

振る舞い図(2) アクティビティ図等

㈱NTTデータ 技術開発本部
梅村 晃広 (mda-info-ml@rd.nttdata.co.jp)

1 はじめに

今回は前回に続いてUML2.0の動的構造を表すダイアグラム（振る舞い図）の概説を行ない、またそれに関連する概念としてアクションとOCLを説明する。

なお、本稿はUML1.Xについてのある程度の知識を前提としている。UML1.Xの詳細に興味がある読者は、本誌で過去に連載されたUML 紹介記事[1]等を参照していただきたい。

2 動的構造のモデル化

本稿で説明する項目を表1に示す。これらの項目は前回説明した相互作用図と合わせて、対象の振る舞い（動的構造）をモデル化するために使われる概念である（OCLについては静的制約の記述にも使われることがあるがこれは後述する）。

実際のシステム開発場面では、振る舞いのモデル化は次の2つのフェーズで行われることが多いようである。

(A) いわゆる上流段階でのシステム全体の大きな動きの規定

表1 本稿で説明する項目

項目名	概要	UML1.Xとの主な違い
ユースケース図 Usecase Diagram	アクターを中心としたシステムの使用シナリオを記述する図。	表記法の拡張。
状態機械図 State Machine Diagram	状態の遷移関係を表す図。組み込みシステム等の処理の細部記述に使われる他、業務システムの画面遷移の記述などにも使われる。	ステートチャート図 (State Chart Diagram) から改称。階層化機構等の追加。
アクティビティ図 Activity Diagram	アクションの順序関係を示す図。	階層化機構等の追加。アクション意味論の一部とりこみ。意味の厳密化。
アクション Action	処理の基本単位を表し、アクティビティ図のノードなどになる。	アクティビティの概念との間での再整理。
OCL(オブジェクト制約言語)	図的に記述しきれないモデルの細部の条件などを記述するための言語。振る舞いの規定に使うこともできる。	適用範囲を広げる拡張がされた。

(B) かなり下流段階での詳細な動作の規定

前者(A)の例としては、業務フローの記述や業務システムの画面遷移の記述が挙げられる。また後者(B)の例としては組み込みシステムの状態遷移図による記述などが挙げられる。

UML自体はその名前(Unified Modeling Language)の通り言語であって、各図をどの場面で使わなければならないかを規定はしていないが、実際には提供される各図の特徴によってどういった用途に使いやすいかという性質が生じてくる。

UML1.XからUML2.0への移行では、特に(B)の用途に適するよう

な追加/変更が推進されたことが特徴的である。

ところで、通常の業務プログラムでは、これまでは(B)の用途でUMLが利用されることは少なかった。これは、これまでの開発方法論として、(B)の段階の動作の規定は設計として行うことではなくコーディングとしてプログラミング言語で行うものだとされていたためである。

実際にはこの場合もプログラミング言語によって動作の規定が行われているのであって、結局は、動くシステムを作るためには動作の詳細が厳密に決定できる情報まで何らかの方法で与える必要があることには変わりない。

OMGのMDAの立場は上流から最下流に至るまで一貫してモデルによって記述すべきというものであり、この立場では必然的にUMLで(B)の情報まで記述することが求められることになる。

MDAについては本連載の第6回で説明する予定であるが、本稿で説明するUML2.0における振る舞い図まわりの変更点の多くが上記のような流れを反映したものであることは御留意いただきたい。

3 振る舞い図の変更の方向性

今回説明する振る舞い図の範囲でのUML2.0における変更は、前回の相互作用図と同様に以下の点で顕著である。

- ・構造化・階層化
- ・厳密化

このうち厳密化は、前述のようにMDAとの関係で特に重要になる。もちろんMDAに限らずとも、一般に意味が曖昧な道具を使ったコミュニケーションは行き違いのもとになりやすいため、厳密化は推奨されるべき方向である。

一方、構造化・階層化は、分割統治によってスケーラビリティの問題を解決するものである。これもMDAに求められるような詳細まで記述するほどスケーラビリティの問題が生じやすいから、という側面はあるが、MDAに限らず、UMLを大規模な開発に適用するためには重要である。

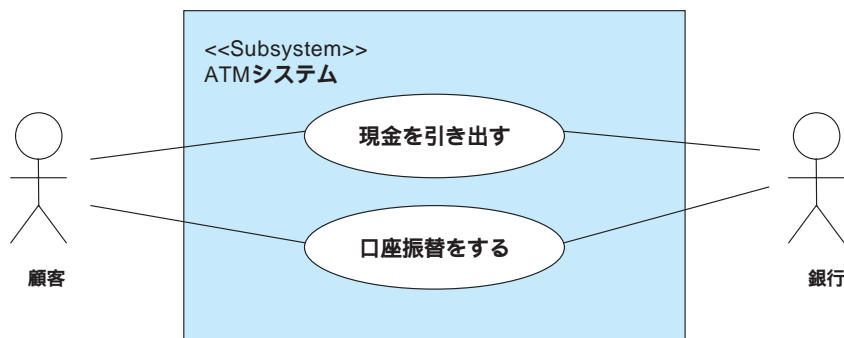


図1 ユースケース図の例

4 相互作用図との性質の違い

ここで、前回説明した相互作用図と、今回説明するアクティビティ図や状態機械図との性質の違いについていくらか言及しておきたい(なおユースケース図はこの議論に必ずしもあてはまらないので、ここでは除外する)。

前回説明した相互作用図は、オブジェクト指向の用語でいうインスタンスのレベルの様子を記述する図である。インスタンスとは個別例であり、インスタンスレベルの図はもともとの起源として「個別例から全体を推察せよ」という考え方によるものであるということが出来る。

これに対して、アクティビティ図や状態機械図はクラスや型のレベルの図であり、個別例ではなく全体として成立することを包括的に記述するタイプの図である。

この2つのタイプの記述方法はUMLに限らず日常の中でもものを説明するときに使われるものである。

一般には個別例を用いた説明の方

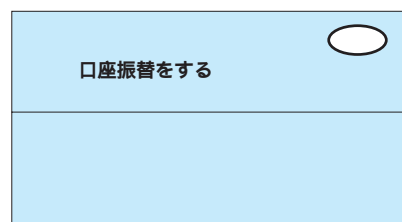


図2 ユースケースの別記法

がわかりやすい場合が多い。ただこの方法は、例として網羅しきれていない抜けの部分が生じやすい方法でもある(例えば本連載も、わかりやすさのために例を使ってUMLの説明をしているが、いくつも抜けているものがある)。

これに対してクラスや型のレベルの記述はインスタンスレベルの記法と比べるとわかりにくくなりがちではあるが、網羅性や厳密性を保ちやすい特徴がある。

このような性質から、MDAとの関係においては、アクティビティ図や状態機械図の利用が議論されることが多い。

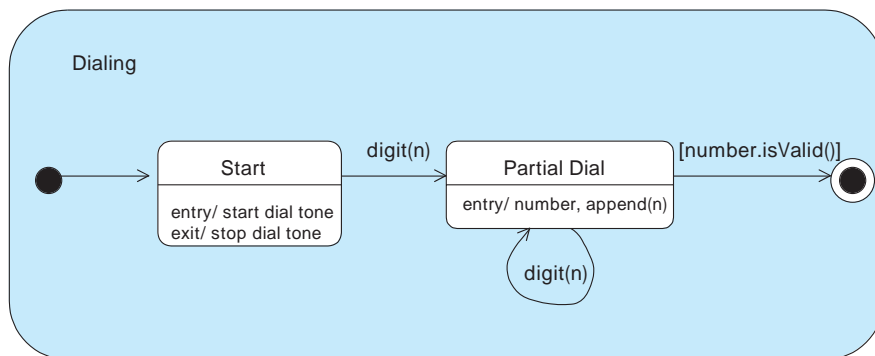


図3 合成状態の例 (状態機械図)

5 ユースケース図

ユースケース図は、システムの使用シナリオを、アクターを中心として記述する図である。

ほとんどの場合2節の(A)の用途で使われる図であるので、MDA対応の影響が少なく、そのせいか変更点は多くない。記法上いくつかの拡張がされたのが主な変更点である。

図1はユースケース図の例である。UML2.0では、この図の人型の部分に任意のアイコンを使ったり、楕円の部分を図2の記法で置き換えてたりできるようになっている。もちろん、従来の記法で描くこともできる。

なお、ユースケース図は実際の使用場面において「ユースケース記述」などの別のドキュメントと合わせて利用されることが多いが、ユースケース図以外のこれらの記法はUML2.0の一部ではない点に注意する。

6 状態機械図

状態機械図は、もともとUML以前からある状態遷移図の概念がUMLに取り入れられたものである。主要な部分は、ノードが状態を表し、エッジが遷移を表すグラフである。状態と遷移の集まりで、一つの状態機械となる。

状態機械図における階層化の方法の一つが合成状態 (composite state) である。合成状態とは1つの状態の中に部分状態機械が含まれているものである。図3はその例である。2つの部分状態および開始状態、終了状態とその間の遷移が、1つの合成状態内に定義されている。図4はオプションとして定義されている合成状態の内部を隠蔽する記法である (鉄アレイ状のアイコンが部分状態の存在を表している)。これらを利用すれば、適切なツールのサポートにより (例えば部分状態を見せたり隠したりする機構をサポートすることにより)、図的にもスケラブルな記述が可能になる。

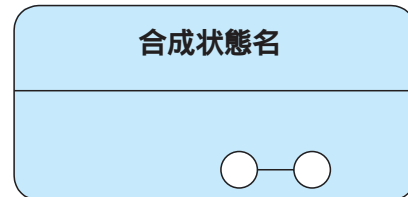


図4 合成状態の内部を隠蔽する記法

合成状態の概念はUML1.Xから存在したが、UML2.0では、部分状態機械への入場点、退場点の記述などの概念が追加され、より利用しやすくなった。

また、構造化の一種として、状態機械の継承 (拡張 (extension) という語を用いている) の概念が明確化された。これにより状態機械を差分で定義することが可能になった。

この他には、インタフェースの動的な仕様を状態機械によって定めるためのプロトコル状態機械 (protocol state machine) の概念が定義された点などが、UML2.0における新規の点である。

7 アクティビティ図

アクティビティ図 (文献[4]では「活動図」と訳されているが、ここではより広く使われているアクティビティ図という訳を採用する) はUML1.Xでも存在し、ビジネスフローの記述などに用いられてきた。

このような流れからフローチャートの並列拡張という見方がされる場合が多いと思われる。しかし、アクティビティ図では、制御フローだけ

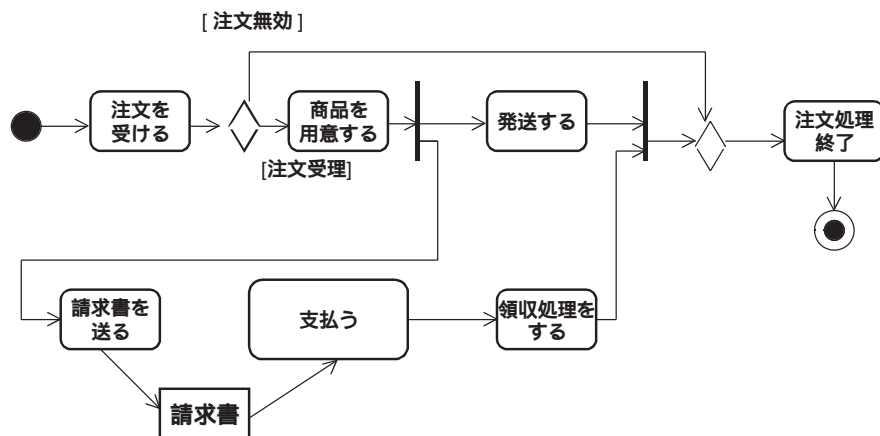


図5 アクティビティ図の例

でなくオブジェクトフローの記述なども行うことができる。これらを統一的に考えるためには、ペトリネットのような意味解釈（トークンが揃うと可能な選択肢のどれかが発火するというもの）をするのが妥当である。

実際、UML1.Xの段階では、アクティビティ図は状態図の一種とされていたが、UML2.0になってこのペトリネット的な独立した意味論が明

確化されたことにより、アクティビティ図による記述をより厳密に解釈することが可能となった。

構造化・階層化の方法には、一つには、アクティビティパーティションという概念によるものがある。これは図的にはスイムレーン (swimlane) として表される。UML1.Xでもスイムレーンは存在したが、UML2.0では多次元化したり、階層化したりすることができるよう

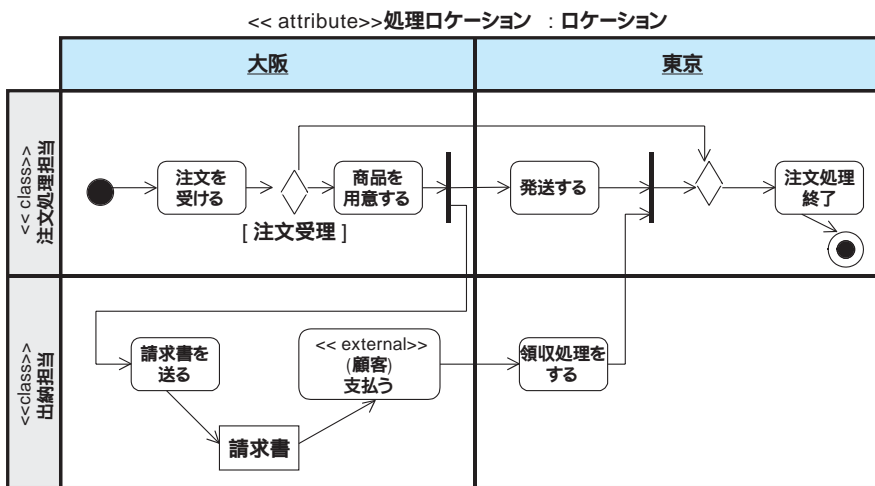


図6 多次元スイムレーンの例(アクティビティ図)



図7 下位のアクティビティを持つノード

になっている。図6は2次元のスイムレーンを用いた例である。

これ以外の階層化の方法としては、アクティビティ図内のノードの一つとして下位のアクティビティを表すノードを置く方法がある（これは、後述のアクションの一種である振る舞い呼び出しアクションを利用している）。下位のアクティビティを持つノードは図7のように記述される（フォーク状のアイコンが下位のアクティビティの存在を表している）。これを利用して、下位のアクティビティを別図で定義することなどによってスケーラブルな記述が可能となる。

UML2.0での変更点として、アクティビティ図内に描かれるノードの種類追加も挙げられる。

一つには、プログラミング言語レベルの詳細までアクティビティ図で記述するという観点から導入されたノードである。図8の例外ハンドラノード (exception handler node) はその例である。



図8 例外ハンドラの記法

この他に、意味論が厳密化されたことから区別する必要ができた概念を表すノードも存在する。データストアノード (data store node) がその例である。

8 アクション

ユースケース図、状態機械図、アクティビティ図は図の名称であったが、本節で説明するアクションについては、アクション図という図があるわけではない。

アクションは典型的にはアクティビティ図のノードとして現れ、処理の基本単位を表すものである。

アクションは、UML1.4の拡張として試験的にUMLに導入されたが、UML2.0を定めるにあたって、アクティビティとの間の関係が整理され、現在のような位置付けとなった。

アクションが定める処理の基本単位とは、例えば次のようなものである。

- ・変数/属性の読み/書き
- ・基本関数 (四則演算等)
- ・メソッド呼び出し
- ・他の振る舞いの呼び出し
- ・シグナルの送信



図9 シグナル送信を表すノード

このように、非常に基本的な処理を表すのがアクションである。しかしUMLでプログラミング言語相当の詳細度、厳密度でのモデリングをするためには、これらの概念が必要となる。

アクションの図的な表現については、現在のUML2.0仕様の中ではほとんど決められていない。シグナル送信アクションについては、図9のような特殊な記号で表すが、それ以外は、アクティビティ図のノードとして現れるときに、そのノードの印の中にアクション名を書くことぐらいしかできない。

しかし例えば、MDAツールなどではアクションを詳細に記述するための独自のテキスト記法などを導入しているものもある。

本稿はUML2.0の仕様を説明するのが目的であるのでこの点にはこれ以上立ち入らないが、アクションの概念はMDAとの関係において特に重要になる。

9 OCL

OCLも図の名称ではない。OCLはObject Constraint Language (オブジェクト制約言語) の略であり、UMLに制約を記述するための言語としてもともと定義されたものである。

制約とは、図的には記述しきれない条件を、UMLモデル上に式などによって付加したもののことである。図10はクラス図であるが、こ

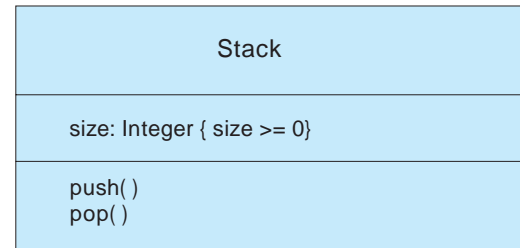


図10 制約記述の例

の中で属性 size につけられた {size >= 0} が制約である。

制約によって規定できるのはこのような静的な条件だけではない。例えば図10のpush()やpop()に制約をつけることにより、これらの操作の動的な振る舞いを規定することができる。すなわち、制約は振る舞いを規定する方法として利用することができるが、これは他の振る舞い図が果たす役割と共通点を持っている。

振る舞いについて、事前条件や事後条件といった制約によってその外部仕様を明確化し、設計者間の齟齬のない設計をすすめるという考え方は「契約にもとづく設計」(Design by Contract)として知られており、質の高いソフトウェア開発のための方法として注目されている(参考文献[7]参照)。このことは、特にMDAの考え方の中で重要となる。

UMLにおいて制約を記述するためには任意の言語を使用して良いとされているが、OCLはその可能な記述方法の一つとして定められたものである。

OCLはその名の通り一つの言語であり、UML全体からはかなり独立性がある。本連載の他の部分は仕

様書[2]によって仕様が定められているのに対して、OCLについては仕様書[3]によって仕様が定められている。

プログラミング言語論的な言い方をすると、OCLは純粋な関数型言語の一種であり、従って副作用的な動作(システムの状態を変える動作)は記述できない。しかし、副作用的な動作の結果どうなったかは記述することができ、制約記述には十分な能力を持っている。

例えば、図10のpush(), pop()について、事前条件・事後条件のいくつかをOCLで記述すると以下のようになる(UML図内にOCLの式で記述された制約を表現する方法は[2]や[3]で完全に規定されているわけではないので、ここではOCLの式だけを示す)。

push()の事後条件:

```
size= size@pre+1
```

pop()の事前条件:

```
size >=1
```

pop()の事後条件:

```
size= size@pre - 1
```

10 まとめ

今回は、相互作用図以外の振る舞い図と関連する概念について説明した。また、これらのUML2.0における変更点の多くが、2節の(B)に示した用途にもUMLを使えるようにするためのものであることを説明した。

なお、UML2.0の準拠レベルによっては、本稿に記述した記法や概念のすべてが含まれるとは限らない点に注意しておく。

UMLのMDA傾向については、その考え方自体に批判も多くある(例えば[6]参照)。しかし、開発の過程の中で、抽象度が違うだけで同じようなことを表す図やドキュメントを何度も作り直したり、ミスや変更によってその相互の対応がとれなくなったりすることを防ぐべきであることは明白であり、その一つの方法として最下流までの段階的詳細化過程を一貫した方法で記述可能とし、可能な範囲で適切な自動化と組み合わせることは妥当な解であろうと思われる。

また、MDA傾向とは別に、厳密さの増加ということはそれ自体評価できる。これまで図の持つ多義性を積極的に利用することがなかったとは限らない(例えば、曖昧な図で埋めた提案書を顧客に持って行き、顧客とのやりとりの中で都合の良いように解釈を付加していく、など)。しかし実際には図が表す意味が曖昧だったことによってその場では気付かないコミュニケーションミスが起こっており、後で問題が発生した、という方が多いのではないだろうか?

今回説明した項目の部分については、現在のUML2.0仕様案でもまだ検討の余地が見られる点がいくつか存在する。アクティビティやアクションについては、まだ実験的なところが残っている。特にアクションに

についてはまだ記法が整備されていない。また、プログラミング言語相当の記述をできるように拡張されたとはいえ、プログラミング言語論や計算モデル論の分野でこれまでに議論されてきた概念が、必ずしも妥当な形で取り入れられているとは言い難い面がある。厳密さの面でもまだまだ不満もある。さらに、OCLに至っては、[3]と[2]の2つの仕様書の間でかなりの不整合がある。これらの点は、今後、仕様の最終版になるにつれ、またはUML2.0以降のUML2.Xにおいて整備されていくと思われる。

今回は、本連載のこれまでとは趣向を変えて、より実例的な例題を用いてUML2.0によるモデル記述の説明をする予定である。

参考文献

- [1] 山本修一郎,UML の基礎と応用(連載記事),ビジネスコミュニケーション,Vol.38, No.9, 2001- Vol.40, No.3, 2003.
- [2] OMG, UML2.0 Superstructure Final Adopted specification, <http://www.omg.org/UML>, 2003.
- [3] OMG, UML2.0 OCL Specification, <http://www.omg.org/UML>, 2004.
- [4] OMG (OMG Japan SIG 翻訳委員会 UML 作業部会訳), UML 仕様書, アスキー, 2001.
- [5] UML2.0 入門(本連載) 第1回~第3回, ビジネスコミュニケーション, Vol.41, No.4-6, 2004.
- [6] Martin Fowler, Model Driven Architecture, (和訳: <http://capsctrl.que.jp/kdmsnr/wiki/bliki/?ModelDrivenArchitecture>).
- [7] Bertrand Meyer, Object-Oriented Software Construction(second edition), Prentice Hall, 1997.