修 士 論 文

# ユースケースモデルに基づく<br>動的モデル作成支援環境に関する研究

指導教官　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

山崎 英和

2001 年 2 月 15 日

**要 旨**

本研究では，ユースケースモデルに記述されるシナリオから得られたオブジェクト間のメッセージシーケンスを MSC で記述し，それを MSC → ObCL コンバータによって動的モデル ObTS/ObCL を作成する環境により，ユースケースモデルから動的モデルまでの一貫した開発方法を提案する．

**キーワード**: オブジェクト指向，ユースケース，MSC，ObTS，ObCL

# 目 次

# 第 1 章

# はじめに

ソフトウェアシステムが巨大化，あるいは複雑化している今日，システムを設計する際には様々なモデルを用いて設計を行なう手法が主流と成りつつある．

ユースケースモデルはシステムの要求機能を定義するものとして一般的に使用されている．ユースケースモデルでは，一つの完結した機能を表すためにユースケースを用いて記述する．また，ユースケースを補足するものとして，そのユースケースの機能を実現するイベントフローを自然言語によって記述したシナリオが与えられている．

また，オブジェクト指向開発において，動的モデルは各オブジェクトの振る舞い，つまりシステム内部の振る舞いを記述する重要なモデルである．動的モデルはユースケースモデルで定義された要求機能を満たすように状態遷移図を用いて作成される．

しかし，ユースケースモデルはシステムの開発段階で，ユースケースの追加，修正により何度も変更される．そのため，そのたびに開発者自身が動的モデルを変更する必要があり，効率的な開発が難しくなっている．しかも，状態遷移図になじみのない開発者にとってはなおさらである．

本研究では，動的モデルの作成を計算機により支援する環境を構築し．さらに，この環境を用いたユースケースモデルから動的モデルまでの一貫した開発方法を提案することを目的としている．

状態遷移図はオブジェクト間の相互作用，つまり，メッセージ通信によってオブジェクトが取りうる状態を記述したものである．そのため，ユースケースのシナリオから，比較的抽出しやすいオブジェクトとその間のメッセージシーケンスを抽出してやれば，状態遷移図を自動的に生成することが可能ではないかと考えた．

計算機を用いて動的モデルを作成するためには，抽出したメッセージシーケンスを形式

的に記述する必要がある．そのため，本研究では通信分野で広く利用されているメッセージシーケンス図 (MSC)[1] を用いて記述することにした．また，動的モデルとして再利用性や可読性に優れた ObTS[7]/ObCL[9] を用いることにし，MSC から ObCL に変換する環境を構築した．さらに，この環境を用いたユースケースモデルから動的モデルまでの一貫した開発方法を提案し，最後に，動的モデルを作成する際に，この方法が有効なことを示す．

この研究により開発者はユースケースモデルから動的モデルを一貫して作成することが可能となり，システムの分析/設計に集中することができる．

以下に本論文の構成を示す．第 2 章では本研究の背景とその目的について述べる．第 3 章では本研究の基礎となる MSC について述べる．第 4 章では動的モデル作成に必要な MSC を再定義し，動的モデル作成支援環境を構築した．また，この環境を用いたユースケースモデルから動的モデルを作成する方法を提案した．第 5 章では第 4 章で提案した動的モデル作成方法を用いて ATM システムを記述することでその有効性を確認した． 第 6，7 章でまとめ及び今後の検討を述べる．付録 A には本研究で再定義した MSC のテキスト形式文法を示し，付録 B には ATM システムの例で作成した文書を示した．

# 第 2 章

# 背景と目的

この章では，本研究に関連する事柄，および本研究の目的について述べる．

## 2.1 オブジェクト指向開発方法論

一般的にソフトウェアシステム開発は複雑な仕事であり，正確に動作する信頼性のあるシステムを開発するためには，いくつかの異なる側面を考慮しなければならない．しかし，その複雑性のためすべての要求を同時に取り扱うことは困難である．オブジェクト指向開発方法論では，システム開発でいくつかの異なるモデルを用いることによりこの複雑さを扱う方法が提案されてきた．

システム開発は，システムへの要求を定義した要求モデルを書くことから始まる．要求モデルの 1 つで，広く利用されているものにユースケースモデル (use-case model) がある．ユースケースモデルは Ivar Jacobson が提案したもので [4]，システムに要求されるすべての機能を定義するモデルである．

このユースケースモデルを用いればシステム開発の分析，設計，実装，そしてテストの各段階で，様々なモデルを作成するのに役立つ．これは,「ユースケース主導開発」と呼ばれている．この方法論では，ユースケースモデルは他のすべてのモデルの作成を制御する．つまり，ユースケースによって定義された機能は，論理的で頑強な，けれども実装環境に依存しない分析モデルに構造化される．次にこのモデルを実際の実装環境に適合した設計モデルに変換する．そして，ユースケースは実装モデルでソースコードによって実装される．最後に，ユースケースに定義された機能をシステムが提供するかどうかをテストモデルによりテストする．

## 2.2 ユースケースモデル

ユースケースモデルはアクター (actor)，ユースケース (use-case)，システム境界 (system border) によって記述される．アクターにはシステムと相互作用する人あるいはものを記述し，ユースケースにはアクターのインスタンス（ユーザなど）がシステムとの対話における動作に関連する一連のイベントフローを記述する．また，ユースケースを補足するものとして，イベントフローを自然言語で記述したシナリオが使用される．

ユースケースの主要な目的は以下の通りである．

- システムの要求機能の決定と記述
  このモデルは理解しやすく，ユーザの観点から記述されているため，顧客（あるいは，エンド・ユーザ）との対話により顧客の要求を見つけ出すことができる
- 要求機能から他のモデルへの追跡可能性の提供
  ユースケースモデルの修正とともに，システムの分析/設計や実装に影響を及ぼすユースケースの追跡によって，他のモデルの変更箇所を見つけ出すことができる
- システムをテストするときの基礎として
  開発されたシステムは実際，ユースケースモデルの要求機能を満たしているかどうかというテスト項目とテスト計画の作成に使用される
- ユーザガイドやプロジェクトのスケジュール作成の基礎として

図 2.1 はユースケースモデルの例である．箱はシステム境界を表し，アクターは箱の外部の人によって表現される．ユースケースは箱の内部の楕円で表現される．

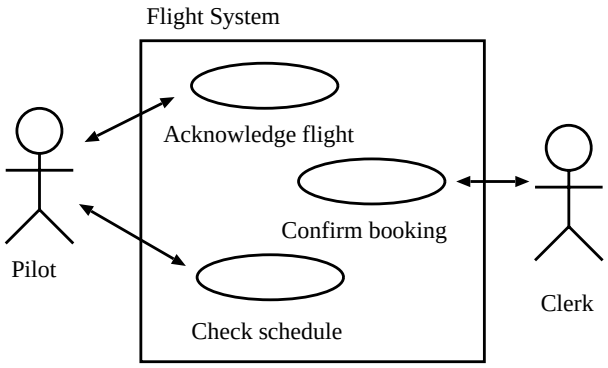



図 2.1: ユースケースモデル

## 2.3 メッセージシーケンス図 (MSC)

ユースケースのシナリオからオブジェクト間の相互作用をモデル化するには，一般にシーケンス図 (sequence diagram) が使われている．シーケンス図の焦点は，メッセージシーケンスであり，メッセージが多くのオブジェクト間でどのように受送信されるかということである．一方，通信分野では通信システム内の構成要素間のメッセージシーケンスを形式的に記述する標準的な記法としてメッセージシーケンス図 (Message Sequence Chart) が広く使われている．

ITU 標準のメッセージシーケンス図（以降，MSC と呼ぶ）は，通信システム内の実行トレースを視覚化するために通信分野で広く利用されている．MSC は，通信を行なうシステムの構成要素（実体）や外部環境の間でメッセージをやり取りすることに重きを置いた特殊なトレース言語として見なすことができる．MSC は，視覚的なレイアウトを持っているため，記述されたシステムの振る舞いを直感的に理解することを可能としている．

MSC は，ITU(以前の CCITT) 研究グループが行なっている勧告の中で長い間使われており，また，産業界においても，異なる使い方や様々な名前で非公式に使われてきた．そのため，これらの MSC を標準化するための活動が ITU によって行なわれている．

最初の MSC の勧告 Z.120(MSC'92) は 1992 年にジュネーブでの ITU 会議で承認され，新しく改訂された勧告 Z.120(MSC'96)[1] は 1996 年 4 月に，最終研究期間の閉会会議で承認されている．この標準化により，体系的なツールのサポート，異なるツール間での MSC の交換，SDL[1]仕様へまたは SDL 仕様からのマッピング，そして ITU 内での使用法の統一が可能となった．

MSC は以下の要件に使うことができる．

- いくつかの実体によって提供されるサービスの概要として
- システムの要求仕様を定義するものとして
- SDL 仕様の詳細化の基礎として
- システムシミュレーションと妥当性の基礎として
- テストケースの選択と仕様化の基礎として
- 通信処理の仕様として

[1]Specification and Description Language（勧告 Z.100, ITU 1996）

- インタフェースの仕様として
- オブジェクト指向分析/設計におけるユースケースの形式化として

MSC は，グラフィカルな構文とテキスト形式の構文を持っており，これに対応する形式的なシンタクスと非形式的なセマンティックス記述が与えられている．現在，Z.120(MSC'96) では，MSC 言語は Basic MSC と構造概念 (structural concepts) から構成されている．

Basic MSC は，インスタンス (instance)，メッセージ (message)，外部環境/ゲート (environment and gates)，一般的な順序 (general ordering)，状態 (condition)，タイマー (timer)，アクション (action)，そしてインスタンス生成と終了 (instance creation/stop) からなる．

一方，構造概念は，coregion，インスタンス合成/分解 (instance composition/decomposition)，インライン表現 (inline expression)，MSC 参照 (MSC reference)，そして High-level MSC(HMSC) からなり，MSC を合成したり異なるレベルの MSC（の一部分）を再利用することを可能としている．

## 2.4 動的モデルと状態遷移図

すべてのシステムには静的構造と動的振る舞いがある．動的モデル (dynamic model) は James Rumbaugh らが提案したモデル [5] で，分析/設計プロセスにおいてシステム内の動的振る舞いを記述するモデルである．このモデルは，システム内部の動きを視覚化しているため，多くのシステム開発の現場で使われている．

動的モデルでは，時間を通じてオブジェクトがメッセージをやり取りし，それに伴って状態が変化する様をイベント (events) と状態 (states) を用いて記述する．あるクラスのイベント，状態，そして状態遷移は抽象化され，状態遷移図として表現できる．動的モデルは複数の状態遷移図から構成される．重要な振る舞いを持つクラス 1 つにつき 1 つの状態遷移図が割り当てることによりシステム全体の活動パターンを示す．各状態機械は平行に動作し，状態を独立に変化させることができる．様々なクラスの状態遷移図は，共有イベントを介して 1 つの動的モデルに統合される．

動的モデルを記述するために広く利用されているものに，David Harel が提案した Statecharts がある．Statecharts は状態遷移図に階層構造/並行性/ブロードキャスト通信を持たせて拡張した記述モデルである．原始的な状態遷移図では状態を平面的に展開するために，記述対象のシステムの規模が複雑になるにつれて状態数や遷移数が爆発的に増加する傾向があるが，Statecharts ではこれを改善するために階層構造によってこれを制御して

いる．また，1つの状態遷移図の中に平行に動作する状態を記述できるよう，平行動作の表現を追加している．さらにイベント通信にブロードキャスト通信を採用している．

しかし，Statecharts ではデータをグローバル変数でしか表現できないため，大規模なシステムを記述する場合には，データに関する可読性が低下してしまう．また，状態の階層化はオブジェクトモデルの構造との対応が取られていないため，どのオブジェクトがどの状態を，あるいは状態遷移とそれに伴う動作を持つのかがが明確ではないという欠点もある．

## 2.5 ObTS/ObCL/ObML

### ObTS

伊藤が提案した ObTS[7] は Statecharts の計算モデルに基づく記述モデルである．ObTS は記述対象のシステム構造をオブジェクトの階層構造で表現し，システムの動作を状態遷移，関数的な属性計算，属性つきイベントのブロードキャスト通信を用いてモデル化している．ObTS では，オブジェクトは属性と状態遷移図を持っている．オブジェクトは動作の委譲のために内部オブジェクトを持つことができる．内部オブジェクトも同様に属性と状態遷移図を持ち，階層的に定義できる．オブジェクトが最初に実行する状態遷移を初期遷移としている．

オブジェクトは入力イベントを受け取って条件をチェックし，その条件が成り立てば状態遷移を行ない，属性を評価して出力イベントを出す．これを ObTS では，遷移を表す矢印に，“入力イベント [条件式]/出力イベント [属性計算]” という形で遷移規則を付加することで記述する．

図 2.2 は ObTS のグラフィカルな例である．ここでは，わかりやすいように本来状態遷移に付加される遷移規則の入出力イベントや条件/属性計算を省略している．角丸の四角はオブジェクトを，円は状態を，矢印は状態遷移を，四角はオブジェクト属性を，そして黒丸からの矢印は初期遷移を表している．この図から，オブジェクト “X” は属性 “$a_1$”，“$a_2$”，状態 “$s_1$”，“$s_2$”，内部オブジェクト “Y” からなっており，オブジェクト “Y” は属性 “$a_3$”，“$a_4$”，状態 “$s_3$”，“$s_4$” から構成されていることがわかる．

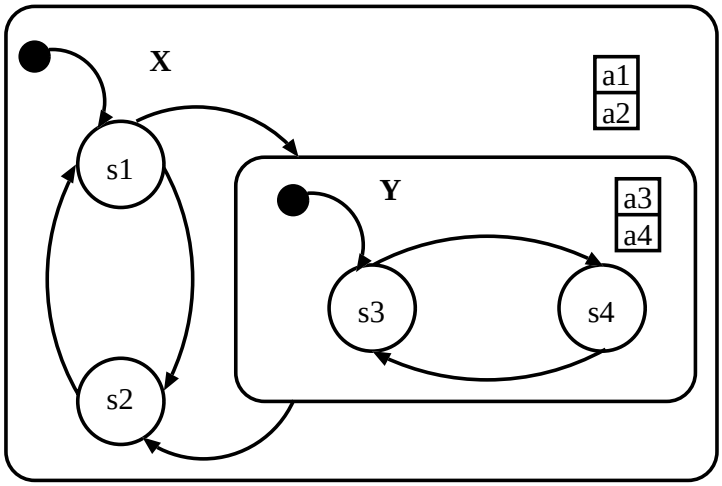



図 2.2: ObTS モデルのグラフィカルな表現

図 2.3 は，図 2.2 の内部オブジェクト “Y” の状態遷移に遷移規則のラベルを付加したものである．この例では，オブジェクト “Y” が状態 “$s_3$” であるときにイベント “$e_1$” を受取り，条件式が成り立てば，状態 “$s_4$” への遷移が起こる．このとき属性 “$a_3$” の値が 1 増やされる．また，オブジェクト “Y” が状態 “$s_4$” であるときにイベント “$e_2$” を受け取れば状態 “$s_3$” への遷移が起り，このときイベント “$e_3$” が出力され，属性 “$a_4$” の値が 0 になる．

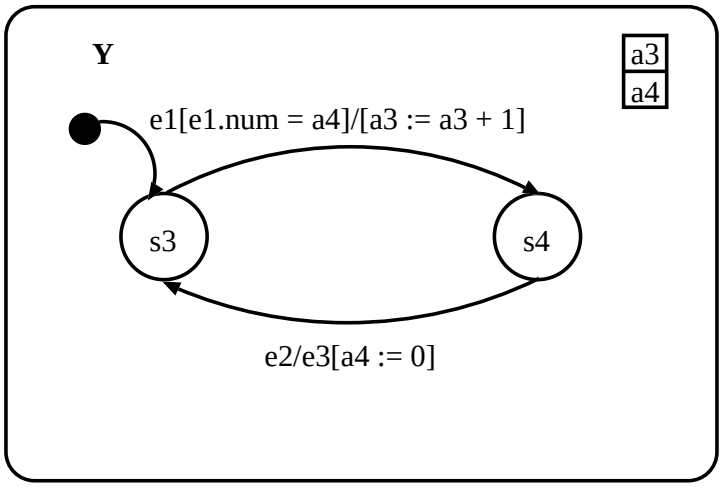



図 2.3: オブジェクトの動作

### ObCL

久保秋は，ObTS モデルに基づく記述体系として仕様記述言語 ObCL[9] を考案した．ObCL ではオブジェクトクラス，イベントクラス，フィールドクラス，属性クラスの集まりと一つのシステム記述によりシステムが表現される．

オブジェクトクラスはオブジェクトの記述とその再利用のためのものであり，オブジェクトクラス名，フィールド参照，属性，操作，状態，内部オブジェクト宣言，状態遷移規則により構成される．さらに，状態遷移規則は，遷移元，入力イベント，遷移条件，動作，遷移先，出力イベントによって指定する．イベントクラスはイベントの記述と再利用のためのものである．クラス名と属性の他に属性を隠蔽するための操作を持つ．ObCL ではイベントの通信範囲を限定するためにフィールドという概念が導入されており，フィールドの記述/再利用のためにフィールドクラスがある．ここでは，そのクラスのインスタンス（つまりフィールド）に流れるイベントを記述する．これに関連して，イベントはフィールドに流れるのでフィールド名とイベント名の対によって “f.e3” のように書かれる．また，同様にイベント属性はフィールド名とイベント名と属性名を連結して，“f.e3.val” のように書かれる．システム記述は対象システムのトップレベルに現われるオブジェクトを指定するためのものである．図 2.4 に，図 2.3 のオブジェクト “Y” の ObCL 記述例を示す．

### ObML

ObCL で記述したものを Standard ML 上で実行/テストすることができる．実際には Standard ML 上に ObTS のシミュレータ ObML があり，ObCL コードを ObCL コンバーターにより ML コードに変換して実行する．ObML では ObTS の様々な概念が ML の関数や型で表されており，シミュレータの実行結果も ML のデータとして返るので，ユーザが自分で ML 関数を作って自由度の高い実行やテストを行なうことができる．

```
Class Y
  field f:F                 - フィールド参照
  attribute a3,a4:Int       - 属性
  state s3                  - 初期状態
  state s4                  - 状態
  transition                - 初期遷移/状態遷移規則の記述開始
  start is                  - 初期遷移 start
    source init
    desitination s3         - 初期状態
  end
  t10 is
    source s3               - 遷移元状態
    input f.e1              - 入力イベント
    when  f.e3.num := a4    - 遷移条件
    do    a3 := a3 + 1      - 動作(属性計算)
    desitination s4         - 遷移先状態
  end
  t11 is
    surce s4
    input f.e2
    do    a4 := 0
    output f.e3             - 出力イベント
    destination s3
  end
end
```

図 2.4: オブジェクトクラス記述例

## 2.6 本研究の目的とアプローチ

前節までに，動的モデルはシステムの動的振る舞いを記述する重要なモデルであることを述べた．

実際の開発においてはシステムに対する要求仕様は何度も変更され，その都度，ユースケースモデルに対して新しい機能の追加，あるいは機能の修正といった変更が行なわれる．システム開発で作成されるすべてのモデルはユースケース主導開発によって作成されるため，ユースケースモデルが変更された場合，各モデルもそれに応じて変更する必要がある．

そのため，動的モデルはユースケースモデルに基づいて何度も作成/変更される．しかしこのプロセスでは，ユースケースモデルが変更される度に開発者自身が動的モデルを作成/変更する必要があり，効率的な開発を行なうことが難しくなっている．

そこで，本研究では動的モデル作成を計算機によって支援する環境を構築することにより，ユースケースモデルに基づいた動的モデル作成/変更を効率的かつ，柔軟に行なう一貫した開発方法を提案する．

動的モデルはオブジェクト間の振る舞いをイベントと状態を用いて記述したものである．そのため，ユースケースモデルからオブジェクトとその間のメッセージシーケンスを抽出することによって，計算機を用いて動的モデルを生成することができるのではないかと考えた．

ユースケースのシナリオには各機能を実現するためのイベントフローが記述されているため，そこからメッセージシーケンスを抽出することは比較的容易なことである．しかし，シナリオは自然言語で記述されるため，計算機を用いてシナリオからメッセージシーケンスを抽出することは困難なことである．そのため，一般的にユースケースのメッセージシーケンスを記述するモデルとして使用されているシーケンス図を用いることによって形式的な記述を与えることにした．

通信分野ではメッセージシーケンスの記述に，標準言語メッセージシーケンス図 (MSC) が広く使われている．この言語の特徴は，グラフィカルな構文とそれに対応するテキスト形式構文を持ち，それぞれに形式的なシンタクスと非形式的なセマンティクスが与えられていること，そしてメッセージシーケンスの構造を扱うための概念が用意されていることである．そのため，本研究では MSC を用いてメッセージシーケンスを形式的に記述することにした．また，動的モデルを作成するために，メッセージシーケンスの構造とメッセージのやり取りに伴うオブジェクトの振る舞いをより詳細に記述する必要があるため

MSC の再定義を行なった．

また，本研究では作成する動的モデルとして可読性と再利用性に優れている ObTS と ObTS に基づく仕様記述言語 ObCL を用いることにし，MSC から ObCL に変換する環境を構築した．生成された ObCL コードは，ObCL コンバーターによって ML コードに変換され，シミュレーター ObML 上で実行/テストすることができる．この結果をフィードバックすることによって分析/設計の段階で動的モデルを洗練することができる．

MSC については 3 章で詳しく説明し，それを基に独自に定義した MSC を 4 章で述べる．

# 第 3 章

# メッセージシーケンス図(MSC)

本章では，本研究の基礎である MSC について詳しく説明する．

## 3.1 MSCの概要

MSC は，Basic MSC と構造概念により構成されている．Basic MSC は，オブジェクト間のメッセージの流れを表現し，一つの Basic MSC には，システムの部分的な振る舞いを記述する．構造概念は，MSC を合成したり異なるレベルの MSC（の一部分）を再利用することを可能とするための概念である．

MSC はテキスト形式で，あるいはグラフィカル形式で表現することができる (図 3.1, 図 3.2)．この 2 つの表現には，それぞれ形式的なシンタクスと非形式的なセマンティクスが Z.120[1] で与えられている．さらに，Basic MSC のテキスト形式構文では各インスタンスごとのイベント順序に着目したインスタンス指向テキスト形式シンタクスと，MSC 全体のイベント順序に着目したイベント指向テキスト形式シンタクスの 2 通りの記述法がある（図 3.2，図 3.3）．

以下では，Basic MSC と構造概念について述べる．

### 3.1.1 Basic MSC

Basic MSC はオブジェクト間のメッセージシーケンスを記述するのに必要な，基本的な構成要素を含んでいる．それらは，インスタンス，メッセージ，外部環境とゲート，一般的な順序，状態，タイマー，アクション，インスタンス生成，そしてインスタンス終了

から構成される．

### インスタンス

インスタンスは実体の相互作用から成り，インスタンスの実体はその実体の性質を持ったオブジェクトである．グラフィカル表記では，インスタンスの先頭部分にはインスタンス名に加えて実体名，例えば，プロセス名を記述することができる．また，インスタンス本体（軸）には，イベント順序を記述する．

### メッセージ

MSC でのメッセージは，入力と出力との間の関係である．メッセージ出力は,（ゲートを通じて）外部環境やインスタンスから出力される．また，メッセージ入力は,（ゲートを通じて）外部環境やインスタンスに入力される．

出力メッセージや入力メッセージが失われた場合には，不完全なメッセージ (incomplete message) が出現し，1 つのイベント[1]として記述する．

2 つのインスタンス間でのメッセージ交換は，メッセージ入力とメッセージ出力の 2 つのイベントに分けることができる．テキスト形式の表記では，メッセージ入力はキーワード **in** によって記述し，メッセージ出力はキーワード **out** によって記述する．それに続いて，メッセージ名，そしてオプションとして，メッセージインスタンス名を記述する．また，1 つのメッセージには丸括弧の間にパラメータを記述することができる．

メッセージ出力とメッセージ入力との対応は，1 対 1 に定義しなければならない．テキスト形式表記では，入力と出力の対応づけは，メッセージ名とアドレスから行なっている．グラフィカルな表記では，メッセージは矢印で記述する．

メッセージの喪失（例えば，メッセージは送信されたが受信されないような場合）は，テキスト形式表記ではキーワード **lost** で記述し，グラフィカル表記では，黒丸で記述する．

テキスト形式表記において，キーワード **before** によって，異なるインスタンス上のイベントの時間順序が定義することができる．グラフィカル表記では，各インスタンスの軸に沿って記述されたイベントによって，イベントの全ての順序が決まる．異なるインスタンスのイベントは，メッセージによってのみ順序付けられる．

[1]メッセージとイベントの関係について．メッセージが受送信されたときに起るのがイベントである．

### 外部環境とゲート

ゲートはMSCと外部環境とのインタフェースを表現する．MSC境界に属するメッセージや順序関係によってゲートが記述される．

### 一般順序

一般順序は，メッセージの順序からは推測できないが，2つのイベントが時間的に順序づけられている時に使われる．

### 状態

状態は，MSCに含まれる全てのインスタンスから参照されるグローバルシステム状態（グローバル状態）やインスタンスの部分集合から参照される状態（局所状態）を表す．テキスト形式表記では，状態は，各インスタンスごとに，キーワード **condition** と状態名で定義する必要がある．いくつかのインスタンスに参照されている状態は，キーワード **shared** と共に，その状態を参照しているインスタンスの集合を列挙したインスタンスリストによって定義し，全てのインスタンスに参照されるグローバル状態は，キーワード **shared all** によって定義する．

グローバル状態は，HMSC（後述）において，MSCをどのように合成することができるのかを制限するために使うことができる．

### タイマー

MSCでは，タイマーセットとその後のタイムアウト，あるいは，タイマーセットとその後のタイマーリセット（時間管理）を記述できる．それに加えて，それらの要素は複数のMSCをまたいで，別々に使用できる．グラフィカル表記では，タイマーセット記号は,（曲がった）線でインスタンス軸とをつなぐ砂時計で表される．タイムアウト記号は,砂時計記号にくっついて，インスタンス軸を指すメッセージ矢印で表される．タイマーリセット記号は,（曲がった）線でインスタンス軸とをつなぐ×印で表される．

タイマーインスタンス名やタイマー存続時間は，テキスト形式やグラフィカル表記の両方でオプションとして記述できる．

#### アクション

メッセージ交換に伴って，MSC 内にアクションを記述できる．アクションは，自然言語で記述する．

#### インスタンス生成

SDL と同様に，インスタンスの生成と終了を MSC 内に記述できる．インスタンスは，他のインスタンスによって生成される．インスタンスが生成する前に，そのインスタンスとメッセージを交換することはできない．

#### インスタンス終了

ある意味，インスタンス終了は，インスタンス生成と対称な事柄である．しかし，他のインスタンスから生成されるインスタンス生成とは違って，インスタンス終了は，自分自身でしか終了することができない．

### 3.1.2 Basic MSC の記述例

Basic MSC の例として，図 3.1 を考えます．この例では，インスタンスとして “Initiator” と “Responder” が，メッセージとして “ICONreq”, “ICON”, “ICONind”, “IDISind” が，状態として “Disconnected”, “wait”, “disconnected” （“Disconnected” がグローバル状態，“wait” と “disconnected” が共有状態）が，アクションとして “setting_count” が，そして，タイムアウトとしてタイマーを 5 にセットしリセットする “T(5)” が記述されています．

この図ではまず，ユーザが “Initiator” に要求 (“ICONreq”) を送信し，“Initiator” が，“Responder” に要求 (“ICON”) を送信する．その後，“Responder” はユーザに要求 (“ICONind”) を送信している．また，“Initiator” は，“Responder” に要求 (“ICON”) を送ったあとアクション “setting_count” を行ない，5 待った（タイムアウト）後にユーザに “IDISind” を送信している．また，“Initiator” は，“ICONreq” を受信すると状態 “Disconnected” から遷移し，“ICON” を送信し，アクション “setting_count” を行ない，タイマーを 5 にセットした後状態 “wait” に遷移し，タイムアウト後に状態 “disconnected” に遷移しており，一方，“Responder” は “ICON” を受信すると状態 “Disconnected” から遷移し，“ICONind” を送信した後状態 “wait” に遷移していることがわかる．

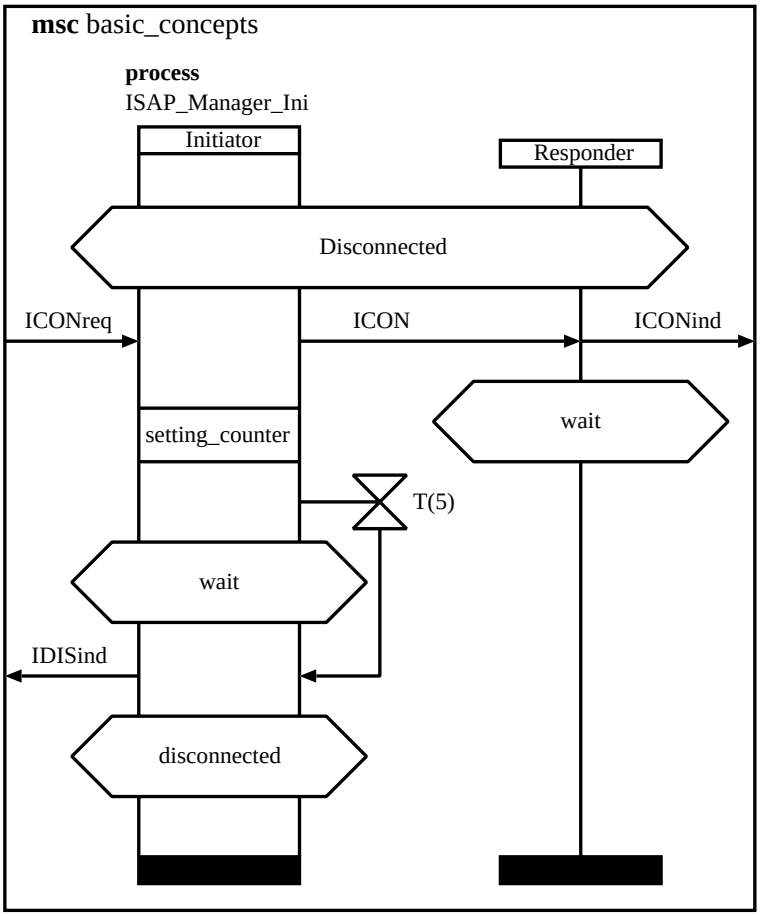



図 3.1: グラフィカル表記

図3.1のテキスト形式表記を，インスタンス指向テキスト表記（図3.2）とイベント指向テキスト形式表記（図3.3）で示す．インスタンス指向テキスト表記では，各インスタンスごとに入出力イベント，アクション，タイマーセット/リセット/タイムアウト，そして状態といったBasic MSCの要素を時系列順（そのインスタンスの上から順）に記述していく．一方，イベント指向テキスト形式表記では，各インスタンスをまたいでBasic MSCの要素を時系列順に記述していく．

```
msc basic_concepts; inst Initiator: process ISAP_Manager_Ini, Responder;
        instance Initiator: process ISAP_Manager_Ini;
            condition Disconnected shared all;
            in ICONreq from env;
            out ICON to Responder;
            action setting_counter;
            set T(5);
            condition wait shared;
            timeout T;
            out IDISind to env;
            condition desconnected shared;
        endinstance;
        instance Responder;
            condition Disconnected shared all;
            in ICON from Initiator;
            out ICONind to env;
            condition wait  shared;
        endinstance;
endmsc;
```

図 3.2: インスタンス指向テキスト形式表記

```
msc basic_concepts; inst Initiator: process ISAP_Manager_Ini, Responder;
Initiator:          instance process ISAP_Manager_Ini;
Responder:          instance;
all:                condition Disconnected;
Initiator:          in ICONreq from env;
                    out ICON to Responder;
Responder:          in ICON from Initiator;
                    out ICONind to env;
                    condition wait;
                    endinstance;
Initiator:          action setting_counter;
                    set T(5);
                    condition wait;
                    timeout T;
                    out IDISind to env;
                    condition desconnected shared;
                    endinstance;
endmsc;
```

図 3.3: イベント指向テキスト形式表記

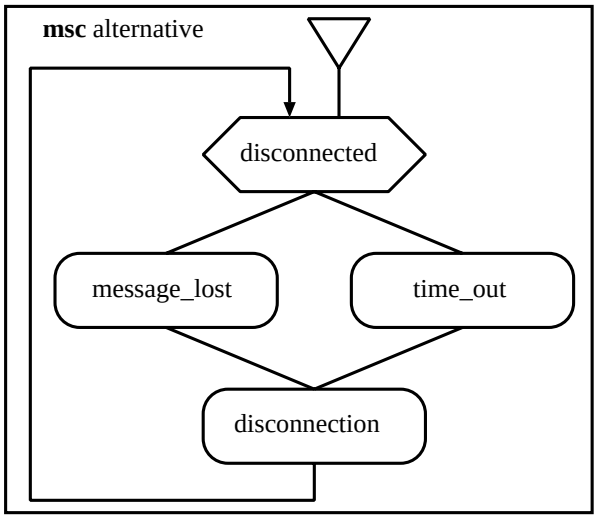
### 3.1.3 構造概念

構造概念は，インスタンス上の順序づけされていないイベントを記述するための coregion，インスタンス間の抽象度を同じにするために詳細化や抽象化を行なうインスタンス分解と合成 (instance decomposition/composition)，Basic MSC 内で，分岐，平行合成，ループ，例外，そしてオプション処理を用いてイベント列合成を行なうためのインライン表現 (inline expression)，MSC ドキュメント内の他の MSC を参照するための MSC 参照 (MSC reference)，そして，MSC の集合がどのように結び付いているかを定義する High-level MSC(HMSC) によって構成されます．MSC ではこれらの概念によって，MSC を合成したり異なるレベルの MSC（の一部分）を再利用することを可能としている．

HMSC は，開始記号 (start symbol)（各 HMSC は，開始記号を 1 つしか持つことができない），終了記号 (end symbol)，MSC 参照，状態，接続点 (connection point)，そして，平行フレームから構成されている．

HMSC の例として，図 3.4 でグラフィカル表記とテキスト形式表記を示す．この例では，開始記号，状態 “disconnected”，3 つの参照 MSC “message_lost”，“time_out”，そして，“disconnection” を使って記述している．

この図では，システムは最初に状態 “disconnected” から始まり，MSC “message_lost” と “time_out” のどちらかを参照し，最後に MSC “disconnection” を参照してから状態 “disconnected” に戻るという実行系列が記述されている．




**msc** alternative;
**expr** L1;
L1: **condition** disconnected **seq** (L2 **alt** L3);
L2: message_lost **seq** (L4);
L3: time_out **seq** (L4);
L4: disconnection **seq** (L1);
**endmsc**;

図 3.4: グラフィカル表記とテキスト形式表記

# 第 4 章

# 動的モデル作成支援環境

本章では，動的モデル作成支援環境について述べる．

## 4.1 動的モデル作成支援環境の概要

本研究で構築した動的モデル作成支援環境は，ユースケースモデルのシナリオから抽出したオブジェクト間のメッセージシーケンスを入力とし，動的モデルを出力する環境である（図 4.1）．

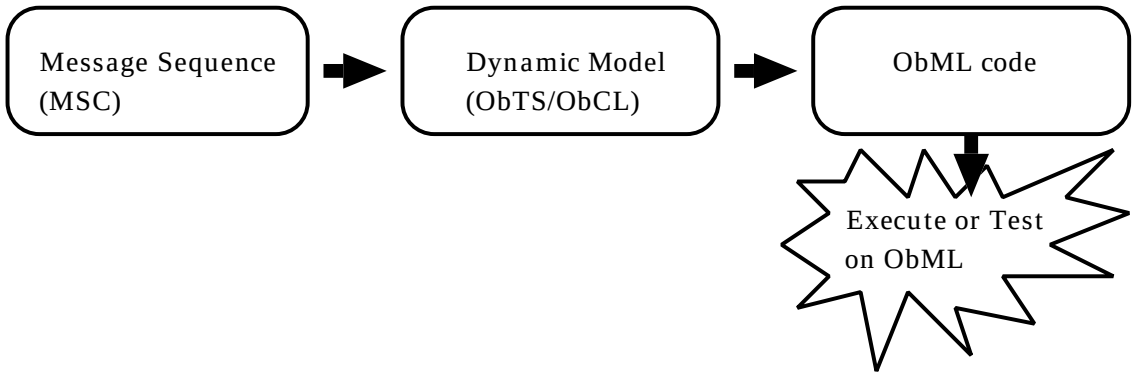



図 4.1: 動的モデル作成支援環境を使った動的モデルの作成

2 章で述べたように，この環境ではオブジェクト間のメッセージシーケンスを記述するために MSC を，また，動的モデルとして ObTS/ObCL を使用し，MSC → ObCL コンバーターにより変換を行なっている．この環境により，ユースケースモデルからオブジェクトとその間のメッセージシーケンスを抽出すれば，動的モデルを作成することが可能と

なる．また，作成された ObCL コードを ObCL コンバータを使って ObML コードに変換すれば，シュミレータ ObML 上で実行/テストを行なうことが可能である（図 4.1）．このように，この環境を用いれば，ユースケースモデルから動的モデルまでを効率的に開発することが可能となる．

以下の節では，動的モデル作成支援環境で用いるために再定義したメッセージシーケンス記述言語 MSC，MSC → ObCL コンバーター，そしてこの環境を用いた動的モデル作成プロセスについて説明を行なう．

## 4.2 メッセージシーケンス記述言語 MSC

動的モデル作成支援環境で用いる MSC は，オブジェクト間のメッセージシーケンスとその相互作用によるオブジェクトの動作を記述するために Basic MSC を，また，MSC の分岐/合流を記述するために構造概念の HMSC を使用する．しかし，この MSC は，前章で述べた MSC とはいくつか表記法が異なる．そのため，この節ではこのことについて Basic MSC と HMSC とに分けて詳しく述べていく．また，付録 A に，動的モデル作成支援環境で用いる MSC のテキスト形式の文法を示す．

### 4.2.1 Basic MSC

本研究では，Basic MSC をオブジェクト間のメッセージシーケンスとその相互作用によるオブジェクトの動作を記述するものとして使用している．また，各インスタンスごとのメッセージ通信の順序に着目するため，インスタンス指向テキスト形式構文を使用することにした．Basci MSC を記述する際には，同じ Basic MSC を MSC ドキュメント内に二重に定義してはならないものとする．以下では，標準の Basic MSC と本研究で用いる Basic MSC との違いを基本的な構成要素ごとに述べていく．

#### インスタンス

インスタンスには，ユースケースのシナリオから抽出したオブジェクトを記述する．また，外部環境もインスタンスとして記述する．ここでいう外部環境は，ユースケースでいうアクターにあたる．なお，外部環境に対応するインスタンスは，“USER” で始まる名前をつけるものとする．また，インスタンスは，途中で生成されたり消滅することはない．

インスタンスのグラフィカル表記の例を図 4.2 に，テキスト形式表記の例を図 4.3 に示す．ここで，“USER” は外部環境（アクター）であり，“ATM” と “BANK” がオブジェクトである．

### メッセージ

インスタンス間のメッセージ通信を記述する．メッセージとイベントの関係は，オブジェクトが出力イベントを送信すれば，同時にメッセージが送信されるものとし，オブジェクトがメッセージを受信したら，同時に入力イベントを受信するものとする．送信されたメッセージは途中で失われることはなく，同期して受信される．そのため，メッセージの順序とイベント順序は同じである．また，各オブジェクトにおけるイベント順序は，必ず一意に決まっていなくてはならない．

メッセージには，Int 型と String 型のパラメータ（以降，属性と呼ぶ），そして，文字列定数の 3 種類の属性を複数記述することができる．Int 型，あるいは String 型の属性を持つ場合，属性名に続いて，コロン (:)，そして，キーワード **Int**，あるいは **String** を記述する．文字列定数の場合は，文字列を二重引用符 (") で括って記述する．あるオブジェクトが受信したイベント属性は，そのオブジェクトが次のメッセージを受信，あるいは，メッセージを送信すると失われる．

また，属性を記述する際には，以下のことに気をつける必要がある．

- あるインスタンス間で使われる同じメッセージの属性の型は，すべて等しくなければならない
- 入力イベント属性を出力イベント属性に渡す場合は，必ず明記する
- 入力イベント属性をアクション部分で使用する場合は，必ず明記する
- 入力イベント属性を HMSC の分岐条件部分で使用する場合は，必ず明記する
- 入力イベントのすべての属性は，存在しない場合を除いて，アクション，分岐条件，出力イベントのうち少なくとも一ヶ所には出現しなくてはならない

メッセージの送受信相手は，テキスト形式表記では，**inst** 部分で宣言されていなければならない．さらに，インスタンスはメッセージを自分自身に送信することはないものとする．なお，オブジェクトの出力イベント属性で，文字列定数でもなく，入力イベント属性と異なる属性をオブジェクト属性とみなすものとする．

メッセージのグラフィカル表記の例を図 4.2 に，テキスト表記の例を図 4.3 に示す．ここで，メッセージ “msg” は，オブジェクト “ATM” から “USER” へ送信されるメッセージであり，属性として文字列定数 “input password.” を持つ．メッセージ “passin” は，属性 “USER” から “ATM” へ送信されるメッセージであり，属性として String 型の “password” を持つ．そして，メッセージ “checkaccount” は，オブジェクト “ATM” から “BANK” へ送信されるメッセージであり，属性として Int 型の “num” と String 型の “password” の 2 つの属性を持つ．また，この図よりメッセージは，“msg”，“passin”，そして “checkaccount” の順に通信していることがわかる．

#### 外部環境とゲート

外部環境はインスタンスとして記述するので，使用しない．

#### 一般的な順序

メッセージの順序とイベント順序は一意に決まっているので，使用しない．

#### 状態

Basic MSC では，オブジェクト間のメッセージシーケンスとそれに伴うオブジェクトの動作の 2 つにのみ着目して使用するため，状態は使用しない．

#### タイマー

送信されたメッセージは途中で失われることはなく同期して受信されるため，タイマーは使用しない．

#### アクション

アクションでは，オブジェクトの動作を関数的な属性計算によって記述する．属性計算で参照できる属性は，そのオブジェクトが持つオブジェクト属性，そのオブジェクトが直前に受け取った入力イベント属性，そして文字列定数である．また，注釈として自然言語でも記述できる．この場合，キーワード **action** の次に，キーワード **note**，コロン (:) に続けて自然言語で記述する．この時，アクションの終りにはセミコロン (;) を記述するこ

とに注意が必要である．注釈は ObCL の動作部分にそのまま出力される．なお，属性計算部分では，文字列定数でもなく，評価式の左辺の属性と右辺の入力イベント属性と異なる属性をオブジェクト属性とみなすものとする．

アクションのグラフィカル表記の例を図 4.2 に，テキスト形式表記の例を図 4.3 に示す．ここで，“count” は，オブジェクト属性である．このように，アクションではオブジェクト属性に属性計算によって得られた値を代入するように記述しなければならないとする．そのため，式の左辺の項は，オブジェクト属性である．

### インスタンス生成と終了

すべてのインスタンスはシステム起動時に生成され，途中で新しいインスタンスを生成したり，存在していたインスタンスが終了したりしない．そのため，使用しない．

### Basic MSC 記述時の注意事項

Basic MSC 記述の際には，以下の規則を守る必要がある．

- 各オブジェクトには，入力イベントを受け取って,（属性を評価して），出力イベントを出すという一連の動作を必ず記述する
- 各オブジェクトは，入力イベントで始まる
- Basic MSC 内の各オブジェクトは，入力イベント，アクション，および出力イベントで始まる．特に，アクション，または出力イベントで始まる場合は，メッセージシーケンスが条件分岐，または合流した時である
- Basic MSC 内の各オブジェクトは，入力イベント，あるいは，出力イベントで終わる．特に，入力イベントで終わる場合は，メッセージシーケンスが条件分岐，または合流する時である
- 上記より，各オブジェクトのメッセージシーケンスでは，アクションだけからなるメッセージシーケンスを記述できない

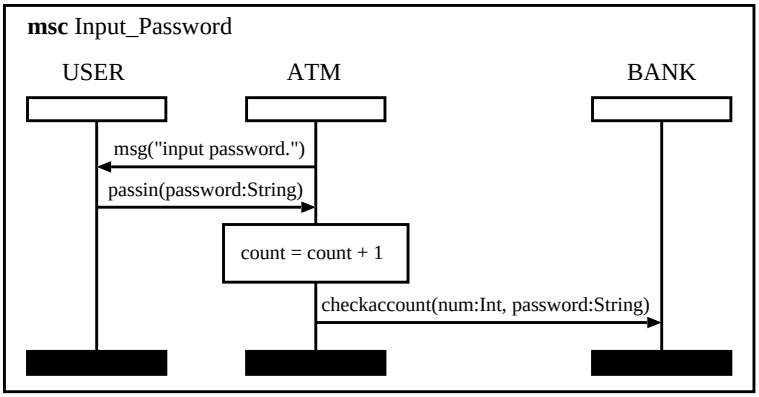



図 4.2: Basic MSC のグラフィカル表記の例

```
msc Input_Password;
inst USER, ATM, BANK;
    instance USER;
        in msg("input password.") from ATM;
        out passin(password:String) to ATM;
    endinstance;
    instance ATM;
        out msg("input password.") to USER;
        in passin(password:String) from USER;
        action count = count + 1;
        out checkaccount(num:Int, password:String) to BANK;
    endinstance;
    instance BANK;
        in checkaccount(num:Int, password:String) from ATM;
    endinstance;
endmsc;
```

図 4.3: Basic MSC のテキスト形式表記の例

### 4.2.2 HMSC

本研究では，HMSC を MSC の分岐/合流を記述するものという観点から使用している．そのため，分岐/合流の目印として状態を使用している．この HMSC では，要素の 1 つである平行フレームは使用しない．また，HMSC のテキスト形式表記には，メッセージシーケンスの分岐についてさらに詳細に記述できるように，if-else 文を用いて条件分岐を追加した．なお，HMSC を記述する際には，root の HMSC を MSC ドキュメントの一番先頭に記述し，同じ HMSC を二重に定義してはならない．また，HMSC 以外の構造概念は使用しないものとする．

HMSC は参照 MSC によって MSC を階層的に記述できるため，システムの振る舞いを抽象度の高いレベルから段階的に詳細なレベルに記述していくことが可能である．

#### 条件分岐

条件分岐は，キーワード **if**，あるいは **else if** に続けて分岐条件を丸括弧で括って記述することにより表現する．分岐条件では，属性に関する条件式で記述する．条件式で参照できる属性は，条件分岐を行なうオブジェクトが持つオブジェクト属性とそのオブジェクトが直前に受け取った入力イベントの属性，そして，文字列定数である．メッセージシーケンスのループを記述する際は，条件分岐によって記述する．また，注釈として自然言語でも記述できる．この場合，分岐条件の先頭で，キーワード **note**，そして，コロン (:) に続けて自然言語で記述する．注釈は，ObCL の遷移条件部分にそのまま出力される．なお，分岐条件部分では，文字列定数でもなく，入力イベント属性と異なる属性をオブジェクト属性とみなすものとする．

HMSC で条件分岐を使って記述する必要があるのは，以下の 3 つの場合である．

1. 複数の入力イベントごとに，それぞれの遷移が存在する場合
2. 入力イベントの属性値により，遷移が存在するか決まる場合
3. 入力イベントの属性値により，複数の遷移が存在する場合

これらの場合は，すべて HMSC に条件分岐として記述する．ただし，1. の場合は分岐条件を記述する必要はない．

### 状態

状態は，HMSC の初期状態と終了状態，そして，Basic MSC の分岐/合流の目印としてのみ使用する．条件分岐/合流がないところでは状態は使用しない．

### 参照 MSC

HMSC が一度に参照できる MSC の数は 1 つとする．動的モデル作成支援環境で用いる MSC は，Basic MSC と HMSC とで構成されるため，参照できる MSC もこの 2 つのみである．

### HMSC 記述時の注意事項

HMSC 記述の際には，以下の規則を守る必要がある．

- 状態は，HMSC の初期状態と終了状態，そして，Basic MSC の分岐/合流の目印としてのみ使用する
- **seq** 部分のラベル名は，その行の先頭のラベル名とは異なる
- **condition** 文の次に，**condition** 文が続いてはならない
- **if-else** 文の次に，**condition** 文，または，**if-else** 文が続いてはならない
- MSC 参照文の次に，**if-else** 文，または，MSC 参照文が続いてはならない

HMSC のグラフィカル表記の例を図 4.4 に，テキスト形式表記の例を図 4.5 に示す．ここでは，分岐条件として “count” を使って “count” が 3 より大きいかどうかで与えている．また，分岐条件を使ってループ（状態 “disconnected” →参照 MSC“failure” →状態 “disconnected”）を記述している．

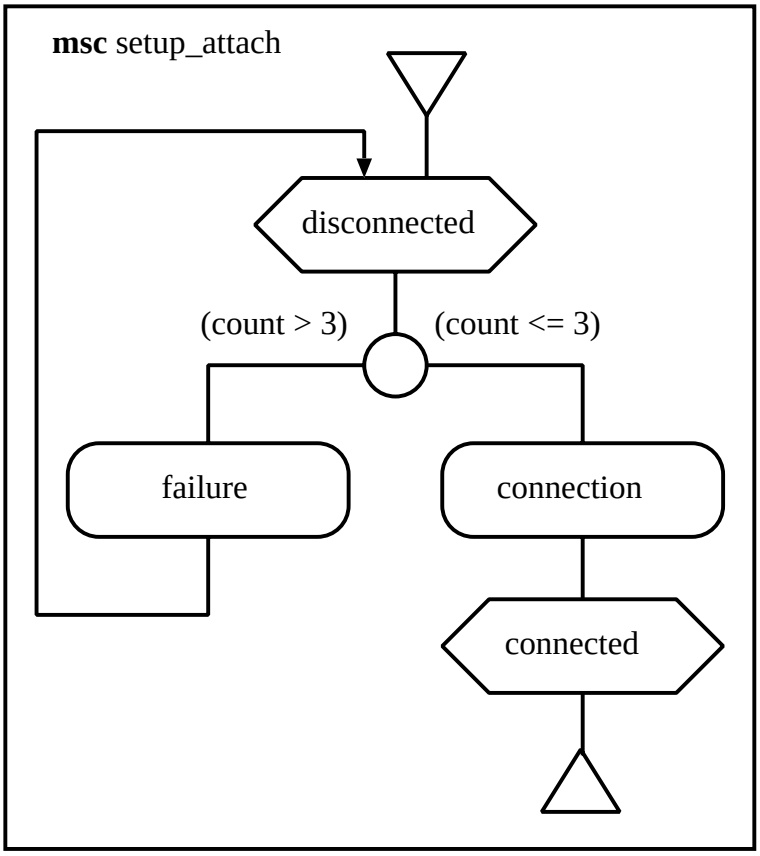



図 4.4: HMSC のグラフィカル表記の例

```
msc setup_attach;
expr L1;
        L1: condition disconnected seq (L2);
        L2: if (count > 3) seq (L3);
            else if ((count < 3) and (count == 3)) seq (L4);
        L3: failure seq (L1);
        L4: connection seq (L5);
        L5: condition connected seq (L6);
        L6: end;
endmsc;
```

図 4.5: Basic MSC のテキスト形式表記の例

## 4.3 MSC→ObCL コンバーター

MSC → ObCL コンバーターは，MSC によって記述したオブジェクト間のメッセージシーケンスを，動的モデル仕様記述言語 ObCL に変換するものである．以下の節では，ObCL の属性クラス/イベントクラス/フィールドクラス/オブジェクトクラスの各クラス，およびシステム記述ついての説明と生成アルゴリズムについて述べる．

### 4.3.1 属性クラス

属性クラスを用いることで，属性の型として ML 上の任意の型を使用することができる．本研究ではデフォルトで用意されている Bool 型，Int 型，および String 型のみを使用しているため，新たに定義する必要はない．

### 4.3.2 イベントクラスの生成

イベントクラスは，イベントそのものの記述とその記述の再利用のためのもので，クラス名と属性の他に属性に対する操作を持つ．ただし，本研究では，属性に対する操作は扱わない．

また，属性も操作も持たない単純なイベントクラスとして，**GENERIC_EVENT** クラスが予約されており，すべてのイベントクラスはこのクラスを継承して記述する．

イベントクラスの記述例を以下に示す．ここでは整数型の属性 “val” を持つイベントクラス “ATTRIBUTED” を定義している．

```
event ATTRIBUTED
  inherit GENERIC_EVENT
attribute val:Int
end
```

コンバーターはイベントクラスを生成するために，まず，Basic MSC に記述されたオブジェクト間のイベント通信から，ある 2 つのオブジェクト間を行き来するイベントの通信路を 1 つのフィールドとみなし，そのフィールドに属するイベントを抽出する．次に，各フィールドごとに，そのフィールドに属するイベントから，Int 型属性を最も多く持つイベントの Int 型属性の数と，String 型属性を最も多く持つイベントの String 型属性の数を求める．そして，この二つの数から，**GENERIC_EVENT** クラスを継承することによって各フィールドごとにイベントクラスを生成する．生成されたイベントクラスは，Int

型属性のみを持つクラスと String 型属性を含むクラスの 2 つに分類できる．

図 4.6 にフィールド抽出の例を示す．ここでは，MSC A, B, C からフィールド “FIELD1” と “FIELD2” が抽出できることを示している．また，“FIELD1” には，イベント “e1”, “e2”, “e3”, “e6”，“e7” が，“FIELD2” には，イベント “e4”, “e5” が属していることがわかる．

図 4.7 に，フィールド “FIELD1” に属するイベントから “FIELD1” のイベントクラスを生成する例を示す．図で示した Int 型属性の最大数が 0 でないとき，**GENERIC_EVENT** クラスを継承して，その数だけ Int 型属性を持つクラスを生成する．ここでは，2 つの Int 型属性を持つ “FIELD1_INTEVENT” クラスを生成している．次に，String 型属性の最大数より，“FIELD1_INTEVENT” クラスを継承して，その数だけ String 型属性を持つクラスを生成する．ここでは，1 つの String 型属性を持つ “FIELD1_STREVENT” クラスを生成している．なお，Int 型属性の最大数が 0 であるとき，“FIELD1_STREVENT” クラスは **GENERIC_EVENT** クラスを継承して生成される．

### 4.3.3 フィールドクラスの生成

フィールドクラスは，フィールドの記述とその記述の再利用のためのもので，そのクラスに属するインスタンス（つまり，フィールド）に流れるイベントを記述する．

フィールドクラスの記述例を以下に示す．ここでは “GENERIC_EVENT” クラスのイベント “e1”，“e2” と，“ATTRIBUTED” クラスのイベント “e3” が流れるようなフィールドクラス “F” を定義している．

```
field F
  event e1,e2:GENERIC_EVENT
  event e3:ATTRIBUTED
end
```

フィールドクラスに関連して，イベントはフィールドに流れるのでフィールド名とイベント名の対によって “f.e3” のように記述する．また，同様にイベント属性はフィールド名とイベント名と属性名を連結して，“f.e3.val” のように記述する．

コンバーターは，フィールドクラスを生成するために Basic MSC に記述された同一フィールドに属する全てのイベントを，属性を持たないイベント，Int 型属性しか持たないイベント，そして，String 型属性を持つイベントの 3 つのグループに分け，フィールドクラスを生成する．

図 4.8 に，フィールド “FIELD1” に属するイベント属性から “FIELD1” のフィールドクラスを生成する例を示す．ここで，“FIELD1_INTEVENT”，そして “FIELD1_STREVENT”

は，図 4.7 で定義したイベントクラスである．

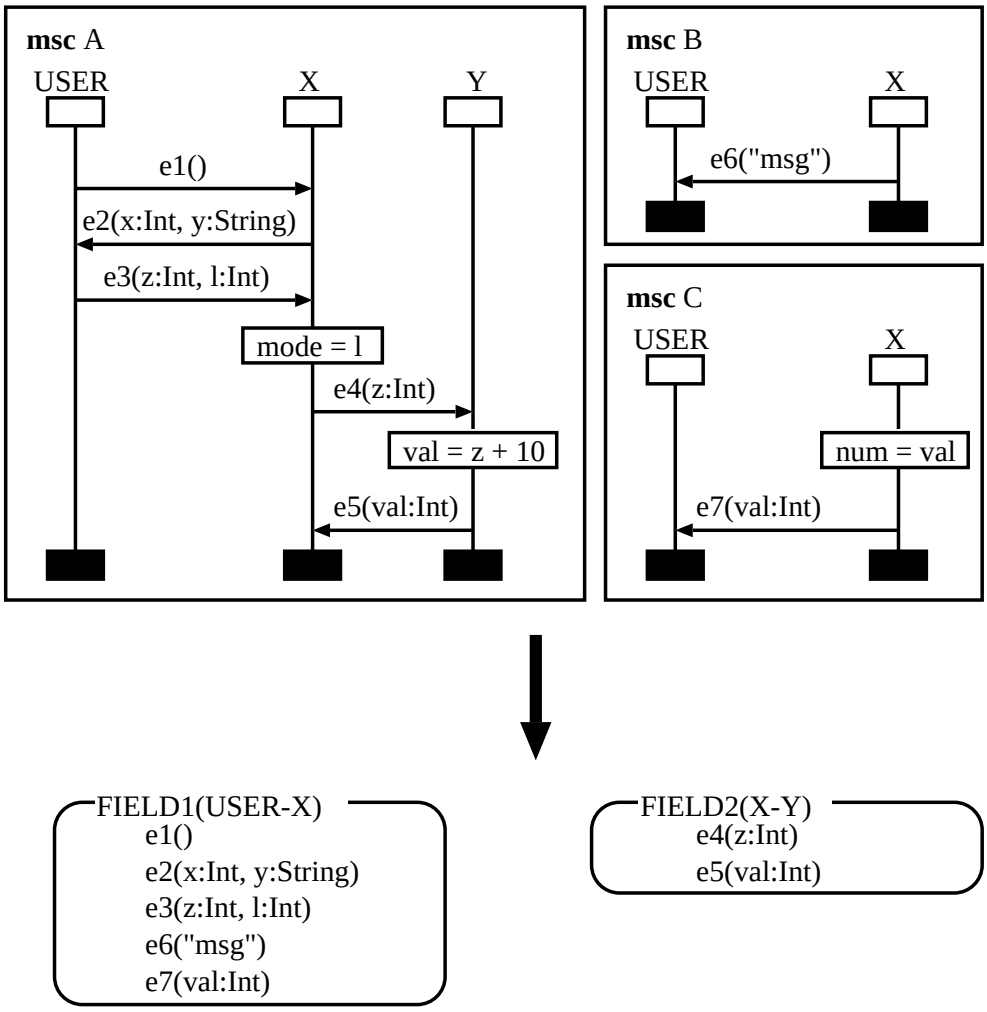



図 4.6: フィールドの抽出

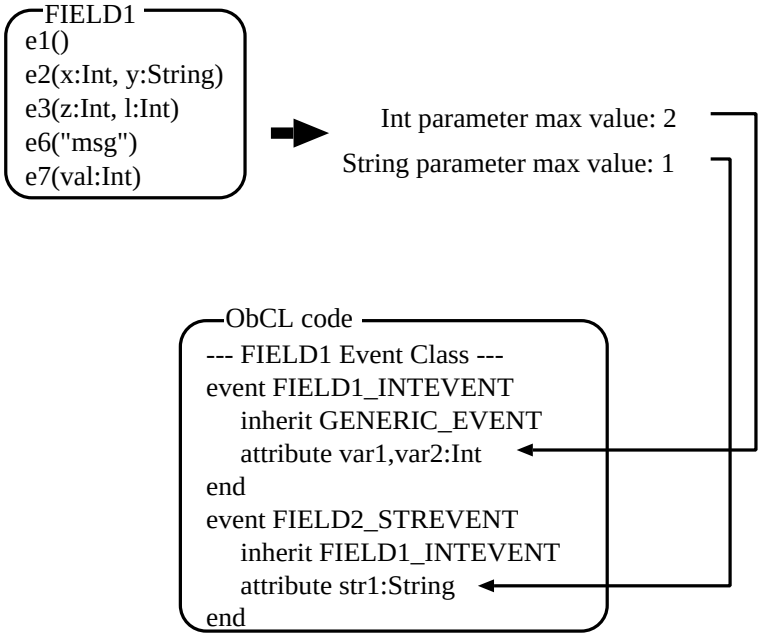



図 4.7: イベントクラス生成

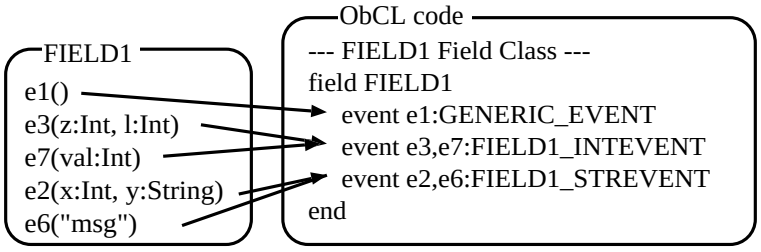



図 4.8: フィールドクラス生成

### 4.3.4 オブジェクトクラスの生成

オブジェクトクラスは，オブジェクトの記述とその再利用のためであり，オブジェクトクラス名，フィールド参照，属性，操作，状態，内部オブジェクト宣言，状態遷移規則から成る．

フィールド参照はそのクラス（オブジェクト）がどのフィールドに属すのかを示し，属性はそのクラスが局所データとして持つ属性である．操作は属性に関する操作であって状態遷移規則中で使用できる．状態はそのクラスが取りうる状態で，内部オブジェクトはそのクラスの持つ内部オブジェクトのクラスとオブジェクト名を記述する．また，並行動作する内部オブジェクト集合がある場合には，そこに含まれる個々のオブジェクト名とは別に，並行動作する内部オブジェクト全体の集合名を指定する必要がある．状態遷移規則は，遷移元，入力イベント，遷移条件，動作，遷移先，出力イベントによって指定する．遷移元/遷移先は状態名，内部オブジェクト名，または並行動作する内部オブジェクト集合名を指定する．入力イベント/出力イベントには単一のイベント，またはイベント集合が指定できる．遷移条件は属性に関する条件式で，動作は属性の値を計算する式の並びである．ただし，初期遷移の場合は遷移元に “init” と書き，入力イベントや遷移条件などは指定しない．

図 2.4 に，オブジェクト “Y” の ObCL 記述例を示す．本研究では，操作および内部オブジェクト宣言は扱わないものとする．以下にオブジェクトクラスの各部分についてコンバーターが行なうことを述べる．

#### オブジェクトクラス名とフィールド参照

コンバーターは，オブジェクトクラス名を Basic MSC に記述されたインスタンスから抽出する．フィールド参照は，ある 2 つのオブジェクト間を行き来するイベントの通信路を 1 つのフィールドとみなしているため，フィールドから，このオブジェクトが含まれるフィールドを抽出する．図 4.6 より，“X” は “FIELD1” と “FIELD2” を，“Y” は “FIELD2” を参照していることがわかる．

### 属性

コンバーターは，属性を Basic MSC に記述されたアクションと出力イベント，そして HMSC に記述された分岐条件内，それぞれの文字列定数以外の属性によって抽出する．アクション記述部分では，評価式の左辺の属性と右辺の入力イベント属性と異なる属性が，出力イベント記述部分では，入力イベント属性と異なる属性が，そして，分岐条件記述部分では，入力イベント属性と異なる属性が，オブジェクト属性であるとみなしている．図 4.6 より，“X” の属性は，アクション部分より “mode” と “num” の Int 型属性を，出力イベント部分より Int 型属性 “x” と String 型属性 “y” を，分岐条件部分より，Int 型属性 “mode” を持つことがわかる．“Y” についても同様にして，Int 型属性 “val” を持つことがわかる．

### 状態

状態は，次に述べる状態先規則より抽出する．

**状態遷移規則**

状態遷移規則は，遷移元，入力イベント，遷移条件，動作，遷移先，そして，出力イベントにより記述する．ただし，入力イベントには単一のイベントのみを記述する．

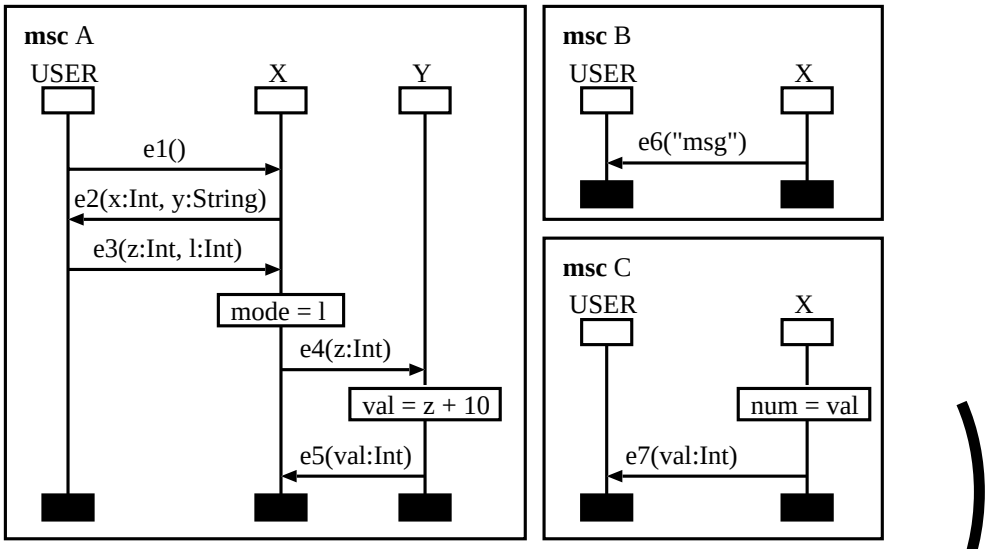
コンバーターは状態遷移規則を生成するために，まず，Basic MSC から，各オブジェクトごとに，入力イベント/アクション/出力イベントを 1 つの組とした MSC 要素を切り出し，また，HMSC から，遷移元/分岐条件/参照 MSC 名/遷移先を 1 つの組とした HMSC 要素を切り出す（図 4.9，図 4.10）．

HMSC 要素は，システムの実行系列順につながっているので，各 HMSC 要素に基づいてそれに参照されている MSC 要素を合成することによって各オブジェクトごとのイベント列が生成される（図 4.11）．図 4.12 は図 4.11 を基に各オブジェクトごとのイベント列に着目したものである．

そのイベント列から入力/出力イベント両方を持つ MSC 要素を，あるいは，入力イベントを持つ MSC 要素とそれに連続した出力イベントを持つ MSC 要素を 1 つの遷移とみなして，状態遷移規則を生成する．この時，ある HMSC 要素が参照している MSC 要素集合の先頭 MSC 要素が入力イベントを持っていれば，その遷移に HMSC 要素の遷移元状態を付加し，また，最後の MSC 要素が出力イベントを持っていれば，その遷移に HMSC 要素の遷移先状態を付加する．それら以外の遷移先/遷移元状態は，各オブジェクトごとに “s10” のように “s” と通し番号の対によって記述する．また，ある HMSC 要素が参照している MSC 要素集合の先頭 MSC 要素が入力イベントを持たない場合，その MSC 要素に HMSC 要素の遷移条件を付加する．

次に，生成された状態遷移規則から同じ状態遷移内の入力イベント属性と遷移条件，アクション，出力イベントの属性が同じでないか調べ，同じである場合，イベント属性名（フィールド名とイベント名と属性名を連結したもの）と置換する．ここで，属性名は Int 型属性ならば “var”，String 型属性ならば “str” と属性名識別番号を対として生成する．また，出力イベント属性はオブジェクトの動作によって属性値を設定する必要があるので，イベント属性名を用いて動作（属性計算）部分に追加する．

これらの操作によって得られた，各オブジェクトごとのオブジェクトクラスの例を図 4.13，図 4.14 に示す．




| msc element | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| object name | X | X | X | X |
| input event | e1 | e3 | e5 | e4 |
| action | | mode = l | | val = z +10 |
| output event | e2 | e4 | | e5 |

**msc** A

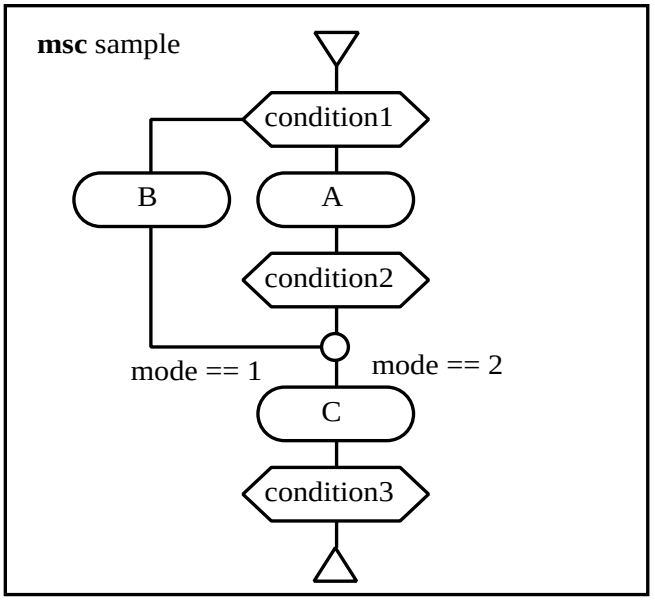
| b1 |
|---|
| X |
| |
| |
| e6 |

**msc** B

| c1 |
|---|
| X |
| |
| num = val |
| e7 |

**msc** C

図 4.9: MSC 要素の切り出し




| hmsc element | hmsc1 | hmsc2 | hmsc3 |
|---|---|---|---|
| state befor transition | condition1 | condition2 | condition2 |
| conditional statement | | mode == 1 | mode == 2 |
| reference msc element | a1,a2,a3,a4 | b1 | c1 |
| state after transition | condition2 | condition1 | condition3 |

図 4.10: HMSC 要素の切り出し

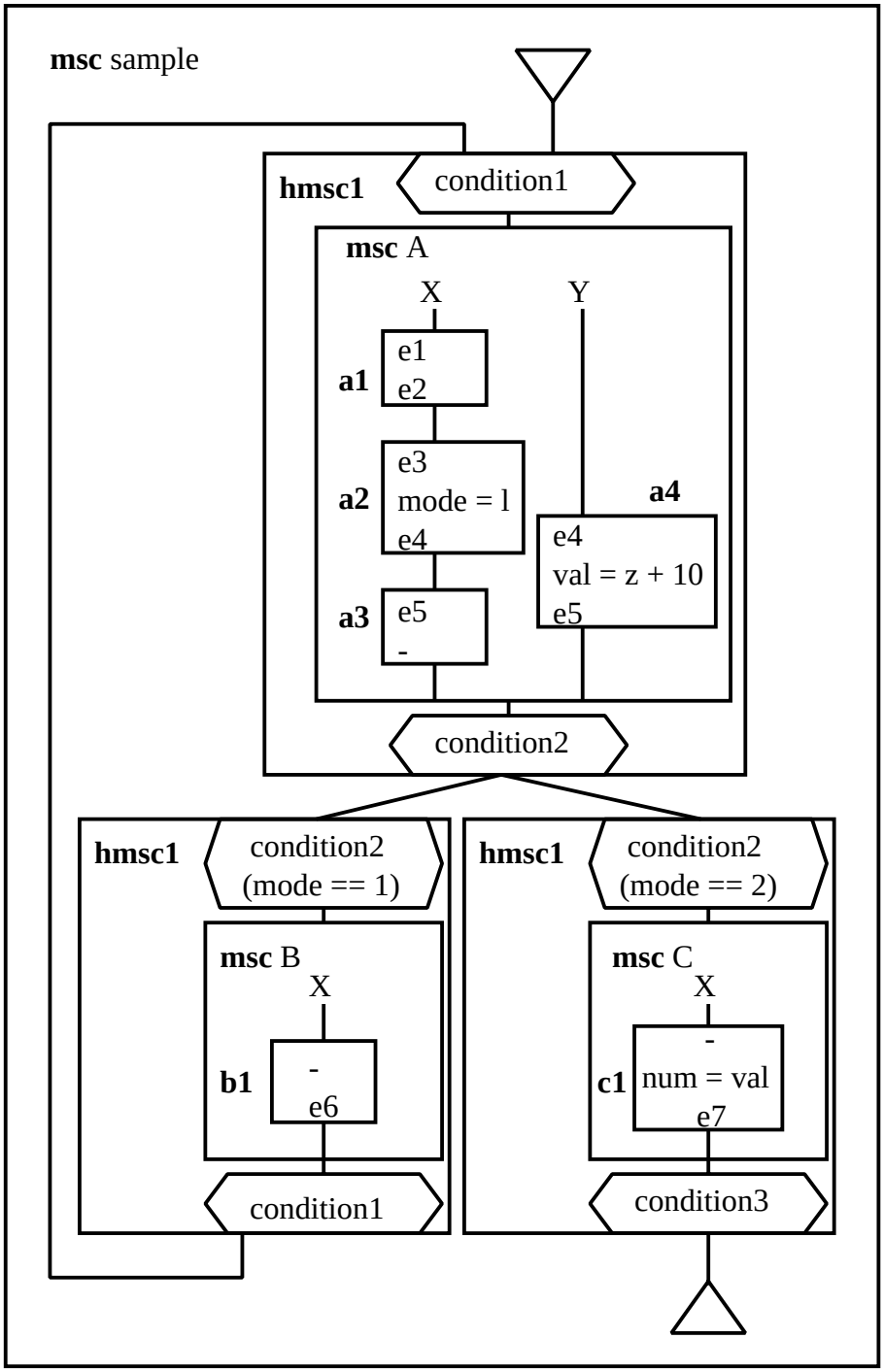



図 4.11: HMSC 要素に基づいたと MSC 要素の合成

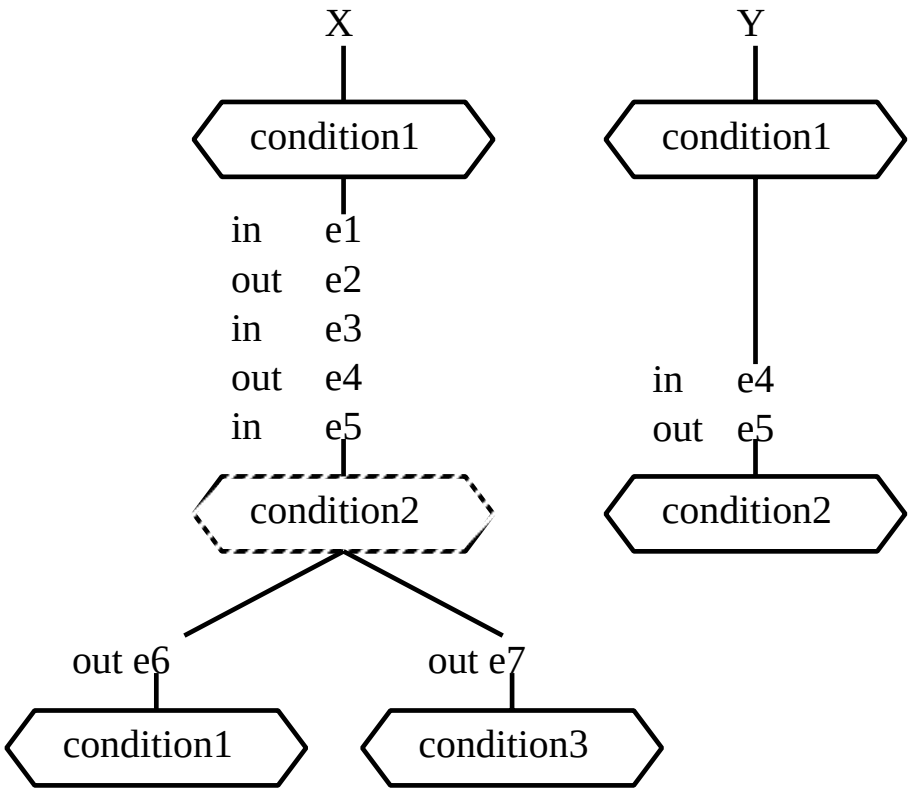



図 4.12: 各オブジェクトごとのイベント列

```
--- ObCL code ---
--- X
class X                          -X のクラス
  field F1:FIELD1;F2:FIELD2
  -フィールド参照 (F1 はクラス FIELD1 のインスタンス)
  attribute mode,num,x:Int   -Int 型属性
  attribute y:String         -String 型属性
  state condition1           -初期状態
  state s11,s12,s13,condition3 -遷移状態
  transition                 -初期遷移/状態遷移規則の記述開始
  start is                   -初期遷移 start
    source init
    destination condition1   -初期状態
  end
  t10 is                     -状態遷移 t10
    source condition1        -遷移元
    input F1.e1              -入力イベント
    do F1.e2.var1 := x;      -動作（属性計算）
       F1.e2.str1 := y
    destination s11          -遷移先
    output F1.e2             -出力イベント
  end
  t11 is                     -状態遷移 t11
    source s11
    input F1.e3
    do mode := F1.e3.var2;
       F1.e4.var1 := F1.e3.var1
    destination s12
    output F2.e4
  end
  t12 is
    source s12
    input F2.e5
    when mode = 1            -遷移条件
    do F1.e6.str1 := "msg"
    destination condition1
    output F1.e6
  end
  t13 is
    source s12
    input F2.e5
    when mode = 2
    do num := F2.e5.var1;
       F1.e7.var1 := F2.e5.var1
    destination condition3
    output F1.e7
  end
end
```

図 4.13: X のオブジェクトクラス (ObCL コード)

```
---Y
class Y                          -Y のクラス
  field F2:FIELD2
  attribute var:Int
  state condition1
  state condition2
  transition
  start is
    source init
    destination condition1
  end
  t10 is
    source condition1
    input F2.e4
    do val := F2.e4.var1 + 10
       F2.e5.var1 := val
    destination condition2
    output F2.e5
  end
end
```

図 4.14: Y のオブジェクトクラス (ObCL コード)

### 4.3.5 システム記述

システム記述により，記述対象のシステムのオブジェクト階層のトップ（または root）にあたるオブジェクトを指定する必要がある．ここで記述するのは，各クラスのオブジェクトを定義と，各オブジェクトの持つオブジェクト属性の初期化である．

MSC → ObCL コンバーターはシステム記述を生成しないので，生成された ObCL コードに付け加える必要がある．

以下に図 4.13 で定義したクラスに対するシステム記述例を示す．ここでは，クラス “X” のインスタンス “x1” の属性 “mode” を 0 で初期化している．

```
system Sys
  object x1(mode := 0):X
end
```

## 4.4 動的モデル作成プロセス

これまでに，オブジェクト間のメッセージシーケンスを記述するためにMSCを再定義し，この言語から動的モデルObTS/ObCLを作成する環境を構築した．

この環境を用いて以下のステップを繰り返すことにより，ユースケースモデルから動的モデルを一貫した方法で効率的に作成することができる．

1. ユースケースモデルの作成
   開発するシステム境界および，アクター，要求機能を定義
2. MSCの記述
   ユースケースのシナリオから，オブジェクトとメッセージシーケンス，それに付随する動作を抽出し，MSCで記述する
3. ObCLへの変換
   MSC → ObCLコンバーターによりObCLに変換
4. ObCLコードの修正
   システム記述の追加と状態の整理
5. ObML上でのテスト
   生成されたObCLをObML上でテスト
6. 最初のステップに戻る
   テストした結果を，ユースケースモデルにフィードバックする

この際，はじめからシステム全体の動的モデルを作成するのではなく，機能ごとに動的モデルを作成し，それらを合成することによりシステム全体の動的モデルを作成すべきである．これにより，各ユースケースの変更に対して柔軟に動的モデルを生成できる．

以下の節では，ステップ2のMSCの記述，ステップ4のObCLコードの修正，およびステップ5のテストについて詳しく説明する．

### 4.4.1 MSCの記述

ここでは，ユースケースモデルで定義されたユースケースを基に，MSCを記述する方法について述べる．

まず，ユースケースのシナリオからHMSCを記述することから始める．ここで，1つのHMSCはシステム，あるいはユースケースが持つメッセージシーケンスの大ざっぱな

構造を記述するものとみなせば理解しやすい．そのHMSCの構造があまりにも複雑な時はそれをいくつかのブロックに分けて，参照MSCを用いることにより抽象化する．一方，Basic MSCは，意味のあるなしに関わらず一連のメッセージを記述したものである．

HMSCを記述する際に大事なことは，シナリオから分岐やループを発見することである．ユースケースには機能の基本フローを記述する主シナリオとともに，代替処理やエラー処理などの代替フローを記述する副シナリオが記述されている．そのため，最低副シナリオの数だけ分岐やループがあるといえる．次に分岐やループの分岐条件を抽出し，また分岐や合流（ループは，分岐と合流によって記述される）を行なうポイントに何らかの状態を記述する．この時，状態と状態の間に，1つの参照MSCを記述する．参照MSCにより，Basic MSCかHMSCのどちらかが参照される．

HMSCでは，システムのユースケースを基にメッセージシーケンスの構造をトップダウンに記述していくとよい．root HMSCには，ユースケースモデル上でアクターと関連しているユースケースを参照 MSCを用いて記述する．そして，次のHMSCでは参照されているのユースケースのメッセージシーケンス構造を記述する．ここで，そのユースケースと関連しているユースケースがある場合，参照 MSCを用いて記述する．これを繰り返すことにより，システムを表すHMSCが記述できる．

次に，HMSC内のBasi MSCを記述するためにユースケースモデルからオブジェクトを抽出する．ユースケースモデルで定義されたユースケースからオブジェクトを抽出するために,「インタフェース」,「実体」，および「制御」の3つの抽象概念を用いたオブジェクトの識別法がIvar Jacobsonによって提案されている[4]．ここで,「インタフェース」とはシステムとアクターとのインタフェースを表すものであり，システムのインタフェースに属する振る舞いを持つ.「実体」とは保存，更新，および操作の対象となるもので，システム内のデータ要素として表される.「実体」は情報（データ）を取り扱う振る舞いを持つ．そして「制御」とはシステムのロジックを知っていて何らかの振る舞いを管理するものである．また，何らかの共有リソースに対するアクセスを制御するものも「制御」にあたる.「制御」は，システム内の上記2つの概念に属さない振る舞いを持つ．各ユースケースは，これら3種類のオブジェクトに完全に分割される．オブジェクトは当然，いくつかのユースケースに共通であってもよい．このことは，異なるユースケースで再利用できるオブジェクトであるため，望ましいことである．また，システムの変更に対しても再利用できることを示している．

オブジェクトを抽出した次は，各ユースケースのシナリオに定義された振る舞いをメッセージシーケンスとそれに伴う動作としてオブジェクトに振り分ける（図4.15）．ここ

でいう，動作とは入力イベントによってそのオブジェクトの属性が変化することである．ユースケースから振る舞いを振り分ける方法は，上記のオブジェクトの抽出方法を基に振り分ける．

HMSC 上の分岐条件，Basic MSC 上のアクション，出力イベントの各属性は，4.2.1 節のメッセージの部分で述べた「入力イベントのすべての属性は，存在しない場合を除いて，アクション部，分岐条件部，出力イベント部のうち少なくとも一ヶ所には出現しなくてはならない」という規則とシナリオにより，そのオブジェクトが直前に受け取った入力イベントの属性から詳細化していくとよい．これを各オブジェクトのイベントシーケンスの先頭から順次適用することにより，完全な HMSC と BasicMSC が導き出せる．

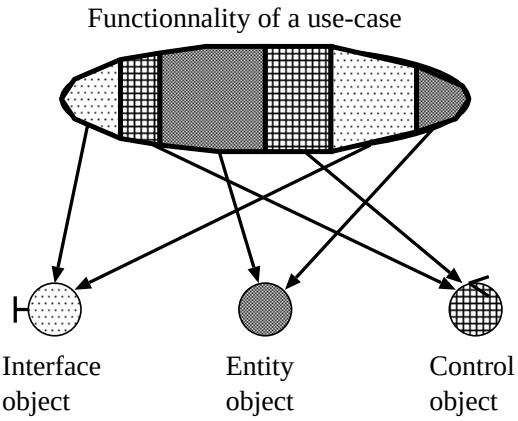



図 4.15: ユースケースの機能のオブジェクトへの割り当て

### 4.4.2 ObCL コードの修正

#### システム記述

以前に述べたように，MSC → ObCL コンバーターによって，生成された ObCL コードにはシステム記述は含まれない．そのため，ObCL コードの最後にシステム記述を追加する必要がある．

#### 状態遷移の整理

生成された ObCL コードには，冗長な状態遷移が存在する．システム全体の冗長な状態遷移は，各機能に対応した状態遷移の冗長な部分の和集合である．そのため，各ユースケースに対応した状態遷移から ObML を用いて冗長な部分を発見することにより，システム全体の状態遷移から冗長な部分を取り除くことができる．

各ユースケースに対応した状態遷移の中から冗長な部分を見つける際に，一般的に言えることは，初期状態と最終状態が異なる場合は同じ状態であると見なすことができるということである．これは，システムは一連の動作を繰り返し行なうという事実によるものである．

#### オブジェクト属性の整理

生成された ObCL コードに出現するオブジェクト属性の意味を定義することにより，この属性が必要かどうか判断を行なう．

### 4.4.3 テスト

動的モデル作成支援環境を用いてまず機能ごとに動的モデルを作成してテストを行ない，各機能に対応する状態遷移の冗長な部分を発見する．その後，システム全体の動的モデルを作成し，各機能のテスト結果を基に冗長な部分を取り除く．この方法により，冗長な状態遷移を効率よく整理することができる．

ObML シミュレータでは，ステップ実行によるテストとテストスクリプトによるテストを行なうことができる．ステップ実行によるテストは ObTS モデルでのステップ動作にしたがって進行する．シミュレータと対話的にテストをすることができるので，動作確認をしながら仕様を検討することができる．テストスクリプトによるテストは，システムコ

ンフィギュレーション (各ステップごとのオブジェクトの動作状態とイベントに関するスナップショットのリスト) を ML 言語スクリプトとして記述したものである．このスクリプトを用いることでシステムに対して連続的にテストをすることができる．

図 4.16 は ML 言語で記述されたテストスクリプトの構造を示している．テストスクリプトはイベント列生成器とイベント列解析器から成り，入力イベント列を発生させるイベント列生成器と，そのイベントによりシミュレータが出力した結果を解析するイベント列解析器はともに ML 言語で記述されている．

設計者はテストしたいシステムに対して，入力イベントとそれに対応する出力イベントをテストスクリプトに記述することで，設計者の意図したイベントごとの動作を柔軟にテストすることができる．

野村 [8] によると，このテストスクリプトをシステムから切り出だされた各機能に対して用意することにより，機能別にテストをすることが容易になり，さらに，各機能のテストスクリプトを再利用することで合成後のシステムに対するテストスクリプトも簡単に構築することができ，合成後のシステムに対しても柔軟なテストが行なうことが可能になることが確認されている．

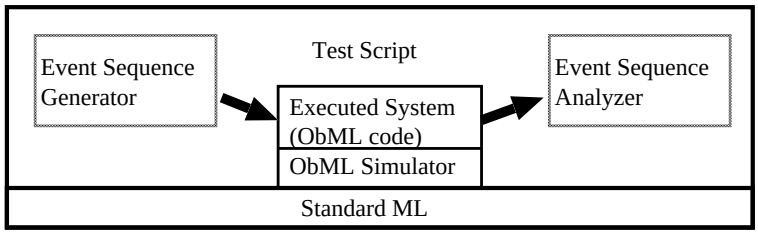



図 4.16: テストスクリプトの構造

# 第 5 章

# 動的モデル作成支援環境を用いた例題

ここでは，前章で提案した動的モデル作成プロセスを用いて，ATM システムの開発を行なうことにより，動的モデル作成支援環境の有効性を示すことにする．

ここでは前章のプロセスに従い，まず，各機能ごとに動的モデルを生成し，次に，システム全体の動的モデルを生成することにする．

## 5.1 ATM の仕様

ATM システムは，以下の機能を持つ．

- 払い戻し: withdraw
- 預入れ: deposit
- 残高照会: balance_inquiry
- メンテナンス（現金の補充）: maintenance

このシステムは，1 つの ATM と 1 つの銀行からなっており，銀行は，複数の口座をもっているとする．

このシステムのユースケースモデルを付録 B.1 に示す．

## 5.2 オブジェクトとメッセージシーケンスの抽出

前節で作成したユースケースモデルを基に，4.4.1 節の方法を用いて，オブジェクトとメッセージシーケンスの抽出を行なう.

#### オブジェクトの抽出

ユースケースモデルより以下のオブジェクトを抽出した.

- ATM オブジェクト (ATM)
- 口座オブジェクト (BANK)

#### メッセージの抽出

ユースケースモデルより各オブジェクトのメッセージシーケンスを抽出した．付録 B.2 に ATM システムで使われるメッセージを各機能ごとに示す.

## 5.3 「払い戻し」に対応したMSCの作成

前節より得られた ATM システムのオブジェクトとその間のメッセージシーケンスから「払い戻し」に対応した MSC を記述する．まず，root の HMSC にメッセージシーケンスの大ざっぱな流れを記述することから始める．この例では，root HMSC は「サービス選択」に対応した Basic MSC と「払い戻し」に対応した HMSC を参照している．この「払い戻し」に対応した HMSC では,「払い戻し」の機能を実現するためのメッセージシーケンス構造を記述する．この時，その構造があまりにも複雑な場合は，いくつかのブロックに分割して参照 MSC を用いて抽象化するとよい．この MSC を付録 B.3 に示す.

### 5.3.1 各属性の抽出

HMSC 内の遷移条件部の属性，Basic MSC 内のオブジェクト属性とアクション部の属性は，4.4.1 節に述べたように，そのオブジェクトが直前に受信した入力イベントの属性を基に，MSC のメッセージシーケンスの先頭から順次，求めていきます.

図B.4では，カード挿入を意味するイベント “cardin” によって，口座番号 “account_num” を ATM オブジェクトが受け取っている．この場合，シナリオからこの属性は次の出力イベント属性に渡されないことがわかるので，オブジェクトの属性に代入する形で保持する必要がある．ここでは，属性 “val” を使って値を保持している．

また，図B.9では，図B.8の口座残高確認と残高増減を意味するイベント “checkamount” によって，口座番号 “num” と金額 “-val” を BANK オブジェクトが受け取っている．ここで，シナリオより口座番号 “num” と一致する口座の残高から金額 “-val” だけ足した時の残高が 0 以上になるかどうかでシナリオが分岐するので，これらの属性を用いてこの Basic MSC を参照している HMSC “WITHDRAW” に分岐条件を記述する．また，増減を行なった後の口座残高が 0 以上の場合，実際に口座残高を増減する動作を記述するために，口座残高を保持するオブジェクト属性（ここでは，“account_balance”）が必要なことがわかる．

図B.6では，パスワード入力を意味するイベント “passin” によって，パスワード “password” を ATM オブジェクトが受け取っている．次に，ATM オブジェクトはシナリオからこの属性と以前に入力された口座番号 “num” を用いて，認証要求を行なうので，出力イベント “checkaccount” は，パスワード “password” と口座番号 “num” の 2 つの属性をもつ．

## 5.4 「払い戻し」に対応したObCLへの変換

付録 B.4 に，MSC → ObCL コンバーターによって得られた「払い戻し」に対応した ObCL コードを示す．

## 5.5 ObCL コードの修正

ここでは，4.4.2 節で示したように，生成された ObCL コードを修正する．

### 5.5.1 システム記述

ATM クラスからオブジェクト “atm1” を，BANK クラスからオブジェクト “acc1” を定義する．“atm1” オブジェクトでは，ATM 残高を 100 で初期化し，“acc1” オブジェクトでは，それぞれパスワード入力回数，口座残高，口座番号，口座暗証番号を初期化する．以下に追加されたシステム記述を示す．

```
system s
  object atm1(
    ATM_balance := 100
    ):ATM
  object acc1(
    count := 0;
    account_balance := 100;
    account_num     := 100;
    pass            := "12345"
    ):BANK
end
```

### 5.5.2 状態と遷移の整理

BANK クラスは繰り返し同じ動作を行なう必要があるため，初期状態 “Before_Passin” と最終状態 “ATM_Idle” は同であるとみなすことができる．5.6 節で詳しく述べるが，ObML から状態 “ATM_Idle” と状態 “Checked_ATM_Balance” は同じ状態でなければならないことがわかる．次のページに変更を行なった BANK クラスの ObCL コードを示す．

### 5.5.3 オブジェクト属性の整理

ATM クラスの属性 “val”，“num”，“ATM_balance” は，それぞれ金額の一時保持用属性，口座番号の一時保持用属性，ATM 残高である．また，BANK クラスの属性 “count”，“account_balance”，“account_num”，“pass” は，それぞれパスワード入力回数，口座残高，口座番号，口座暗証番号である．

BANK クラスの修正 (ObCL コード)

```
--- BANK Class
Class BANK
  field F2:FIELD2
  attribute count:Int
  attribute account_balance:Int
  attribute account_num:Int
  attribute pass:String
  state ATM_Idle
  transition
  start is
    source init
    destination ATM_Idle
  end
  t10 is
    source ATM_Idle
    input F2.checkaccount
    when ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 = pass )
    do  count := 0
    destination ATM_Idle
    output F2.ok
  end
  t11 is
    source ATM_Idle
    input F2.checkaccount
    when ( ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 <> pass ) ) and ( ( count < 2 ) )
    do  count := count + 1
    destination ATM_Idle
    output F2.ng
  end
  t12 is
    source ATM_Idle
    input F2.checkaccount
    when ( ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 <> pass ) ) and ( count = 2 )
    do  count := 0
    destination ATM_Idle
    output F2.ng2
  end
  t13 is
    source ATM_Idle
    input F2.checkamount
    when ( F2.checkamount.var1 = account_num )
         and ( ( account_balance + ( F2.checkamount.var2 ) > 0 )
         or ( account_balance + ( F2.checkamount.var2 ) = 0 ) )
    do  account_balance := account_balance + ( F2.checkamount.var2 )
    destination ATM_Idle
    output F2.ok
  end
  t14 is
    source ATM_Idle
    input F2.checkamount
    when ( F2.checkamount.var1 = account_num )
         and ( account_balance + ( F2.checkamount.var2 ) < 0 )
    do  count := 0
    destination ATM_Idle
    output F2.ng2
  end
end
```

## 5.6 「払い戻し」に対応したObCLコードのテスト

ObML を用いてステップ実行によるテストを行ない動的モデルを修正する．ここでは，B.4 節の「払い戻し」に対応する ObCL コードから，BANK クラスが繰り返し動作できるように初期状態 “Befor_Passin” を最終状態 “ATM_Idle” によって修正した段階でテストを行なった．

いま，ATM クラスのオブジェクト “atm1” の ATM 残高が 100，BANK クラスのオブジェクト “acc1” の口座番号が 100，パスワードが”12345”，そして口座残高が 100 であるとする時，システムに次のメッセージシーケンスを入力した場合，このシステムは止まってしまう．

```
>F1.cardin(var1=Int 100)                        - 口座番号 100 のカード挿入
>F1.request_service(str1=String "withdraw") - 「払い戻し」を選択
>F1.passin(str1=String "12345")                 - 暗証番号"12345"を入力
>F1.amountin(var1=Int 200)                      - 金額を 200 に設定
>F1.confirm                                     - 金額確認
>F1.cardin(var1=Int 100)                        - 口座番号 100 のカード挿入
>F1.request_service(str1=String "withdraw") - 「払い戻し」を選択
>F1.passin(str1=String "12345")                 - 暗証番号"12345"を入力
```

ここでシステムが停止したときの，各オブジェクトの状態は ATM: Checked_Account，BANK: Checked_ATM_Balance で，この時 BANK オブジェクトが遷移していないことがわかる．よって，これより状態 “ATM_Idle” と状態 “Checked_ATM_Balance” は同じ状態でなければならないことがわかる．

付録 B.5 に,「払い戻し」に対応した動的モデルをテストするために与えたいろいろな入力イベント列を示す．

## 5.7 ATM システムの動的モデル

前節では，ユースケース「払い戻し」に対応する動的モデルを作成した．それと同じプロセスを用いて残りのユースケース「預入れ」,「残高照会」，そして「メンテナンス」に対応する動的モデルを作成し，テストすることにより，各動的モデルから冗長な部分を発見する．ATM システムの動的モデルを作成するためには，HMSC 上で各機能を表す MSC を合成することにより行なう．

### 5.7.1 ATM システムの MSC の作成

付録 B.6 に，ATM システム全体の MSC を示す．ただし,「払い戻し」に対応する MSC は，5.3 節で用いたものを再び使用する．

### 5.7.2 ATM システム ObCL への変換

付録 B.7 に，MSC → ObCL コンバーターによって得られた ATM システム全体の ObCL コードを示す．

### 5.7.3 ObCL コードの修正

BANK クラスでは「払い戻し」で行なったような修正に加えて，BANK クラスへの入力イベント “checkaccount” と “checkamount” に対する状態遷移が各機能ごとに繰り返し定義されているため，整理する必要がある．BANK クラスの修正結果を次ページに示す．なお，システム記述は「払い戻し」の時と同様に記述する．

```
--- BANK Class
Class BANK
  field F3:FIELD3
  attribute count:Int
  attribute account_balance:Int
  attribute account_num:Int
  attribute pass:String
  state ATM_Idle
  transition
  start is
    source init
    destination ATM_Idle
  end
  t10 is
    source ATM_Idle
    input F3.checkaccount
    when ( F3.checkaccount.var1 = account_num )
         and ( F3.checkaccount.str1 = pass )
    do  count := 0
    destination ATM_Idle
    output F3.ok
  end
  t11 is
    source ATM_Idle
    input F3.checkaccount
    when ( ( F3.checkaccount.var1 = account_num )
         and ( F3.checkaccount.str1 <> pass ) ) and ( count < 2 )
    do  count := count + 1
    destination ATM_Idle
    output F3.ng
  end
  t12 is
    source ATM_Idle
    input F3.checkaccount
    when ( ( F3.checkaccount.var1 = account_num )
         and ( F3.checkaccount.str1 <> pass ) ) and ( count = 2 )
    do  count := 0
    destination ATM_Idle
    output F3.ng2
  end
  t13 is
    source ATM_Idle
    input F3.checkamount
    when ( F3.checkamount.var1 = account_num )
         and ( ( account_balance + ( F3.checkamount.var2 ) > 0 )
         or ( account_balance + ( F3.checkamount.var2 ) = 0 ) )
    do  account_balance := account_balance + ( F3.checkamount.var2 )
    destination ATM_Idle
    output F3.ok
  end
  t14 is
    source ATM_Idle
    input F3.checkamount
    when ( F3.checkamount.var1 = account_num )
         and ( account_balance + ( F3.checkamount.var2 ) < 0 )
    do  count := 0
    destination ATM_Idle
    output F3.ng2
  end
  t15 is
    source ATM_Idle
    input F3.checkbalance
    when F3.checkbalance.var1 = account_num
    do  F3.balanceout.var1 := account_balance
    destination ATM_Idle
    output F3.balanceout
  end
end
```

## 5.8 事例研究のまとめ

ATM システムの動的モデルを，本研究で構築した動的モデル作成支援環境を用いてユースケースモデルから作成した．

この環境を用いると動的モデルを作成する上で曖昧な記述を ObCL へ変換する過程や ObML 上での実行/テストの過程で洗練することができ有効であることがわかる．また，MSC はグラフィカル形式による可読性，テキスト形式による記述の形式化，HMSC による様々な構造のメッセージシーケンスに対する記述性，そして参照 MSC によって繰り返し MSC を利用することで再利用性に優れている言語であることがわかる．また，あるユースケースの変更に対して，それにより影響を受けた MSC のみ変更すればよいため，ユースケースモデルの変更に対してこの環境は柔軟であることがわかった．

一方，生成される動的モデルには冗長な状態遷移が含まれてしまうという問題が起る．前にも述べたがシステム全体に対応した動的モデル中の状態遷移の冗長な部分は，各機能に対応した状態遷移の冗長な部分の和集合になっている．そのため，各機能ごとに動的モデルを作成しテストすることにより，システム全体に対応した動的モデルの中から冗長な部分を発見することができる．

# 第 6 章

# まとめ

本研究では，ユースケースのシナリオに基づいた動的モデル作成支援環境を構築した．

ユースケースのシナリオに含まれるオブジェクトとそれらの間のメッセージシーケンスを形式的に記述するために，通信分野で広く使われている MSC を新たに定義して使用した．これにより，メッセージシーケンスの可読性の向上と計算機によって扱うことを可能としている．

また，動的モデルとして再利用性と可読性に優れている ObTS/ObCL を仮定し，MSC 言語から仕様記述言語 ObCL に変換する MSC → ObCL コンバーターを構築した．

MSC → ObCL コンバーターにより，ユースケースの仕様に沿った状態遷移図を効率的に生成することができる．

また，ObCL で記述することにより ObML シミュレータを用いて生成された動的モデルを実行/テストすることができるため，分析/設計の早い段階でユースケースモデルを洗練することができる．

最後に例題で，動的モデル作成支援環境を用いて動的モデルを作成することにより柔軟で効率的な開発が行なえることを示した．

# 第 7 章

# 今後の課題

## 7.1 MSC でサポートしなかった事柄について

本研究で用いた MSC は動的モデルを作成するという目的のため，標準の MSC からサポートしなかった点がいくつかある．その中から，今後取り扱う必要があるかもしれない事柄について述べる．

### タイマー

開発するシステムによっては，同期メッセージばかりではなく，非同期メッセージも用いられる．また，メッセージは途中で失われることも考慮する必要があるかも知れない．これらのイベントはタイマーによって管理される．ObCL は同期メッセージによってイベントをやり取りするために時間を記述することは難しいが，ObCL に時間についての遷移条件を記述すれば表現することができる．

### 状態

Basic MSC 内ではメッセージシーケンスのみに着目するため，状態を使用しなかった．しかし，生成される状態遷移の冗長性を抑えるために，状態がわかるところはインスタンスに直接記述できるようにすればさらに詳細な動的モデルが生成できるであろう．

#### インライン表記と参照 MSC

本研究では HMSC に制御構造を持たせることで，メッセージシーケンスの構造を記述したが，Basic MSC 内に制御構造，状態，そして参照 MSC を用いることにより，各オブジェクトのイベント列に着目した構造を記述することができる．ただし，これを使いすぎると 1 つの Basic MSC がたいへん複雑になってしまう．このような記述に似た記述言語として David Harel らが提案している Live Sequence Chart(LSC) がある [6].

## 7.2 状態遷移の冗長性について

MSC → ObCL コンバーターで生成された動的モデルには，冗長な状態遷移を多く含む．そのため，簡単な状態遷移図では，システムの性質から初期状態と最終状態は同じであると判断したり，明らかに同じ処理を行なっているので統合できると判断できたとしても，複雑な状態遷移図になるとその部分を見抜くことが困難になる．

システム全体に対応する動的モデル内の冗長な部分は，各機能に対応する動的モデルの冗長な部分の和集合になっている．そのため，各機能ごとに動的モデルを作成し，テストすることにより冗長な部分を見つけることによって，システム全体の冗長な部分を取り除くことができる．

## 7.3 ObML との連携について

MSC → ObCL コンバーターで生成された動的モデルを，ObML を用いて実行/テストすることによりユースケースモデルを洗練することができる．

ObML は，ObCL コンバーターによって生成された ObML コードを読込んで，それと共に外部からの入力イベントを与えてやることで実行/テストを行なっている．

そのため，MSC → ObCL コンバーターで ObCL に変換する際にそれらのイベント列を生成してやれば，効率よく実行/テストを行なうことができる．

MSC → ObCL コンバーターは，各 MSC から外部からの入力イベントを抽出できるのでこれを組み合わせることで効率的なテストが行なえる．図 7.1 は，MSC → ObCL コンバーターから得られたイベント “event1”，“event2”，“event3”，“event4”，“event5” の中からイベント “event2”，“event3”，“event5” で構成されるイベント列を用いて，生成された動的モデルのテストを行なう様子を示している．

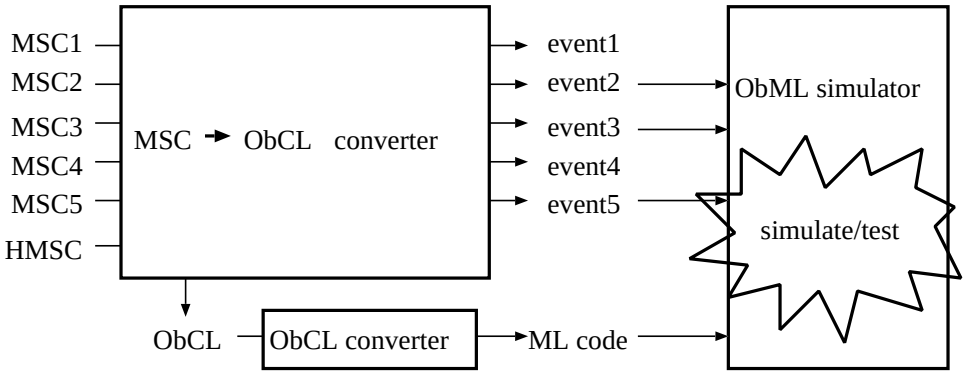



図 7.1: MSC → ObCL コンバーターを用いたテスト

# 謝辞

本研究を行うに当たり，有益な御指導，助言および援助を頂きました片山 卓也教授，伊藤 恵助手に深く感謝致します.

また，本論文をまとめるに当たって御協力いただいたソフトウェア基礎講座のメンバーの皆様に深く感謝致します.

# 参考文献


[1] Z120 (1996). *Message Sequence Chart (MSC)*. ITU-T, Geneva, October 1996.

[2] Schneider G. and Winters J.P. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1998. 邦訳 「ユースケースの適用:実践ガイド」 2000, ピアソン・エデュケーション.

[3] Eriksson H.E. and Penker M. *UML Toolkit*. John Wiley & Sons, 1998. 邦訳 「UML ガイドブック」 1998, トッパン.

[4] Jacobson I. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992. 邦訳 「オブジェクト指向ソフトウェア工学 OOSE use-case によるアプローチ」 1995, トッパン.

[5] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorenson W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991. 邦訳 「オブジェクト指向方法論 OMT モデル化と設計」 1992, トッパン.

[6] Damm W. and Harel D. Lscs: Breathing life into message sequence charts. Technical report, The Weizmann Institute of Science, 1998.

[7] 伊藤恵, 片山卓也. オブジェクト指向方法論のための動的モデル ObTS. 修士論文, 北陸先端科学技術大学院大学, 1995.

[8] 野村昌男, 片山卓也. 組み込みシステム設計のための obts に基づく記述支援環境に関する研究. 修士論文, 北陸先端科学技術大学院大学, 2000.

[9] 久保秋真, 片山卓也. 動的モデル ObTS の大規模組み込みシステム記述に関する研究. 修士論文, 北陸先端科学技術大学院大学, 1998.

# 付 録 A

# MSCのtextual文法

この章では，動的モデル作成支援環境で用いる MSC の textual 文法を示す.

## A.1 一般的な規則

### A.1.1 字句規則

字句規則により字句を定義する．字句は textual 構文の終端記号である.

```
<lexical unit> ::=  <word>
                  | <special>
                  | <note>
                  | <keyword>

<word> ::= <alphanumeric>*

<alphanumeric> ::= <letter>|<decimal digit>

<letter> ::= [a-zA-Z]+

<decimal digit> ::= [0-9]+

<special> ::= <parenthesis>|<end>|<full stop>|<comma>|<colon>

<parenthesis> ::= <left parenthesis>|<right parenthesis>

<left parenthesis> ::= "("

<right parenthesis> ::= ")"

<end> ::= ";"

<full stop> ::= "."

<comma> ::= ","

<colon> ::= ":"
```

```
<note> ::= note:<text>

<text> ::=  {<alphanumeric>
          | <other character>
          | <special>
          | <underline>
          | <space>
          | <apostrophe>}*

<other character> ::= <operator>|<boolean operator>

<underline> ::= "_"

<apostrophe> ::= "'"

<name> ::= -?<word>(<underline><word>)*

<string> ::= \"<text>\"

<operator> ::= "+" |"-" |"*" |"/" |"="

<boolean operator> ::= "and" |"or" |"not" |">" |"<" |"==" |"!="

<keyword> ::=  action
             | condition
             | else if
             | end
             | endinstance
             | endmsc
             | expr
             | from
             | if
             | in
             | inst
             | instance
             | msc
             | out
             | seq
             | to
             | note
             | Int
             | String
```

### A.1.2 コメント

"//" から行の終りまでをコメントとする.

```
<comment> ::= //<text>
```

## A.2 Basic MSC

### A.2.1 Message Sequence Chart

```
<message sequence chart> ::=
        msc<msc head>\(<msc body>|expr<msc expression>\)endmsc<end>

<msc head> ::=
        <msc name><end><msc interface>

<msc name> ::=
        <name>

<msc interface> ::=
        <msc inst interface>

<msc inst interface> ::=
        inst<instance list><end>

<instance list> ::=
        <instance name>(,<instance list>)*

<instance name> ::=
        ?(USER)<name>

<msc body> ::=
        <msc statement>*

<msc statement> ::=
        <old instance head statement><instance event list>

<old instance head statement> ::=
        instance<instance name><end>

<instance event list> ::=
        (<instance event><end>)+

<instance event> ::=
        <orderable event><non-orderable event>

<orderable event> ::=
        <message event>|<action>

<non-orderable event> ::=
        <instance end statement>
```

### A.2.2 インスタンス

```
<instance end statement> ::=
        endinstance
```

### A.2.3 メッセージ

```
<message event> ::=
        <message output>|<message input>
```

```
<message output> ::=
        out<msg identification>to<input address>

<message input> ::=
        in<msg identification>from<output address>

<msg identification> ::=
        <message name>\((<parameter list>|<string>)*\)

<message name> ::=
        <name>

<parameter list> ::=
        <parameter name>:<type>(,<parameter list>)*

<parameter name> ::=
        <name>

<type> ::=
        Int|String

<input address> ::=
        <instance name>

<output adderss> ::=
        <instance name>
```

### A.2.4 アクション

```
<action> ::=
        action(<assignment expression>|<note>)

<assignment expression> ::=
        <object attribute name>=(<name>|<operator>
        |<parenthesis>|<decimal digit>|<string>)*

<object attribute name> ::=
        <name>
```

## A.3 HMSC

```
<msc expression> ::=
        <start><node expression>*

<start> ::=
        <label name><end>

<label name> ::=
        L<decimal digit>

<node expression> ::=
        <label name>:(<node>seq\(<label name>\)|end)<end>
        |<label name>:<if-else>
```

```
<node> ::=
<msc name>|condition<condition name>

<if-else> ::=
        if\(<boolean expression>\)seq\(<label name>\)<end>
        (else if\(<boolean expression>\)seq\(<label name>\)<end>)*

<boolean expression> ::= <note>
        |(<name>|<boolean operator>|<parenthesis>
        |<operator>|<decimal digit>|<string>)*
```

# 付 録 B

# ATMシステム開発事例

## B.1　ATMシステムのユースケース

ATM システム

ATM は，以下の機能を持つ.

- 払い戻し: withdraw
- 預入れ: deposit
- 残高照会: balance_inquiry
- メンテナンス（現金の補充）: maintenance

このシステムは，1 つの ATM と 1 つの銀行からなっており，銀行は，複数の口座をもっている.

リスク要因

複数の ATM と複数の銀行がある場合の通信をどうするか.

システムレベルのユースケース

- アクター: 顧客 (USER)，従業員 (USER_EMPLOYEE)
- ユースケース:
  - サービスを選択する Select_Service_usecase
  - 払い戻し Withdraw_usecase
  - 預入れ Deposit_usecase
  - 残高照会 Balance_Inquiry_usecase
  - メンテナンス Maintenance_usecase

このシステムのユースケースモデルを図 B.1 に示す.

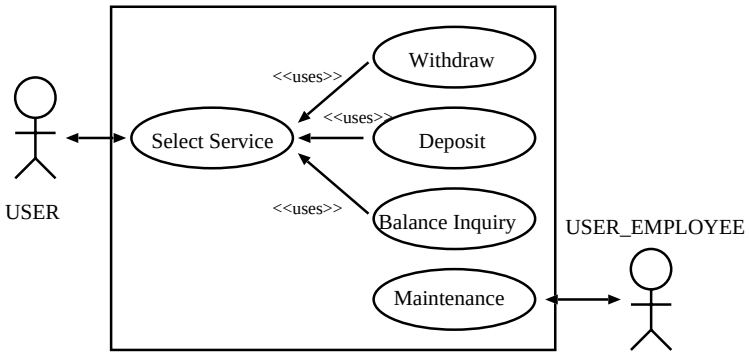



図 B.1: ATM システムのユースケース

## サービスを選択する Select_Service_usecase

このユースケースは，ユーザが ATM にサービスを選択する際のプロセスを記述する．

## アクター

顧客 (USER)

## 利用する (uses) ユースケース

- 払い戻し Withdraw_usecase
- 預入れ Deposit_usecase
- 残高照会 Balance_Inquiry_usecase

## イベントフロー

基本パス

1. ユーザがカードを挿入すると，このユースケースが始まる．
2. ATM は「サービスを選択してください」と表示する．
3. ユーザは，サービスを 1 つ選択する．
4. if ユーザが「払い戻し」を選択した場合，uses「払い戻し」
5. else if ユーザが「預入れ」を選択した場合，uses「預入れ」
6. else if ユーザが「残高照会」を選択した場合，uses「残高照会」
7. このユースケースが終了する．

## 払い戻し Withdraw_usecase

このユースケースは，ATM の払い戻しのプロセスを記述する．

## アクター

顧客 (USER)

## イベントフロー

基本パス

1. ユーザが「払い戻し」を選択すると，このユースケースが始まる．
2. ATM は「暗証番号を入力してください」と表示する．
3. ユーザは，暗証番号を入力する．
4. ATM は入力された情報（口座番号と暗証番号）を，口座に照合する．
5. ATM は「払い戻し金額を入力してください」と表示する．
6. ユーザは，払い戻し金額を入力する．
7. ATM は金額と「金額を確認してください」を表示する．
8. ユーザは，確認を選択する．
9. ATM は入力された情報（払い戻し金額）と ATM の残高とを比較する．
10. ATM は入力された情報（口座番号と払い戻し金額）を，口座に照合する．
11. ATM はカードとお金を出力して，このユースケースが終了する．

副シナリオ

- 暗証番号が無効である場合:
  もう一度，暗証番号を入力させる．
  3 回間違えると,「はじめからやり直してください」と表示し，このユースケースが終了する．

- 払い戻し金額より ATM 残高が少ない場合:
  「はじめからやり直してください」と表示し，このユースケースは終了する．

- 払い戻し金額より口座残高が少ない場合: 「はじめからやり直してください」と表示し，このユースケースは終了する．

## 預入れ Deposit_usecase

このユースケースは，ATM の預入れのプロセスを記述する．

## アクター

顧客 (USER)

## イベントフロー

基本パス

1. ユーザが「預入れ」を選択すると，このユースケースが始まる．
2. ATM は「紙幣を投入してください」と表示する．
3. ユーザは，紙幣を投入する．
4. ATM は金額と「金額を確認してください」を表示する．
5. ユーザは，確認を選択する．
6. ATM は入力された情報（口座番号と預入れ金額）を，口座に照合する．
7. ATM はカードを出力して，このユースケースが終了する．

副シナリオ

- 暗証番号が無効である場合:
  もう一度，暗証番号を入力させる．
  3 回間違えると,「はじめからやり直してください」と表示し，このユースケースが終了する．

## 残高照会 Balance_Inquiry_usecase

このユースケースは，ATM の残高照会のプロセスを記述する．

## アクター

顧客 (USER)

## イベントフロー

基本パス

1. ユーザが「残高照会」を選択すると，このユースケースが始まる．
2. ATM は「暗証番号を入力してください」と表示する．
3. ユーザは，暗証番号を入力する．
4. ATM は入力された情報（口座番号と暗証番号）を，口座に照合する．
5. ATM は口座残高を，口座に照合する．
6. ATM は金額を表示し，カードを出力して，このユースケースが終了する．

副シナリオ

- 暗証番号が無効である場合:
  もう一度，暗証番号を入力させる．
  3 回間違えると,「はじめからやり直してください」と表示し，このユースケースが終了する．

### メンテナンス Maintenance_usecase

このユースケースは，ATM の現金補充のプロセスを記述する．

### アクター

従業員 (USER_EMPLOYEE)

### イベントフロー

基本パス

1. 従業員がメンテナンスアプリケーションを実行すると，このユースケースが始まる．
2. システムは，現在の ATM 残高を表示する．
3. 従業員が現金を補充する．
4. システムは，現在の ATM 残高を表示して，このユースケースが終了する．

## B.2 オブジェクトとメッセージシーケンスの抽出

| メッセージ名 | 方向 | 属性 | 意味 |
|---|---|---|---|
| cardin | USER → ATM | 口座番号 | カードの挿入 |
| request_service | USER → ATM | サービス名 | 払い戻し処理開始 |
| msg | ATM → USER | メッセージ | メッセージの表示 |

表 B.1: サービスを選択するユースケース内のメッセージ表

| メッセージ名 | 方向 | 属性 | 意味 |
|---|---|---|---|
| passin | USER → ATM | 暗証番号 | 暗証番号の入力 |
| amountin | USER → ATM | 金額 | 金額の入力 |
| confirm | USER → ATM | なし | 金額の確認 |
| moneyout | ATM → USER | 金額 | 紙幣の支払い |
| cardout | ATM → USER | 口座番号 | カードの排出 |
| msg | ATM → USER | メッセージ | メッセージの表示 |
| checkaccount | ATM → BANK | 口座番号，暗証番号 | 認証要求 |
| checkamount | ATM → BANK | 口座番号，金額 | 口座残高の確認/増減要求 |
| ok | BANK → ATM | なし | 正常処理応答 |
| ng | BANK → ATM | なし | 非正常処理応答 1 |
| ng2 | BANK → ATM | なし | 非正常処理応答 2 |

表 B.2: 払い戻しユースケース内のメッセージ表

| メッセージ名 | 方向 | 属性 | 意味 |
|---|---|---|---|
| moneyin | USER → ATM | 金額 | 金額の投入 |
| confirm | USER → ATM | なし | 金額の確認 |
| cardout | ATM → USER | 口座番号 | カードの排出 |
| msg | ATM → USER | メッセージ | メッセージの表示 |
| checkamount | ATM → BANK | 口座番号，金額 | 口座残高の確認/増減要求 |
| ok | BANK → ATM | なし | 正常処理応答 |

表 B.3: 預入れユースケース内のメッセージ表

| メッセージ名 | 方向 | 属性 | 意味 |
|---|---|---|---|
| passin | USER → ATM | 暗証番号 | 暗証番号の入力 |
| cardout | ATM → USER | 口座番号 | カードの排出 |
| msg | ATM → USER | メッセージ | メッセージの表示 |
| msg2 | ATM → USER | 金額 | 残高の表示 |
| checkaccount | ATM → BANK | 口座番号，暗証番号 | 認証要求 |
| checkbalance | ATM → BANK | 口座番号 | 口座残高の確認 |
| balanceout | BANK → ATM | 金額 | 残高照会応答 |

表 B.4: 残高照会ユースケース内のメッセージ表

| メッセージ名 | 方向 | 属性 | 意味 |
|---|---|---|---|
| maintenance | USER_EMPLOYEE → ATM | なし | メンテナンス処理開始 |
| msg2 | ATM → USER_EMPLOYEE | 金額 | 残高の表示 |
| msg | ATM → USER_EMPLOYEE | メッセージ | メッセージの表示 |
| money_supply | USER_EMPLOYEE → ATM | 金額 | 現金の補充 |

表 B.5: メンテナンスユースケース内のメッセージ表

## B.3 「払い戻し」に対応したMSC

### B.3.1 グラフィカル表記

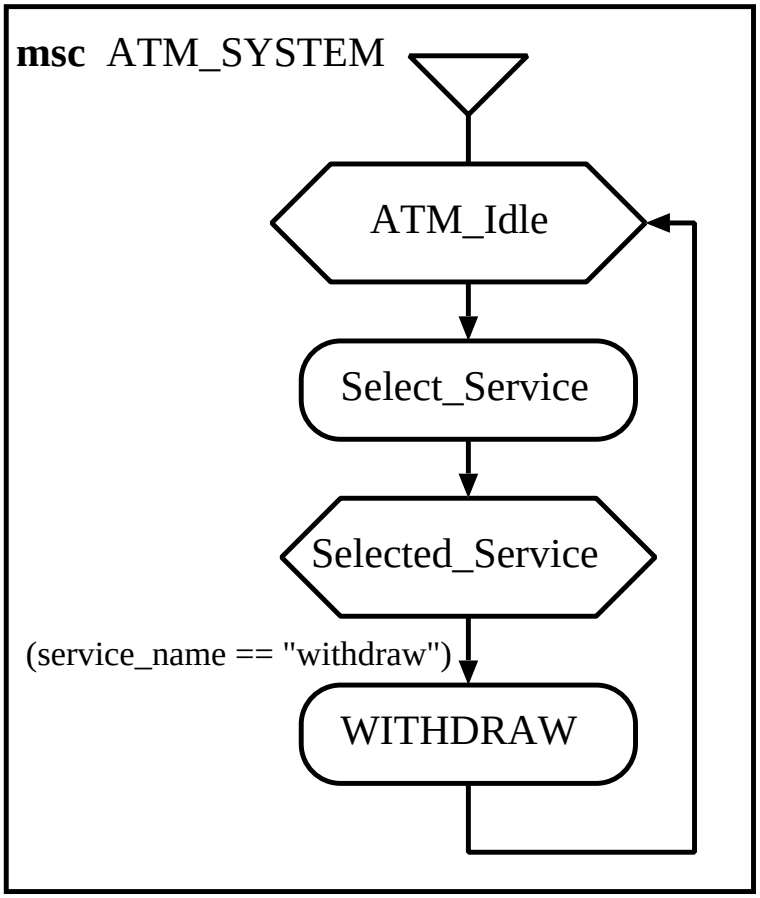



図 B.2: msc ATM_SYSTEM

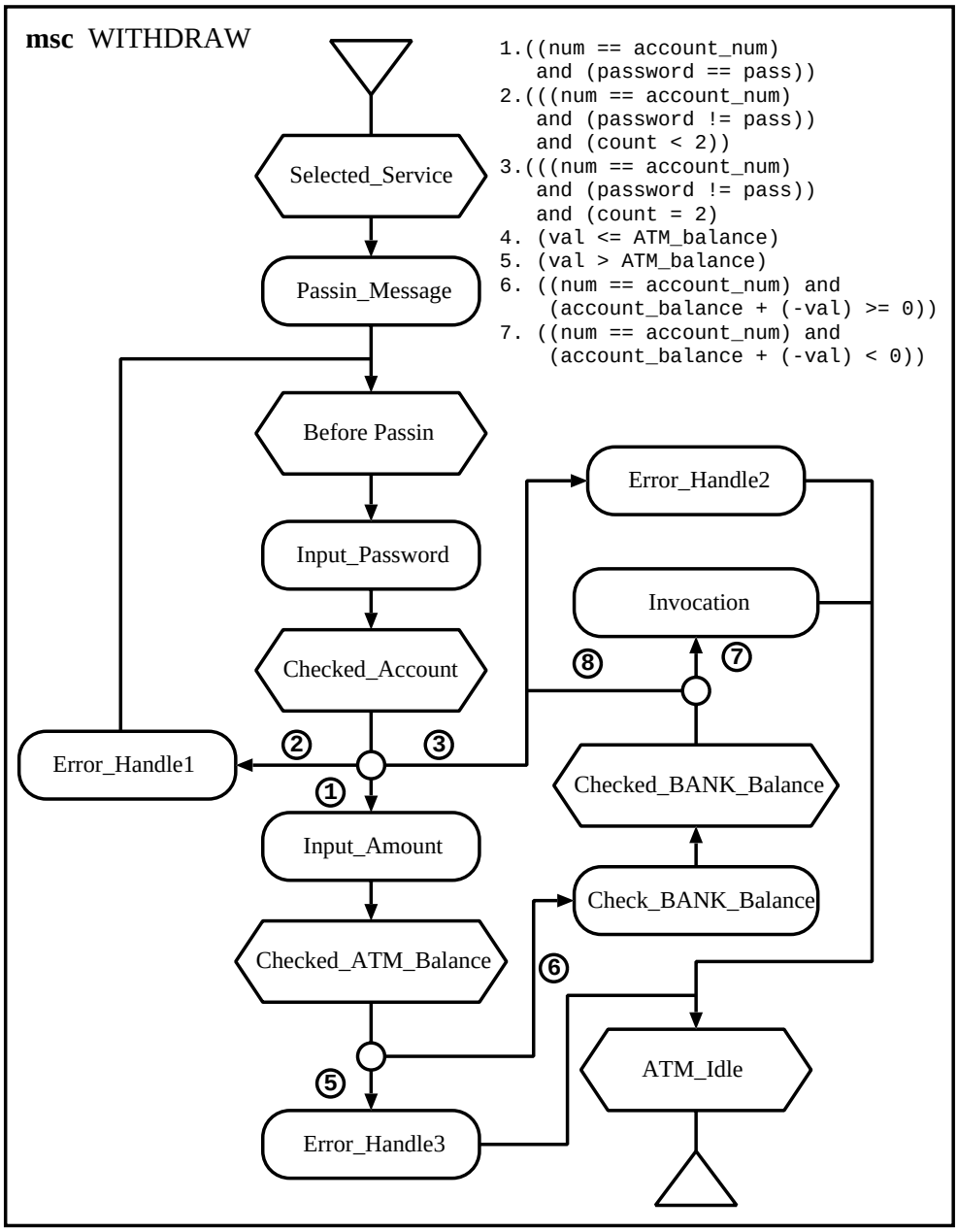



図 B.3: msc WITHDRAW

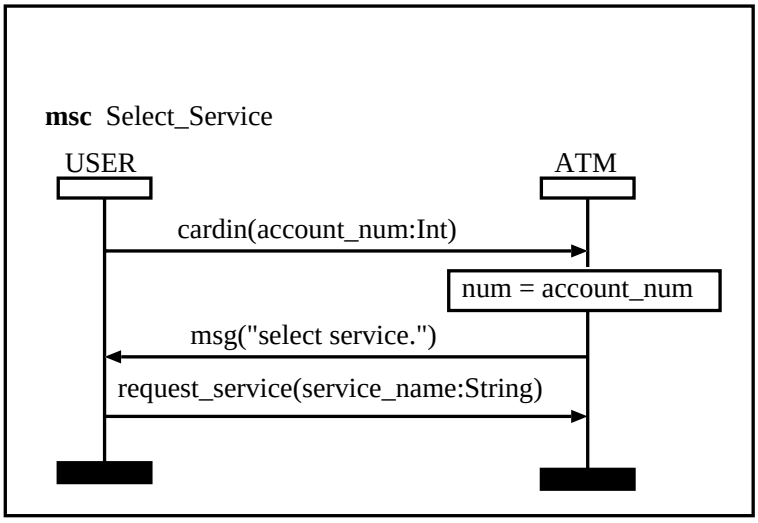



図 B.4: msc Select_Service

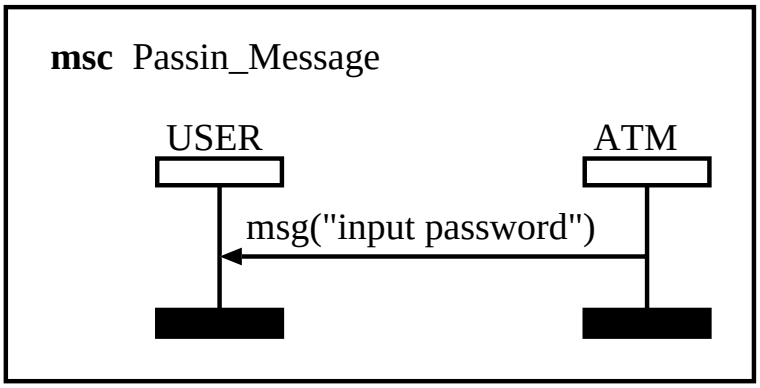



図 B.5: msc Passin_Message

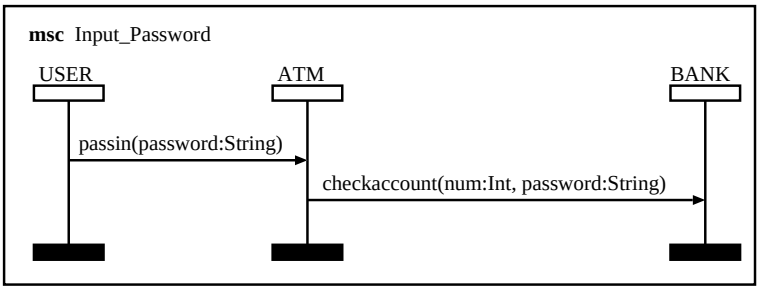



図 B.6: msc Input_Password

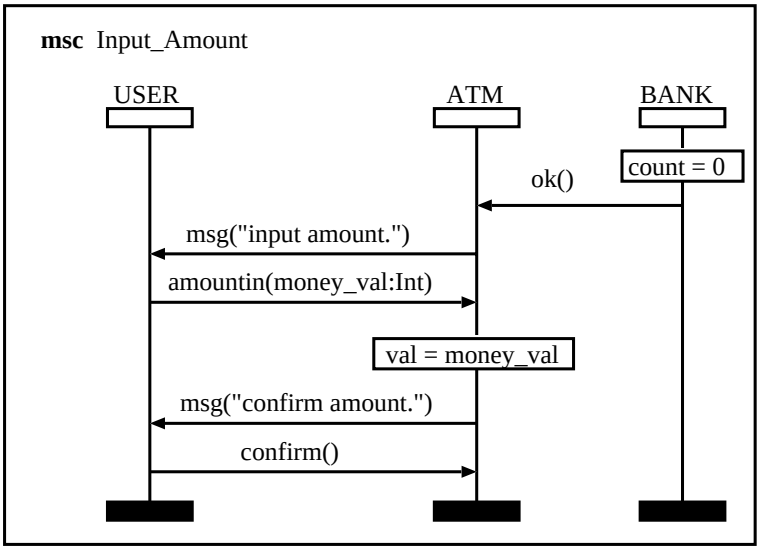



図 B.7: msc Input_Amount

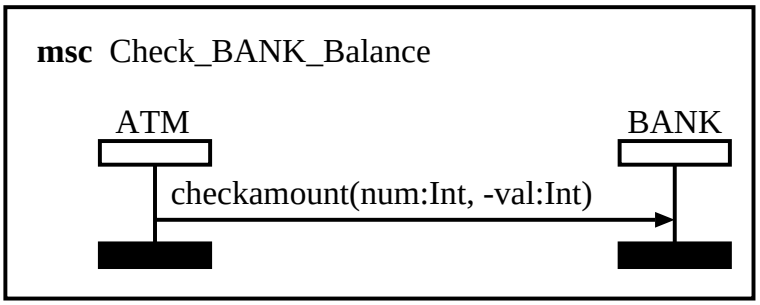



図 B.8: msc Check_BANK_Balance

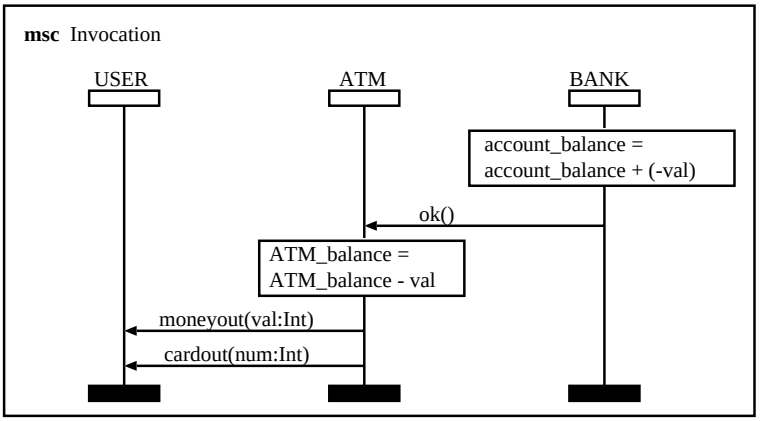



図 B.9: msc Invocation

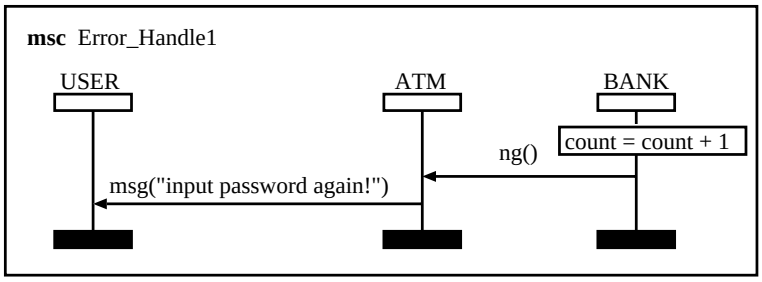



図 B.10: msc Error_Handle1

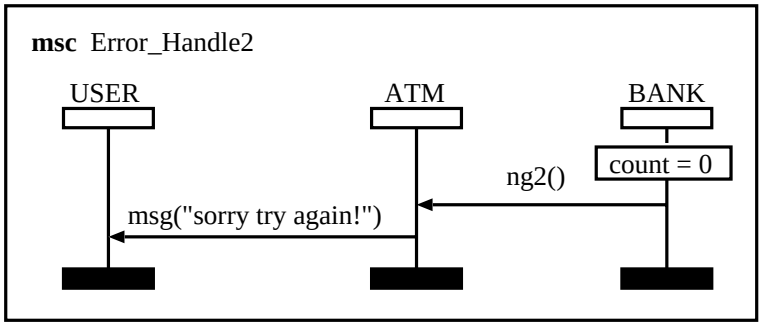



図 B.11: msc Error_Handle2

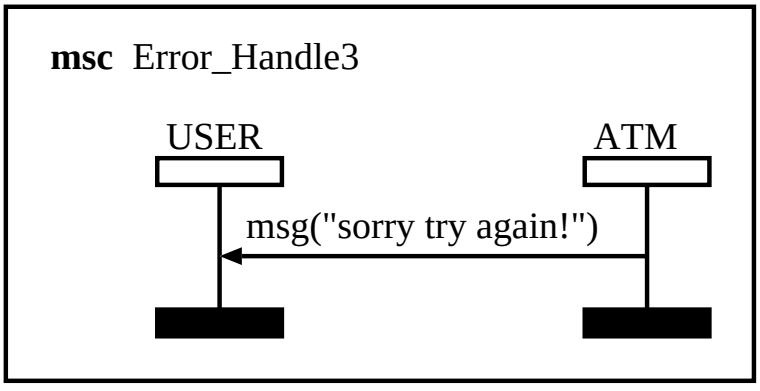



図 B.12: msc Error_Handle3

### B.3.2 テキスト形式表記

```
msc ATM_SYSTEM;
expr L1;
        L1: condition ATM_Idle seq (L2);
        L2: Select_Service seq (L3);
        L3: condition Selected_Service seq (L4);
        L4: if(service_name == "withdraw") seq (L5);
        L5: WITHDRAW seq (L1);
endmsc;

msc WITHDRAW;
expr L1;
        L1:  condition Selected_Service seq (L2);
        L2:  Passin_Message seq (L3);
        L3:  condition Before_Passin seq (L4);
        L4:  Input_Password seq (L5);
        L5:  condition Checked_Account  seq (L6);
        L6:  if((num == account_num) and (password == pass)) seq (L7);
             else if(((num == account_num) and (password != pass))
                    and (count < 2))    seq (L8);
             else if(((num == account_num) and (password != pass))
                    and (count = 2))    seq (L9);
        L7:  Input_Amount  seq (L10);
        L8:  Error_Handle1 seq (L3);
        L9:  Error_Handle2 seq (L11);
        L10: condition Checked_ATM_Balance seq (L12);
        L11: condition ATM_Idle seq (L13);
        L12: if((val < ATM_balance) or (val == ATM_balance)) seq (L14);
             else if(val > ATM_balance) seq (L15);
        L13: end;
        L14: Check_BANK_Balance seq (L16);
        L15: Error_Handle3      seq (L11);
        L16: condition Checked_BANK_Balance seq (L17);
        L17: if((num == account_num)
               and ((account_balance + (-val) > 0)
               or (account_balance + (-val) == 0)))
             seq (L18);
             else if((num == account_num) and (account_balance + (-val) < 0))
             seq (L9);
        L18: Invocation seq (L11);
endmsc;

msc Select_Service;
inst USER, ATM;
        instance USER;
                out cardin(account_num:Int)              to   ATM;
                in  msg("select service.")               from ATM;
                out request_service(service_name:String) to   ATM;
        endinstance;
        instance ATM;
                in  cardin(account_num:Int)              from USER;
                action num = account_num;
                out msg("select service.")               to   USER;
                in  request_service(service_name:String) from USER;
        endinstance;
endmsc;
```

```
msc Passin_Message;
inst USER, ATM;
        instance USER;
                in  msg("input password.")  from ATM;
        endinstance;
        instance ATM;
                out msg("input password.")  to   USER;
        endinstance;
endmsc;

msc Input_Password;
inst USER, ATM, BANK;
        instance USER;
                out passin(password:String) to   ATM;
        endinstance;
        instance ATM;
                in  passin(password:String) from USER;
                out checkaccount(num:Int, password:String) to  BANK;
        endinstance;
        instance BANK;
                in  checkaccount(num:Int, password:String) from ATM;
        endinstance;
endmsc;

msc Input_Amount;
inst USER, ATM, BANK;
        instance USER;
                in  msg("input amount.")    from ATM;
                out amountin(money_val:Int) to   ATM;
                in  msg("confirm amount.")  from ATM;
                out confirm()               to   ATM;
        endinstance;
        instance ATM;
                in  ok()  from BANK;
                out msg("input amount.")    to   USER;
                in  amountin(money_val:Int) from USER;
                action val = money_val;
                out msg("confirm amount.")  to   USER;
                in  confirm()               from USER;
        endinstance;
        instance BANK;
                action count = 0;
                out ok()  to   ATM;
        endinstance;
endmsc;

msc Check_BANK_Balance;
inst ATM, BANK;
        instance ATM;
                out checkamount(num:Int, -val:Int) to   BANK;
        endinstance;
        instance BANK;
                in  checkamount(num:Int, -val:Int) from ATM;
        endinstance;
endmsc;
```

```
msc Invocation;
inst USER, ATM, BANK;
        instance USER;
                in  moneyout(val:Int) from ATM;
                in  cardout(num:Int)  from ATM;
        endinstance;
        instance ATM;
                in  ok()  from BANK;
                action ATM_balance = ATM_balance - val;
                out moneyout(val:Int) to   USER;
                out cardout(num:Int)  to   USER;
        endinstance;
        instance BANK;
                action account_balance = account_balance + (-val);
                out ok()  to   ATM;
        endinstance;
endmsc;

msc Error_Handle1;
inst USER, ATM, BANK;
        instance USER;
                in  msg("input password again!") from ATM;
        endinstance;
        instance ATM;
                in  ng() from BANK;
                out msg("input password again!") to   USER;
        endinstance;
        instance BANK;
                action count = count + 1;
                out ng() to   ATM;
        endinstance;
endmsc;

msc Error_Handle2;
inst USER, ATM, BANK;
        instance USER;
                in  msg("sorry try again!") from ATM;
        endinstance;
        instance ATM;
                in  ng2() from BANK;
                out msg("sorry try again!") to   USER;
        endinstance;
        instance BANK;
                action count = 0;
                out ng2() to   ATM;
        endinstance;
endmsc;

msc Error_Handle3;
inst USER, ATM;
        instance USER;
                in  msg("sorry try again!") from ATM;
        endinstance;
        instance ATM;
                out msg("sorry try again!") to   ATM;
        endinstance;
endmsc;
```

## B.4 「払い戻し」に対応したObCL

```
--- withdraw.obcl ---
--- FIELD1用 Event Class
--- FIELD1は, ATMとUSERで構成されている
event FIELD1_INTEVENT
  inherit GENERIC_EVENT
  attribute var1:Int
end
event FIELD1_STREVENT
  inherit FIELD1_INTEVENT
  attribute str1:String
end

--- FIELD2用 Event Class
--- FIELD2は, ATMとBANKで構成されている
event FIELD2_INTEVENT
  inherit GENERIC_EVENT
  attribute var1,var2:Int
end
event FIELD2_STREVENT
  inherit FIELD2_INTEVENT
  attribute str1:String
end

--- ATMとUSERとの間の Field Class
field FIELD1
  event confirm:GENERIC_EVENT
  event cardin,amountin,moneyout,cardout:FIELD1_INTEVENT
  event msg,request_service,passin:FIELD1_STREVENT
end

--- ATMとBANKとの間の Field Class
field FIELD2
  event ok,ng,ng2:GENERIC_EVENT
  event checkamount:FIELD2_INTEVENT
  event checkaccount:FIELD2_STREVENT
end

--- ATM Class
Class ATM
  field F1:FIELD1; F2:FIELD2
  attribute num:Int
  attribute val:Int
  attribute ATM_balance:Int
  state ATM_Idle
  state s11,Before_Passin,Checked_Account,s14,s15,Checked_BANK_Balance
  transition
  start is
    source init
    destination ATM_Idle
  end
  t10 is
    source ATM_Idle
    input F1.cardin
    do  num := F1.cardin.var1 ;
        F1.msg.str1 := "select service."
    destination s11
    output F1.msg
  end
  t11 is
    source s11
    input F1.request_service
    when F1.request_service.str1 = "withdraw"
    do  F1.msg.str1 := "input password."
    destination Before_Passin
    output F1.msg
```

```
end
t12 is
  source Before_Passin
  input F1.passin
  do  F2.checkaccount.var1 := num ;
      F2.checkaccount.str1 := F1.passin.str1
  destination Checked_Account
  output F2.checkaccount
end
t13 is
  source Checked_Account
  input F2.ok
  do  F1.msg.str1 := "input amount."
  destination s14
  output F1.msg
end
t14 is
  source s14
  input F1.amountin
  do  val := F1.amountin.var1 ;
      F1.msg.str1 := "confirm amount."
  destination s15
  output F1.msg
end
t15 is
  source s15
  input F1.confirm
  when ( val < ATM_balance ) or ( val = ATM_balance )
  do  F2.checkamount.var1 := num ;
      F2.checkamount.var2 := -val
  destination Checked_BANK_Balance
  output F2.checkamount
end
t16 is
  source Checked_Account
  input F2.ng
  do  F1.msg.str1 := "input password again!"
  destination Before_Passin
  output F1.msg
end
t17 is
  source Checked_Account
  input F2.ng2
  do  F1.msg.str1 := "sorry try again!"
  destination ATM_Idle
  output F1.msg
end
t18 is
  source s15
  input F1.confirm
  when val > ATM_balance
  do  F1.msg.str1 := "sorry try again!"
  destination ATM_Idle
  output F1.msg
end
t19 is
  source Checked_BANK_Balance
  input F2.ok
  do  ATM_balance := ATM_balance - val ;
      F1.moneyout.var1 := val ;
      F1.cardout.var1 := num
  destination ATM_Idle
  output {F1.moneyout,F1.cardout}
end
t20 is
  source Checked_BANK_Balance
  input F2.ng2
```

```
    do  F1.msg.str1 := "sorry try again!"
    destination ATM_Idle
    output F1.msg
  end
end

--- BANK Class
Class BANK
  field F2:FIELD2
  attribute count:Int
  attribute account_balance:Int
  attribute account_num:Int
  attribute pass:String
  state Before_Passin
  state Checked_ATM_Balance,ATM_Idle
  transition
  start is
    source init
    destination Before_Passin
  end
  t10 is
    source Before_Passin
    input F2.checkaccount
    when ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 = pass )
    do  count := 0
    destination Checked_ATM_Balance
    output F2.ok
  end
  t11 is
    source Before_Passin
    input F2.checkaccount
    when ( ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 <> pass ) ) and ( count < 2 )
    do  count := count + 1
    destination Before_Passin
    output F2.ng
  end
  t12 is
    source Before_Passin
    input F2.checkaccount
    when ( ( F2.checkaccount.var1 = account_num )
         and ( F2.checkaccount.str1 <> pass ) ) and ( count = 2 )
    do  count := 0
    destination ATM_Idle
    output F2.ng2
  end
  t13 is
    source Checked_ATM_Balance
    input F2.checkamount
    when ( F2.checkamount.var1 = account_num )
         and ( ( account_balance + ( F2.checkamount.var2 ) > 0 )
         or ( account_balance + ( F2.checkamount.var2 ) = 0 ) )
    do  account_balance := account_balance + ( F2.checkamount.var2 )
    destination ATM_Idle
    output F2.ok
  end
  t14 is
    source Checked_ATM_Balance
    input F2.checkamount
    when ( F2.checkamount.var1 = account_num )
         and ( account_balance + ( F2.checkamount.var2 ) < 0 )
    do  count := 0
    destination ATM_Idle
    output F2.ng2
  end
end
```

## B.5 「払い戻し」に対応したObCLコードのテスト

以下に「払い戻し」をテストするための外部入力イベント列を示す.

暗証番号を3回間違える場合

```
F1.cardin(var1=Int 100)
F1.request_service(str1=String "withdraw")
F1.passin(str1=String "error")
F1.passin(str1=String "error")
F1.passin(str1=String "error")
F1.amountin(var1=Int 15)
F1.confirm
```

暗証番号を2回間違える場合

```
F1.cardin(var1=Int 100)
F1.request_service(str1=String "withdraw")
F1.passin(str1=String "error")
F1.passin(str1=String "error")
F1.passin(str1=String "password")
F1.amountin(var1=Int 15)
F1.confirm
```

暗証番号を1回間違える場合

```
F1.cardin(var1=Int 100)
F1.request_service(str1=String "withdraw")
F1.passin(str1=String "error")
F1.passin(str1=String "password")
F1.amountin(var1=Int 15)
F1.confirm
```

100払い戻す場合

```
F1.cardin(var1=Int 100)
F1.request_service(str1=String "withdraw")
F1.passin(str1=String "password")
F1.amountin(var1=Int 100)
F1.confirm
```

200払い戻す場合

```
F1.cardin(var1=Int 100)
F1.request_service(str1=String "withdraw")
F1.passin(str1=String "password")
F1.amountin(var1=Int 200)
F1.confirm
```

## B.6　ATM システムの MSC

ここでは，ATM システムの「預入れ」を除いた MSC を示す.

### B.6.1　グラフィカル表記

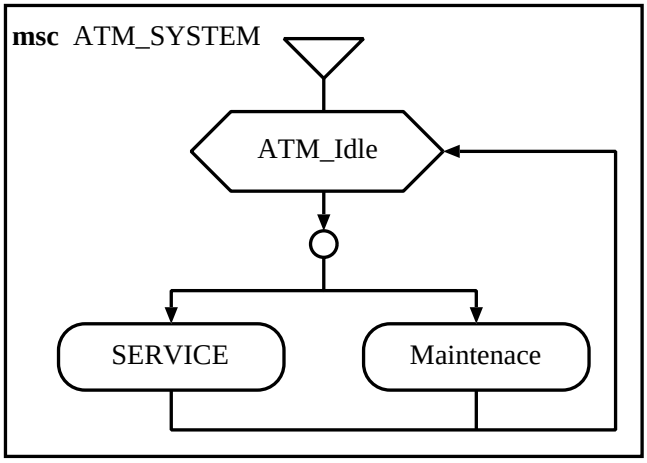



図 B.13: msc ATM_SYSTEM

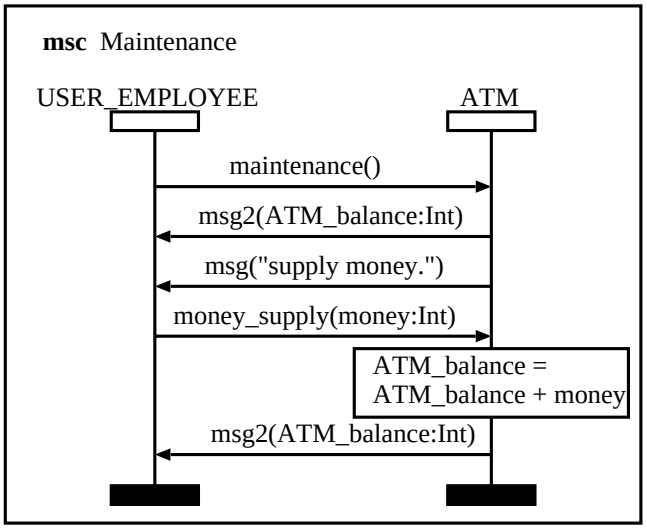



図 B.14: msc Maintenance

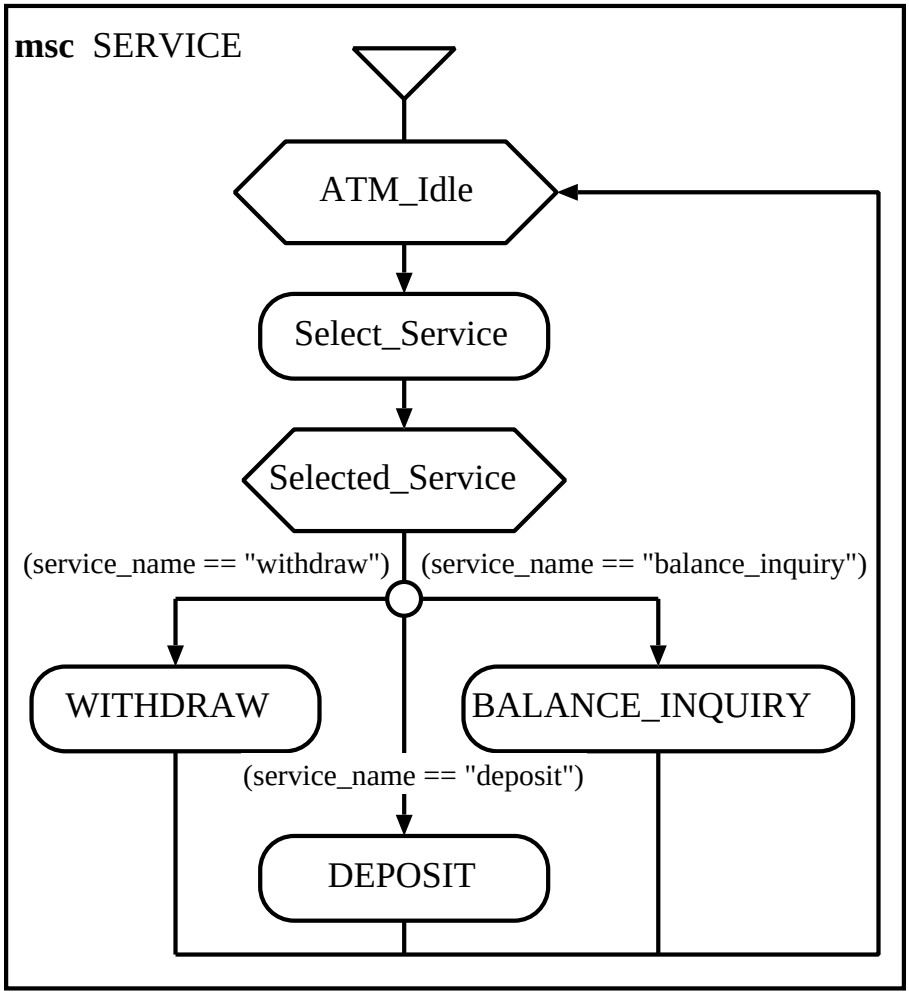



図 B.15: msc SERVICE

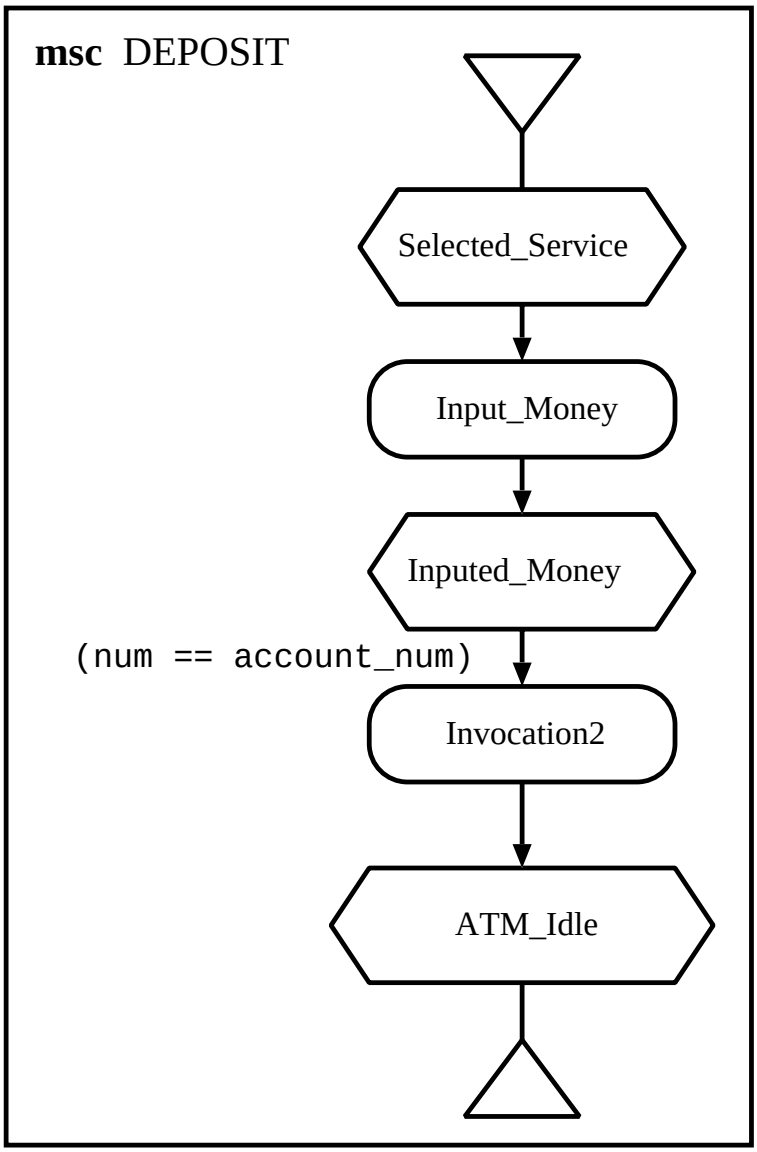



図 B.16: msc DEPOSIT

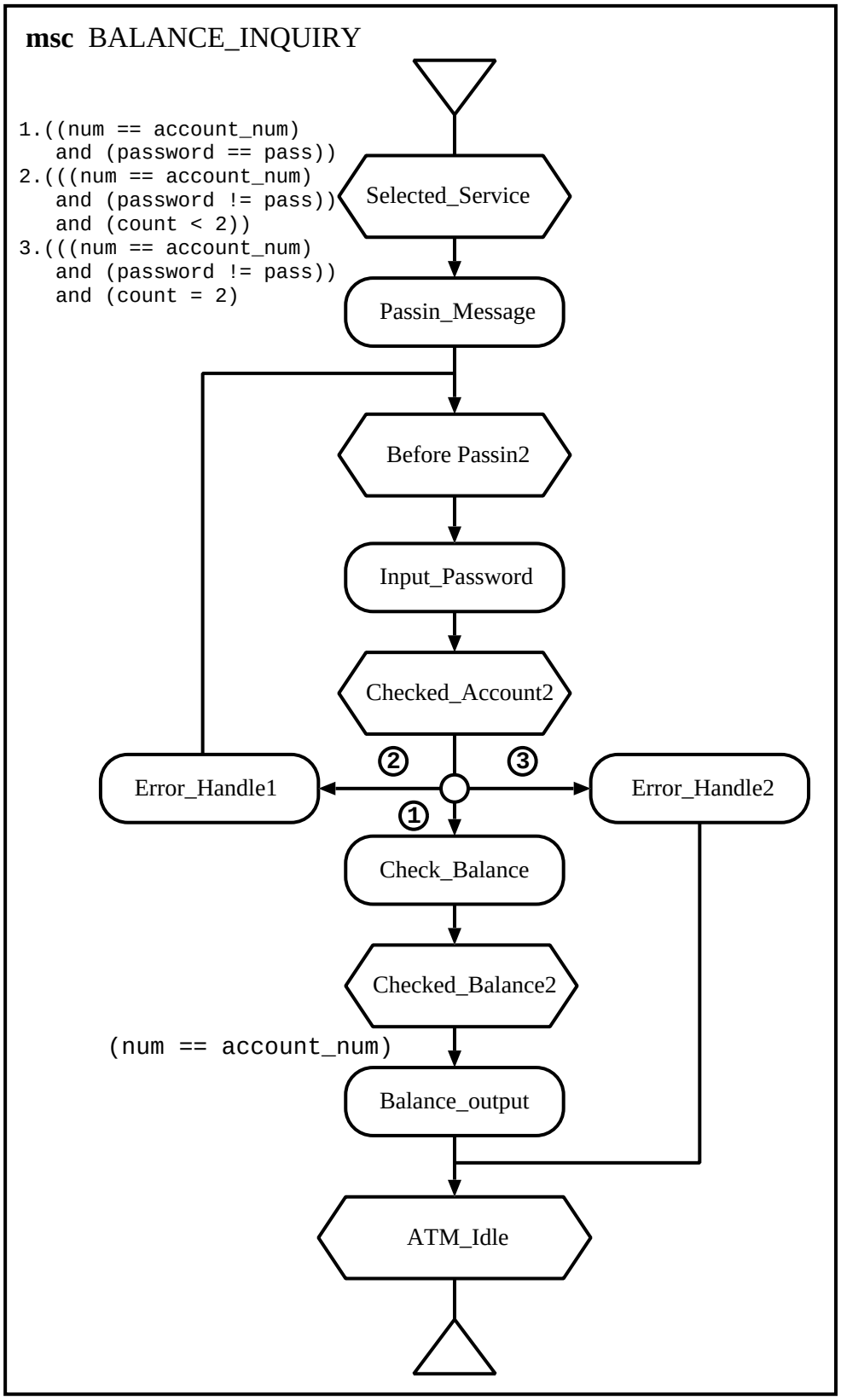



図 B.17: msc BALANCE_INQUIRY

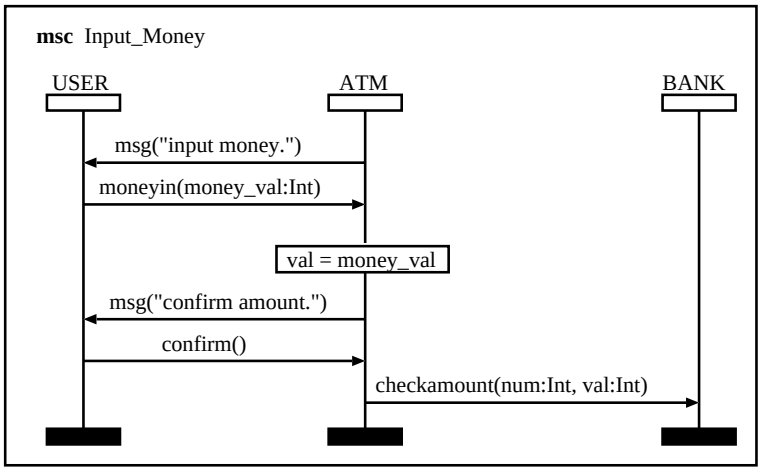



図 B.18: msc Input_Money

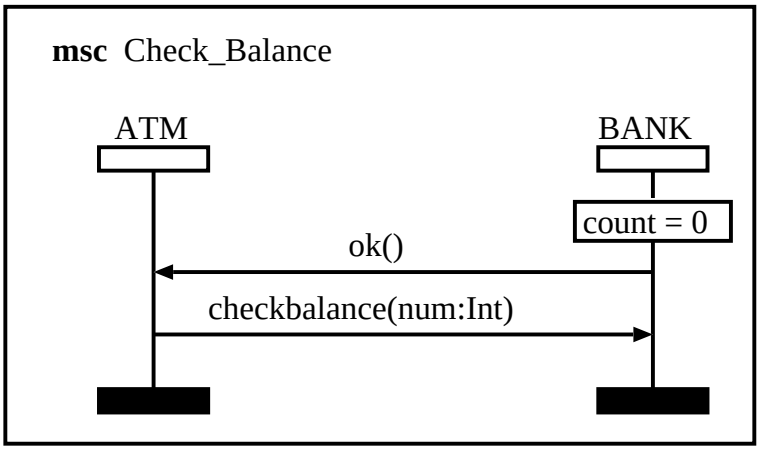



図 B.19: msc Check_Balance

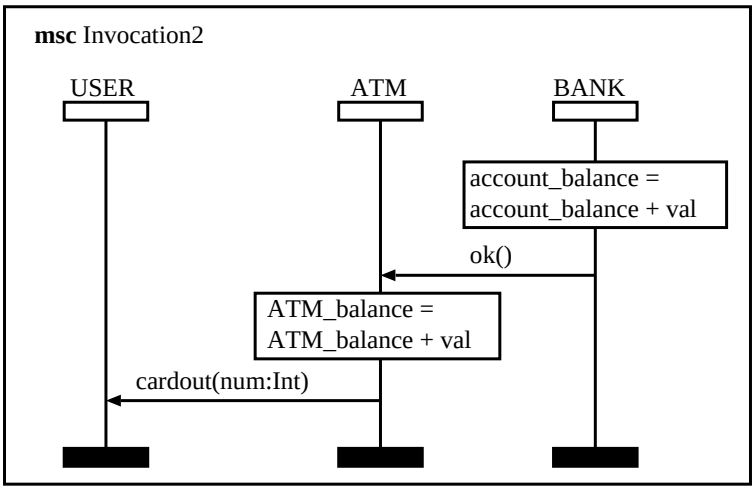



図 B.20: msc Invocation2

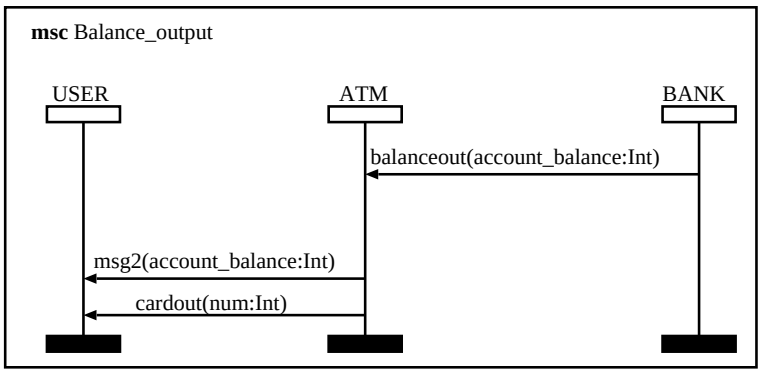



図 B.21: msc Balance_Output

### B.6.2 テキスト形式表記

```
msc ATM_SYSTEM;
expr L1;
        L1: condition ATM_Idle seq (L2);
        L2: if(note: start service) seq (L3);
            else if(note: start maintenance) seq (L4);
        L3: SERVICE seq (L1);
        L4: Maintenance seq (L1);
endmsc;

msc SERVICE;
expr L1;
        L1: condition ATM_Idle seq (L2);
        L2: Select_Service seq (L3);
        L3: condition Selected_Service seq (L4);
        L4: if(service_name == "withdraw") seq (L5);
            else if(service_name == "deposit") seq (L6);
            else if(service_name == "balance_inquiry") seq (L7);
        L5: WITHDRAW seq (L1);
        L6: DEPOSIT seq (L1);
        L7: BALANCE_INQUIRY seq (L1);
endmsc;

msc Maintenance;
inst USER_EMPLOYEE, ATM;
        instance USER_EMPLOYEE;
                out maintenance() to   ATM;
                in  msg2(ATM_balance:Int) from ATM;
                in  msg("supply money.") from ATM;
                out money_supply(money:Int) to   ATM;
                in  msg2(val:ATM_balance) from ATM;
        endinstance;
        instance ATM;
                in  maintenance() from USER_EMPLOYEE;
                out msg2(ATM_balance:Int) to   USER_EMPLOYEE;
                out msg("supply money.") to    USER_EMPLOYEE;
                in  money_supply(money:Int) from USER_EMPLOYEE;
                action ATM_balance = ATM_balance + money;
                out msg2(ATM_balance:Int) to   USER_EMPLOYEE;
        endinstance;
endmsc;

msc DEPOSIT;
expr L1;
        L1:  condition Selected_Service seq (L2);
        L2:  Input_Money  seq (L3);
        L3:  condition Inputed_Money seq (L4);
        L4:  if (num == account_num) seq (L5);
        L5: Invocaton2 seq (L6);
        L6: condition ATM_Idle seq (L7);
        L7: end;
endmsc;
```

```
msc BALANCE_INQUIRY;
expr L1;
        L1:  condition Selected_Service seq (L2);
        L2:  Passin_Message seq (L3);
        L3:  condition Before_Passin seq (L4);
        L4:  Input_Password seq (L5);
        L5:  condition Checked_Account  seq (L6);
        L6:  if((num == account_num) and (password == pass)) seq (L7);
             else if(((num == account_num) and (password != pass))
                    and (count < 2))    seq (L8);
             else if(((num == account_num) and (password != pass))
                    and (count == 2))   seq (L9);
        L7:  Check_Balance seq (L10);
        L8:  Error_Handle1 seq (L3);
        L9:  Error_Handle2 seq (L14);
        L10: condition Checked_Balance sep (L11);
        L11: if (num == account_num) seq (L12);
        L12: Balance_Output seq (L13);
        L13: condition ATM_Idle seq (L14);
        L14: end;
endmsc;

msc Input_Money;
inst USER, ATM, BANK;
        instance USER;
                in  msg("input money.")     from ATM;
                out moneyin(money_val:Int)  to   ATM;
                in  msg("confirm amount.")  from ATM;
                out confirm()               to   ATM;
        endinstance;
        instance ATM;
                out msg("input money.")     to   USER;
                in  moneyin(money_val:Int)  from USER;
                action val = money_val;
                out msg("confirm amount.")  to   USER;
                in  confirm()               from USER;
                out checkamount(num:Int, val:Int) to   BANK;
        endinstance;
        instance BANK;
                in  checkamount(num:Int, val:Int) from ATM;
        endinstance;
endmsc;

msc Check_Balance;
inst ATM, BANK;
        instance ATM;
                in  ok()  from BANK;
                out checkbalance(num:Int) to   BANK;
        endinstance;
        instance BANK;
                action count = 0;
                out ok()  to   ATM;
                in  checkbalance(num:Int) from ATM;
        endinstance;
endmsc;
```

```
msc Invocation2;
inst USER, ATM, BANK;
        instance USER;
                in  cardout(num:Int)        from ATM;
        endinstance;
        instance ATM;
                in  ok()  from BANK;
                action ATM_balance = ATM_balance + val;
                out cardout(num:Int)        to   USER;
        endinstance;
        instance BANK;
                action account_balance = account_balance + val;
                out ok()  to   ATM;
        endinstance;
endmsc;

msc Balance_Output;
inst USER, ATM, BANK;
        instance USER;
                in  msg2(account_balance:Int) from ATM;
                in  cardout(num:Int)  from ATM;
        endinstance;
        instance ATM;
                in  balanceout(account_balance:Int) from BANK;
                out msg2(account_balance:Int) to   USER;
                out cardout(num:Int)  to   USER;
        endinstance;
        instance BANK;
                out balanceout(account_balance:Int) to   ATM;
        endinstance;
endmsc;
```

## B.7 ATMシステムのObCL

```
--- ATM_sys.obcl ---
--- FIELD1用 Event Class
--- FIELD1は, ATMとUSER_EMPLOYEEで構成されている
event FIELD1_INTEVENT
  inherit GENERIC_EVENT
  attribute var1:Int
end
event FIELD1_STREVENT
  inherit FIELD1_INTEVENT
  attribute str1:String
end

--- FIELD2用 Event Class
--- FIELD2は, ATMとUSERで構成されている
event FIELD2_INTEVENT
  inherit GENERIC_EVENT
  attribute var1:Int
end
event FIELD2_STREVENT
  inherit FIELD2_INTEVENT
  attribute str1:String
end

--- FIELD3用 Event Class
--- FIELD3は, ATMとBANKで構成されている
event FIELD3_INTEVENT
  inherit GENERIC_EVENT
  attribute var1,var2:Int
end
event FIELD3_STREVENT
  inherit FIELD3_INTEVENT
  attribute str1:String
end

--- ATMとUSER_EMPLOYEEとの間の Field Class
field FIELD1
  event maintenance:GENERIC_EVENT
  event msg2,money_supply:FIELD1_INTEVENT
  event msg:FIELD1_STREVENT
end

--- ATMとUSERとの間の Field Class
field FIELD2
  event confirm:GENERIC_EVENT
  event cardin,amountin,moneyin,moneyout,cardout,msg2:FIELD2_INTEVENT
  event msg,request_service,passin:FIELD2_STREVENT
end

--- ATMとBANKとの間の Field Class
field FIELD3
  event ok,ng,ng2:GENERIC_EVENT
  event checkamount,checkbalance,balanceout:FIELD3_INTEVENT
  event checkaccount:FIELD3_STREVENT
end

--- ATM Class
Class ATM
  field F1:FIELD1; F2:FIELD2; F3:FIELD3
  attribute ATM_balance:Int
  attribute num:Int
  attribute val:Int
  state ATM_Idle
  state s11,s13,Before_Passin,Checked_Account,
        s16,s17,Checked_BANK_Balance,s22,s23,
        Inputed_Money,Passin2,Account2,Balance2
```

```
transition
start is
  source init
  destination ATM_Idle
end
t10 is
  source ATM_Idle
  input F1.maintenance
  do  F1.msg2.var1 := ATM_balance ;
      F1.msg.str1 := "supply money."
  destination s11
  output {F1.msg2,F1.msg}
end
t11 is
  source s11
  input F1.money_supply
  do  ATM_balance := ATM_balance + F1.money_supply.var1 ;
      F1.msg2.var1 := ATM_balance
  destination ATM_Idle
  output F1.msg2
end
t12 is
  source ATM_Idle
  input F2.cardin
  do  num := F2.cardin.var1 ;
      F2.msg.str1 := "select service."
  destination s13
  output F2.msg
end
t13 is
  source s13
  input F2.request_service
  when F2.request_service.str1 = "withdraw"
  do  F2.msg.str1 := "input password."
  destination Before_Passin
  output F2.msg
end
t14 is
  source Before_Passin
  input F2.passin
  do  F3.checkaccount.var1 := num ;
      F3.checkaccount.str1 := F2.passin.str1
  destination Checked_Account
  output F3.checkaccount
end
t15 is
  source Checked_Account
  input F3.ok
  do  F2.msg.str1 := "input amount."
  destination s16
  output F2.msg
end
t16 is
  source s16
  input F2.amountin
  do  val := F2.amountin.var1 ;
      F2.msg.str1 := "confirm amount."
  destination s17
  output F2.msg
end
t17 is
  source s17
  input F2.confirm
  when ( val < ATM_balance ) or ( val = ATM_balance )
  do  F3.checkamount.var1 := num ;
      F3.checkamount.var2 := -val
  destination Checked_BANK_Balance
```

```
  output F3.checkamount
end
t18 is
  source Checked_Account
  input F3.ng
  do  F2.msg.str1 := "input password again!"
  destination Before_Passin
  output F2.msg
end
t19 is
  source Checked_Account
  input F3.ng2
  do  F2.msg.str1 := "sorry try again!"
  destination ATM_Idle
  output F2.msg
end
t20 is
  source s17
  input F2.confirm
  when val > ATM_balance
  do  F2.msg.str1 := "sorry try again!"
  destination ATM_Idle
  output F2.msg
end
t21 is
  source Checked_BANK_Balance
  input F3.ok
  do  ATM_balance := ATM_balance - val ;
      F2.moneyout.var1 := val ;
      F2.cardout.var1 := num
  destination ATM_Idle
  output {F2.moneyout,F2.cardout}
end
t22 is
  source Checked_BANK_Balance
  input F3.ng2
  do  F2.msg.str1 := "sorry try again!"
  destination ATM_Idle
  output F2.msg
end
t23 is
  source s13
  input F2.request_service
  when F2.request_service.str1 = "deposit"
  do  F2.msg.str1 := "input money."
  destination s22
  output F2.msg
end
t24 is
  source s22
  input F2.moneyin
  do  val := F2.moneyin.var1 ;
      F2.msg.str1 := "confirm amount."
  destination s23
  output F2.msg
end
t25 is
  source s23
  input F2.confirm
  do  F3.checkamount.var1 := num ;
      F3.checkamount.var2 := val
  destination Inputed_Money
  output F3.checkamount
end
t26 is
  source Inputed_Money
```

```
    input F3.ok
    do  ATM_balance := ATM_balance + val ;
        F2.cardout.var1 := num
    destination ATM_Idle
    output F2.cardout
  end
  t27 is
    source s13
    input F2.request_service
    when F2.request_service.str1 = "balance_inquiry"
    do  F2.msg.str1 := "input password."
    destination Passin2
    output F2.msg
  end
  t28 is
    source Passin2
    input F2.passin
    do  F3.checkaccount.var1 := num ;
        F3.checkaccount.str1 := F2.passin.str1
    destination Account2
    output F3.checkaccount
  end
  t29 is
    source Account2
    input F3.ok
    do  F3.checkbalance.var1 := num
    destination Balance2
    output F3.checkbalance
  end
  t30 is
    source Account2
    input F3.ng
    do  F2.msg.str1 := "input password again!"
    destination Passin2
    output F2.msg
  end
  t31 is
    source Account2
    input F3.ng2
    do  F2.msg.str1 := "sorry try again!"
    destination ATM_Idle
    output F2.msg
  end
  t32 is
    source Balance2
    input F3.balanceout
    do  F2.msg2.var1 := F3.balanceout.var1 ;
        F2.cardout.var1 := num
    destination ATM_Idle
    output {F2.msg2,F2.cardout}
  end
end

--- BANK Class
Class BANK
  field F3:FIELD3
  attribute count:Int
  attribute account_balance:Int
  attribute account_num:Int
  attribute pass:String
  state Before_Passin
  state Checked_ATM_Balance,ATM_Idle,s14,Passin2
  transition
  start is
    source init
    destination Before_Passin
  end
```

```
t10 is
  source Before_Passin
  input F3.checkaccount
  when ( F3.checkaccount.var1 = account_num )
       and ( F3.checkaccount.str1 = pass )
  do  count := 0
  destination Checked_ATM_Balance
  output F3.ok
end
t11 is
  source Before_Passin
  input F3.checkaccount
  when ( ( F3.checkaccount.var1 = account_num )
       and ( F3.checkaccount.str1 <> pass ) ) and ( count < 2 )
  do  count := count + 1
  destination Before_Passin
  output F3.ng
end
t12 is
  source Before_Passin
  input F3.checkaccount
  when ( ( F3.checkaccount.var1 = account_num )
       and ( F3.checkaccount.str1 <> pass ) ) and ( count = 2 )
  do  count := 0
  destination ATM_Idle
  output F3.ng2
end
t13 is
  source Checked_ATM_Balance
  input F3.checkamount
  when ( F3.checkamount.var1 = account_num )
       and ( ( account_balance + ( F3.checkamount.var2 ) > 0 )
       or ( account_balance + ( F3.checkamount.var2 ) = 0 ) )
  do  account_balance := account_balance + ( F3.checkamount.var2 )
  destination ATM_Idle
  output F3.ok
end
t14 is
  source Checked_ATM_Balance
  input F3.checkamount
  when ( F3.checkamount.var1 = account_num )
       and ( account_balance + ( F3.checkamount.var2 ) < 0 )
  do  count := 0
  destination ATM_Idle
  output F3.ng2
end
t15 is
  source Selected_Service
  input F3.checkamount
  when F3.checkamount.var1 = account_num
  do  account_balance := account_balance + F3.checkamount.var2
  destination ATM_Idle
  output F3.ok
end
t16 is
  source Passin2
  input F3.checkaccount
  when ( F3.checkaccount.var1 = account_num )
       and ( F3.checkaccount.str1 = pass )
  do  count := 0
  destination s14
  output F3.ok
end
t17 is
  source s14
  input F3.checkbalance
  when F3.checkbalance.var1 = account_num
```

```
    do  F3.balanceout.var1 := account_balance
    destination ATM_Idle
    output F3.balanceout
  end
  t18 is
    source Passin2
    input F3.checkaccount
    when ( ( F3.checkaccount.var1 = account_num )
         and ( F3.checkaccount.str1 <> pass ) ) and ( count < 2 )
    do  count := count + 1
    destination Passin2
    output F3.ng
  end
  t19 is
    source Passin2
    input F3.checkaccount
    when ( ( F3.checkaccount.var1 = account_num )
         and ( F3.checkaccount.str1 <> pass ) ) and ( count = 2 )
    do  count := 0
    destination ATM_Idle
    output F3.ng2
  end
end
```