エディタ等の開発環境の機能を容易に実装できることが挙げられる．一方で短所は，プラグインの開発者が Ecore とは異なる独自のメタモデル **M2** の中身を理解する必要があることや，図 2（Acceleo にて 2 つの文字列値からタイムスタンプを生成する処理を表すクエリへの変換ルールを記述した例）のように，モデル変換ルール等の記述が非常に煩雑になることが挙げられる．

3.2 モデル操作の記述容易性を重視した設計

モデル操作の記述容易性を重視した設計におけるモデルの階層関係を図 1 (b) に示す．本設計では，モデル **M1** とメタモデル **M2** を同時に Ecore モデルとして定義する．各々の種類のデータ，プロセスはサブクラスとして定義し，それらのモデル **M1** のインスタンス化によりオブジェクト **M0** を定義する．本設計を実現するために，データ，プロセス等の各構成要素には Ecore モデルの各要素を対応させる．例え

```
[template public generate(aProcess : AddTimestamp) overrides generate]
insert into [out.name/]
select
[for ("in".eClass().eAttributes) separator(',')]
 [name/]
[/for]
 , UDF.timestamp([time/],[millis/]) as [storedIn/] from ["in".name/];
[/template]
```

図 3 モデル操作の記述容易性を重視した設計におけるモデル変換ルールの例（対象は図 2 と同じ）

ば，プロパティ，入出力ポートは，それぞれ Ecore の `EAttribute`，`EReference` に対応させ，要素名に `in`，`out` 等のプリフィクスを付けることで入出力ポート等の要素の区別を行う．

本設計の長所は，図 3 のようにモデル変換ルール等を簡潔に記述でき，モデル **M1** を理解しやすいことが挙げられる．一方で短所は，メタモデル **M2** で制約を定義できず，モデル **M1** が制約を満たすか検証する必要があること，プラグインによって Ecore モデルとしてのモデル **M1** が動的に追加されるため，モデル・オブジェクトの管理やエディタ等，開発環境の全般的な機能実装が煩雑になることが挙げられる．

4. おわりに

本稿では，3 階層モデルである Madras のデータフローモデルの 2 種類の設計を紹介し，プラグイン開発容易性に関わる各々の長所，短所を示した．独自のメタモデル **M2** を定義する開発環境では，本稿のような設計の選択をする必要が出てくると考えられる．

我々は，プラグイン開発者にとっての使いやすさを考慮し後者の設計を採用した．これにより，我々のような開発環境の開発者の労力は増すが，代わりにプラグイン開発者の労力を減らすことができる．そのため，Madras では EMF に不慣れな開発者でも半日程度のレクチャでプラグイン開発を行うことができている．

参 考 文 献

1) Nomura, Y., Kimura, K., Kurihara, H., Yamamoto, R., Yamamoto, K. and Tokumoto, S.: Massive Event Data Analysis and Processing Service Development Environment Using DFD, *SERVICES 2012*, pp.80–87 (2012).