

2003年度
卒業論文

UML2.0に基づく構造的モデリング

神戸大学工学部情報知能工学科
佐藤 真紗美

指導教官 林晋

2004年2月17日

UML2.0 に基づく構造的モデリング

佐藤 真紗美

要旨

昨年度林研究室で製作された SMART0.2 は、分散環境にある複数の機器をモデリングし、その実行をシミュレーションできるシステムであったが、クラス概念の導入が導入されていなかった。また、2003年6月に Object Management Group(OMG) によって策定された UML(Unified Modeling Language)2.0 により、13種の記法がモデリングに用いられるようになったが、その一方で多数の記法によって表現された1つの Classifier を、ユーザがそれらの記法を結びつけて理解できるような手法が必要である。

そのため、本論文ではまずクラス概念として UML の Classifier を導入し、そのうえで、その Classifier の表現方法の1つとして Model 概念を提案する。Model を表現するために用いた Model 図は、UML の Composite Structure 図による Classifier の内部構造的階層化を軸として、User Interface、Use Case、Sequence 図、State Machine 図および Composite Structure 図を、1つの Classifier の表現として提供するものである。

さらに、本論文では、SMART0.2 の User Interface Modeler における問題点から、組み込み機器の User Interface モデリングにおける概念的な諸問題を論じる。また、SMART ではモデリングしたオブジェクトの実行をシミュレートすることができるが、その実行状態のログを記録し、それをを用いて UML の Sequence 図の形で実行状態をトレースしたり、モデリング上の不具合の発見に役立てたりする方法を提案した。

以上のようなアイデアを元に、SMART0.2 の User Interface Modeler 及び State-chart Modeler を核として、SMART0.3 の UML Modeler を製作した。さらに、コンポーネントベースの開発手法 Catalysis との対比を行い、ログの将来的な方向性について論じた。

目次

第1章 序論	4
第2章 Model	6
2.1 Model 概念	6
2.2 Model 図の概要	6
2.3 Model 図の構造図	9
2.4 Example	9
2.4.1 リモコン付きエアコン Model の概要	9
2.4.2 リモコン付きエアコン Model	10
2.4.3 リモコン Model	12
2.4.4 温度上昇ボタン Model	14
2.4.5 エアコン Model	14
2.5 Model Instance	15
2.6 実行状態	16
2.7 UML2.0 との比較	17
第3章 User Interface	19
3.1 SMART0.2 の UI Modeler の問題	19
3.2 User Interface モデリングの問題点	19
3.2.1 UI Model と UI Instance	19
3.2.2 UI のプロトタイプ	22
3.2.3 User Interface の継承関係	22
3.2.4 論理部分と外見部分	22
第4章 SMART UML Modeler	24
4.1 Model の実装	24
4.2 User Interface	24
4.3 実行状態	25
4.4 ログの取得と解析	25
4.5 Composite Structure のチェック機能	28

第 5 章	結論および今後の展開	31
5.1	結論	31
5.2	モデリングの方法論	31
5.2.1	Component-Based Development - Catalysis -	31
5.2.2	Refinement と Model	31
5.3	ログに関する考察	32
5.3.1	ログの形式	32
5.3.2	ログの活用	33
5.4	SMART の改良	33
5.4.1	User Interface の適切な実装	33
5.4.2	Instanciation 時の activity	33
5.4.3	Instance のコンストラクション	34
5.4.4	Composite Structure 図と Sequence 図の関係	35
5.4.5	より複雑な Composite Structure の記述と実行	35
5.4.6	多機能先行モデリング	36
5.4.7	非同期メッセージのサポート	36
	謝辞	37
	参考文献	38
付録 A	UML2.0	39
A.1	UML2.0 の構成	39
A.2	図の種類	39
A.3	UML2.0 の特徴	40
A.4	Composite Structure	44
A.5	Component	46
付録 B	Catalysis	47
B.1	Basic Concept	47
B.2	Three Modeling Concepts	48
B.3	Refinement の例	50
B.4	Three Levels of Modeling	55
B.5	Three Principles	56
B.6	Catalysis での記法	56

第1章 序論

情報家電という言葉がようやく一般にも浸透し始めた昨今、分散環境を前提として、こうした機器の効率的かつ正確さのある開発手法が求められている。昨年度、林研究室で製作されたシステム SMART0.2 は、こうした中で、複数機器のモデリング及び実行のシミュレーションができるシステムとして開発された。この SMART0.2 は、Unified Modeling Language(UML)1.X の Statechart 図と SMART 用の Action 言語 SAL(SMART Action Language) を用いて機器の動作を記述し、機器の User Interface をグラフィカルにデザインした上で、その仮想的な User Interface を用いて機器の実行状況をテストできるものであった。

この SMART0.2 については、[Mori 03] において、オブジェクト指向のクラス概念を導入する必要性が指摘されている。というのも、SMART0.2 でモデリングされる機器はいずれもオブジェクト単位であり、クラスを用いた抽象的な設計はサポートされていないからである。本論文で製作する SMART0.3 においては、まずこのクラス概念を導入した。

一方、2003年6月に、Object Management Group(OMG)により、UML初のメジャーバージョンアップとなる UML2.0 が策定された。この UML2.0 は、UML1.X の意味論や記法に対して指摘されていた様々な問題点に対処している。たとえば、UML1.X に含まれていた実装に関するメタクラス (Method、Object 等) を排除したり、クラスの内部構造を記述するために Composite Structure という概念を導入したり、また、オブジェクト間のインタラクションに関するメタクラスや記法が整理されたりもしている。その詳細は [UML2 03] に述べられており、その一部を付録 A で説明することとするが、UML2.0 によるモデリングは現在その可能性を試されている段階であるといえよう。

UML2.0 では 13 種類の記法が定義されているが、この記法の中には互いに関係を持つものも多い。たとえば、Composite Structure 図の Part-Connector の関係は Sequence 図の Lifeline-Message の関係に反映されている必要がある。また、Composite Structure 図の Part は、別の Classifier の存在を表現している。1 つの Classifier に対する表現方法が多面的に存在する以上、それらをユーザが結びつけて理解できるようにする手段が必要であると考える、本論文では Model 図を提案する。

本研究は、同研究室の潘沂冰との共同研究の一部であり、潘はUMLモデリングにおけるテスト駆動開発の研究を行っている。特に、本論文で述べている実行状態のログに関する研究は、同研究の一環でもある。

本論文の構成は以下のようなものである。まず第2章においてこのModelという考え方について説明したうえで、Model図の具体例を挙げてこれを説明する。次に、第3章では、SMART0.2のUser Interface Modelerにおける問題点と、主に組み込み機器をターゲットとしてUser Interfaceデザインに関する諸問題を論じる。さらに、第4章では、本論文で製作したSMART0.3のUML Modelerについて説明する。ここには、Model図の実装のみならず、実行状態のログの取得及びその解析についても論じることとする。第5章では、本論文の結論、コンポーネントベースの開発手法Catalysisにおける対比、ログに関する考察、そしてSMART0.3に残された課題を述べる。なお、UML2.0に関する説明とCatalysisに関する説明については、それぞれ付録Aと付録Bとして付した。

第2章 Model

2.1 Model概念

Modelとは、User Interfaceを持つことができる Classifierの「表現」方法の一種である。つまり、ある Classifierの中身を、わかりやすくユーザに伝えるための手段である。

Modelは、この Modelが表す Classifierを必ず1つ持つ。また、Modelは、この Classifierを表現するための次の5つを持っている。

- Composite Structure
- User Interface (UI)
- State Machine
- Interaction (as Sequence Diagram)
- Use Case

これらを持つことから、この Classifierは、“Structured and behaviored classifier with User Interface”と位置づけることができる。

また、Modelは、Instantiateされるためのコンストラクタを持つことができる。Modelの Instanceについては2.5で述べる。

2.2 Model図の概要

Modelをビジュアルに表現するためには Model図を用いる。Model図は Fig2.1のように表記する。なお、青色を用いて表記した部分は説明として付記したもので、これ以降においても同様である。

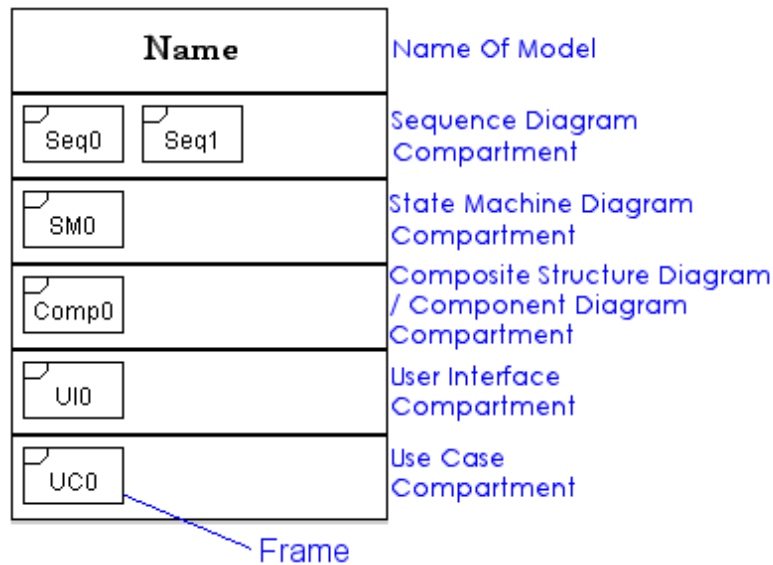


Fig 2.1: Model Diagram

Fig 2.1 で、6 つ連なる枠を、それぞれコンパートメントと呼ぶ。以下では、各コンパートメントの内容について説明する。

Name of Model このコンパートメントには、この Model 図で示される Classifier の名前を書く。なお、この Classifier は、この Classifier に属する属性 (Property)、操作 (Operation) および、この Classifier が処理する可能性のあるイベントのリストを持つことができる。

Sequence Diagram Compartment このコンパートメントには、この Model の内部で起こる相互作用 (Interaction) を表現する Sequence 図へのリンクを持つフレームを書く。このフレームは 0 個以上存在する。

フレームのリンク先である Sequence 図は、この Model が持つ相互作用の動的側面を書いたもので、後述の Composite Structure 図では同じ相互作用の静的側面を表している。したがって Sequence 図の各 Lifeline の名前は、一般に Composite Structure 図の各 Part の名前と同じである。

State Machine Diagram Compartment このコンパートメントには、この Model の内部で起こる状態遷移を表現する State Machine 図へのリンクを持つフレームを書く。このフレームは 0 個または 1 個存在する。

フレームのリンク先の State Machine 図では、各状態 (State) および遷移 (Transition) が Activity を持つことがある。この場合、これらの状態と遷移は、この Activity

を表現した Activity を具体的に記述したもの (Activity 図を含み、テキストでもよい) へのリンクを持つことができる。

Composite Structure Diagram/Component Diagram Compartment このコンパートメントには、この Model の静的な内部構造を表現する Composite Structure 図または Component 図へのリンクを持つフレームを書く。このフレームは 0 個または 1 個存在する。

フレームのリンク先では、この Model の内部構造として、この Model の構成要素が示される。例えば、自動車 Model の中でホイールやエンジンは Part という構成要素であり、「ホイールとエンジンの協調関係」を表現する Collaboration Occurrence もまた構成要素である。

これらの構成要素の存在は、この Model を構成する下位の Model の存在を指しているため、この構成要素に該当する Model を表現する Model 図へのリンクを持つ。

User Interface Compartment このコンパートメントには、このモデルの持つ User Interface へのリンクを持つフレームを書く。このフレームは必ず 1 個存在する。

Use Case Compartment このコンパートメントには、この Model が持つユースケースへのリンクを持つフレームを書く。このフレームは、0 個以上存在する。

このユースケースは、テキストによるユースケース記述や、前述したシーケンス図による記述などでもよい。例えば、シーケンス図がユースケースとして用いられる場合には、このコンパートメントに示されたフレームのリンク先には、この Model のシーケンス図へのリンクが記述されている。

属性 Model によって表現される Classifier は、属性 (Attribute) を持つ。これは、UML2.0 では Property というメタクラスによって表現されている。Property は Classifier の Instance の特性を表現するための構造的特徴のことであるが、Property が直接具体的な値を持つことはない。Property はあくまでモデリング時に Classifier がどのような特徴を持つかを表現するものであり、Classifier の Instance が Instance の特徴として持つ具体的な値については別で示されている。これについては、2.6 で詳述する。

2.3 Model図の構造図

以上のような Model 図の構造を示したクラス図を Fig2.2 に示す。

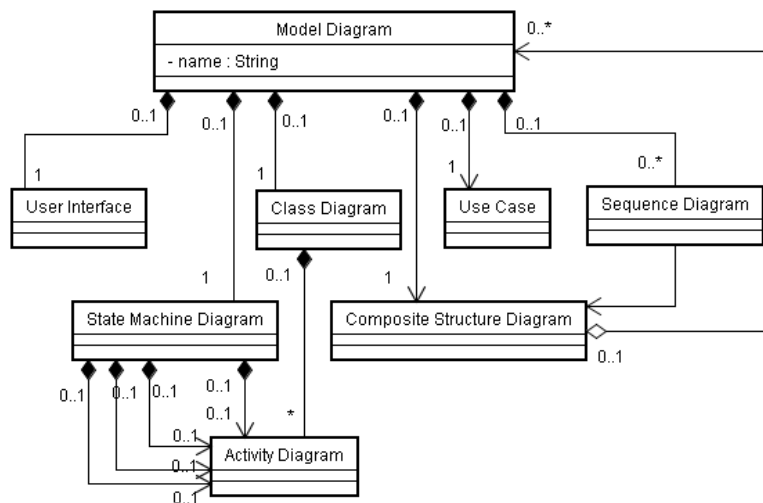


Fig 2.2: Structure Of Model Diagram

2.4 Example

この章では、2.2 で説明した Model 図を、「リモコン付きエアコン」の例を用いて具体的に説明する。

2.4.1 リモコン付きエアコン Model の概要

ここで説明するリモコン付きエアコンは、エアコン本体 (以下では「エアコン」と省略する) とリモコンの2つで構成され、この2つの装置間は赤外線メッセージを送受信する。ただし、本論文では送信された赤外線が確実に受信されることを前提とし、赤外線が障害物に当たって受信されないなどの場合については今後の課題とする。リモコンのユーザインタフェースは、液晶パネル・電源ボタン・温度上昇ボタン・温度下降ボタン・冷房と暖房のモード切替ボタンなどの部品で構成される。

リモコンは、電源ボタンを押されると、エアコンの電源が切れていれば電源を

ONにし、電源が入っていれば電源をOFFにするためのメッセージをエアコンに飛ばす。同様に、温度上昇ボタンが押されたとき、リモコンが持つ設定温度が26度未満であれば設定温度を1度上げて、エアコンに設定温度を転送するメッセージを送り、26度以上であれば設定温度を変化させず、メッセージも飛ばさない。ほかのボタンも同様にそれぞれ役割を持っているものとする。

以上のように、このリモコン付きエアコン Model は、Fig2.3のような Composition 関係を持っている。

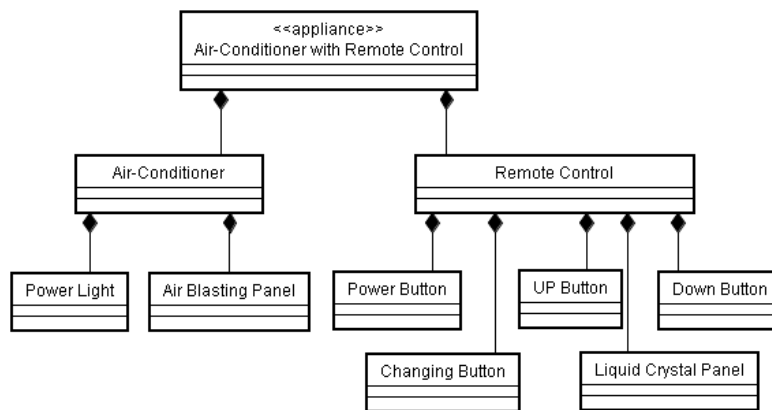


Fig 2.3: Composition Of Air-Conditioner With Remote Control

2.4.2 リモコン付きエアコン Model

Fig2.4 に、このリモコン付きエアコンを Model 図に表したものを示す。

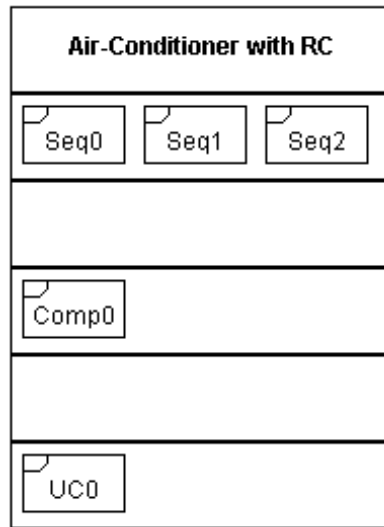


Fig 2.4: Model Diagram Of Air-Conditioner With Remote Control

第2.2章で説明したように、各コンパートメントには最上段を除いてそれぞれフレームのみが表示される。各フレームの中の文字列は、このフレームによってリンクされた図の名前を示している。SMART0.3では、これらのフレームをクリックすると、リンク先の図が開く。

例えば、図の Comp0 には、このリモコン付きエアコンの内部構造を示した Composite Structure 図へのリンクが存在している。したがって、このフレームからは、Fig2.5 に示される Composite Structure 図が導かれる。

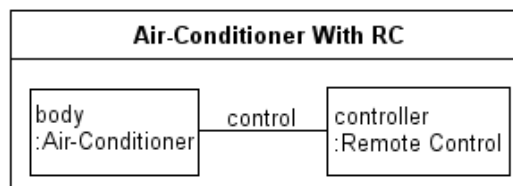


Fig 2.5: Composite Structure Diagram Of Fig2.4

また、Fig2.5 の内部構造の Interaction を表現した Sequence 図は、Fig2.6 のように表される。

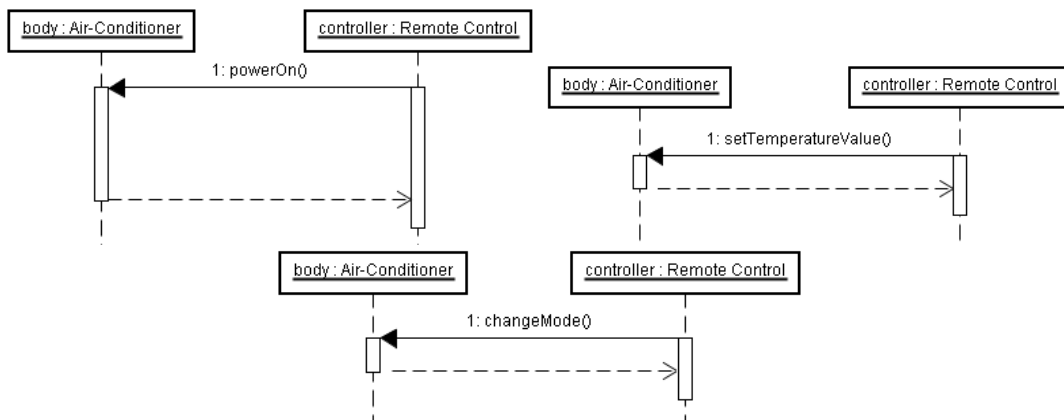


Fig 2.6: Sequence Diagrams Of Fig2.4

Fig2.6からわかるように、Sequence図のLifelineの名前は一般にComposite Structure図のPartの名前と同じになる。また、1つのComposite Structure図に対して複数のSequence図が存在する。これが、Sequence図が1つのModel図に対して任意の数存在する理由である。Fig2.6に示した例の場合、Fig2.4のSeq0,Seq1,Seq2にそれぞれFig2.6の各図が対応する。

このように、あるModelのSequence図は、このModelのComposite Structure図に示される部品間のInteractionを表現する。

なお、このリモコン付きエアコンModelは、以下で述べるエアコンとリモコンの協調関係を示すためのModelであるので、直接にはインスタンス化されない。

2.4.3 リモコン Model

次に、Fig2.5からは、2つのModelの存在が示唆されているとわかる。1つはエアコン本体、もう一つはリモコンである。そこで、新たなModel図が生成される。こうして生成されたりモコンのModel図をFig2.7に示す。

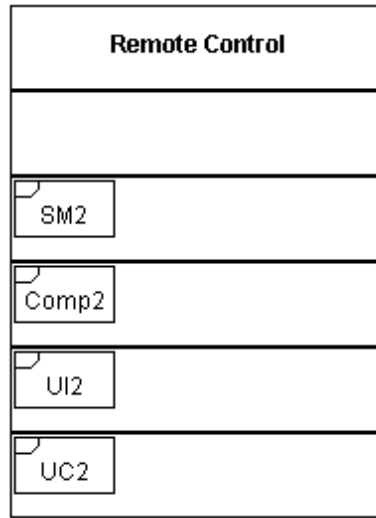


Fig 2.7: Model Diagram Of Remote Control

さらに、Fig2.7のComp2から、Fig2.8に示される Composite Structure 図がリンクされている。

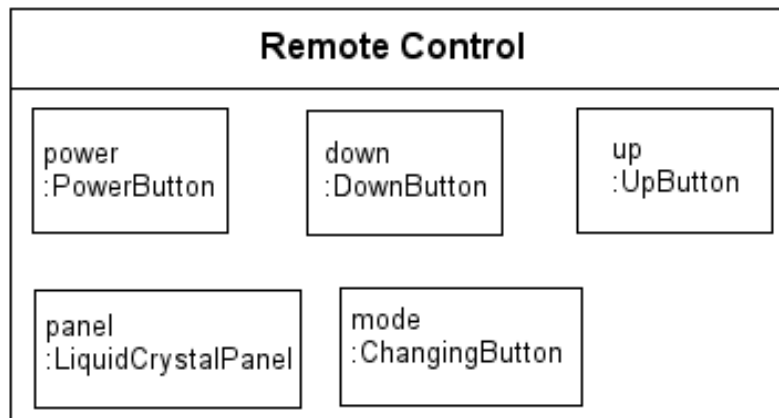


Fig 2.8: Composite Structure Of Remote Control

2.4.4 温度上昇ボタン Model

Fig2.8 には、5 つの新しい Model の存在が示唆されている。すなわち、電源ボタン、温度上昇ボタン、温度下降ボタン、モード切替ボタン、液晶パネルである。これらの Model 図もまた、自動的に生成できる。

Fig2.9 に、温度上昇ボタンの Model 図を示す。

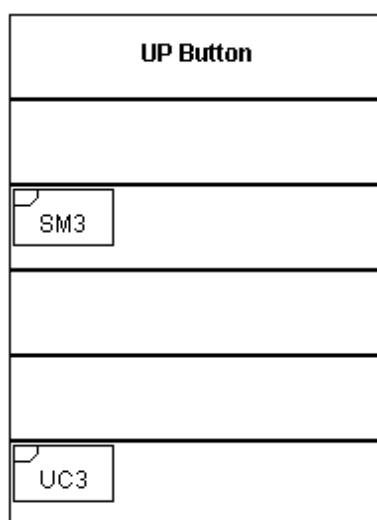


Fig 2.9: Model Diagram Of Up Button

必要のある場合には、Fig2.9 についても Composite Structure 図へのリンクを設定し、より小さな単位の設計を行うことができる。また、電源ボタン、温度下降ボタン、モード切替ボタン、液晶パネルについてもこのように Model 図を生成できる。

2.4.5 エアコン Model

次に、Fig2.10 に、エアコンの Model 図を示す。

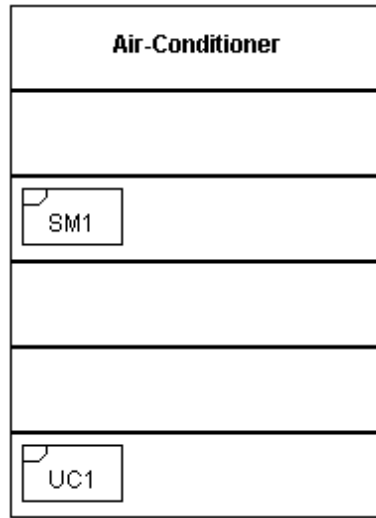


Fig 2.10: Model Diagram Of Air-Conditioner

Fig2.10 のフレームのリンク先や、これより下層に属する Model の存在については、冗長となるので説明を省く。

2.5 Model Instance

Model を Instantiate したものが Model Instance である。Model Instance は、別の Model の classifier の AttributeList(以下「Model の AttributeList」と省略する)に、属性として用いられることがある。

例えば、“Phone” という Model について、Phone の AttributeList には、“Push-Button” という Model の Instance、 b_1, b_2, \dots, b_{10} が存在する。“Push-Button” には、Integer 型の属性 num (このプッシュボタンを押されたときにダイヤルする番号)があるものとして、Table2.5 に、この AttributeList の例を示す。

Type	Name	Initialization
PushButton	b0	num:=0;
PushButton	b1	num:=1;
.	.	.
.	.	.
.	.	.
PushButton	b9	num:=9;

Table 2.1: AttributeList Of Model "PushButton"

Table2.1 の最右列 "Initialization" には、この Instance のコンストラクションの方法を指定する。2.1 で述べたように、Model が Instanciate されるためのコンストラクタは Model 自身が持つが、Initialization 列では、そのコンストラクタを用いて各 Instance を生成するための方法を指定する。

例えば、“PushButton” が次のようなコンストラクタを持つとする。

```
PushButton
  num := <Integer 値>
```

この場合、Table2.1 の b0 は、num に 0 を代入して生成される。特にコンストラクタの指定のない限り、Model は「<属性名> := <値>」の形で表現された Initialization 列をそのまま実行してコンストラクションを行う。

2.6 実行状態

UML2.0 には実行状態の概念がない。UML2.0 では、実装や実行状態に関するメタモデルは排除されており、モデリングにのみ着眼している。例えば、UML2.0 には Instance の仕様を定義するための InstanceSpecification というメタクラスは存在するが、UML1.5[UML1 03] に存在し、個々の Instance 自身を表していた Instance というメタクラスは存在しない。

昨年度実装された SMART0.2 は単なるモデリング・ツールとしての機能のほかに、その作成した Statechart や User Interface を実行する機能を有しているが、実行状態について明確な定義はされていなかった。以上から、SMART0.3 の機能を正確に表すには、Model の実行状態について独自に定義する必要がある。

本論文では、Model の実行状態とは、Model Instance が Action を実行できる状態にあることとする。この状態において初めて Model Instance は属性として具体的な値を持つことができる。なお、UML2.0 で、Property が具体的な値を持つことができないという点については次で述べる。

2.7 UML2.0 との比較

[UML2 03] では、Instance とメタクラス InstanceSpecification は次のように述べられている。

Instance

An entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations.

InstanceSpecification

An instance specification is a model element that represents an instance in a modeled system. An instance specification specifies existence of an entity in a modeled system and completely or partially describes the entity.

したがって、UML2.0 での Classifier の Instance の考え方とは、次のようなものであると考えられる。

$Instance = Classifier + InstanceSpecification + \langle \text{具体的な値} (identity, state) \rangle$

以上から、本論文でここまで述べてきた Model Instance は、実行状態が UML2.0 の Instance に対応し、非実行状態 (モデリング時に Instance の存在を指し示すために置いた Instance) が UML2.0 の InstanceSpecification に対応させることができる。

次に、[UML2 03] で、メタクラス Slot およびメタクラス Property は、次のように説明されている。

Slot

A slot specifies that an entity modeled by an instance specification has a value or values for a specific structural feature.

Property

A property is a structural feature. (中略) When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance.

Property には具体的な値を持つための属性が存在しないが、Slot によって具体的な値を入れるための枠が Instance に準備され、実行状態の Instance はここに具

体的な値を入れて Instance としての個性を持つことができる。ゆえに、UML2.0 において、Instance の持つ属性については、次のように考えていると思われる。

Attribute of Instance = Property + Slot + 〈具体的な値 (value)〉

2.6 で述べた、実行状態の Instance が持つ属性は、この Attribute of Instance に対応させられる。

第3章 User Interface

3.1 SMART0.2のUI Modelerの問題

SMART0.2のUser Interface Modelerでは、User Interfaceの物理的な外見(色や形)と、User Interfaceにユーザが何らかのアクションを起こしたときにソフトウェアに対してイベントを送る論理的な装置を、同一視してデザインしていた。例えば、リモコンを構成している「電源ボタン」というUser Interfaceは、ボタンの形状や色、ボタンの上に書かれている文字といった物理的な外見を持つと同時に、「ボタンを押されたらリモコンの制御部に対して『電源ボタンを押された』というイベントを送る」という論理的な働きも定義されている。

しかし、このUser Interfaceのモデリングの仕方では限られたUser Interfaceしか設計できない。例えば初期のテレビゲームのコントローラには、上下左右移動のための十字キーがあるが、これは外見的には1つのボタンである。しかし、表面のカバーを開ければ、そこにはそれぞれ上下左右の方向を表す4つのセンサがあり、これがゲーム機本体に対してボタン押下イベントを伝える。この場合、1つの外見に対して4つのアクションを持つ装置があり、SMART0.2のように物理的な外見と論理的な装置を1つのものとしてモデリングすることには限界があることがわかる。

UML2.0ではUser Interfaceは全く考えられておらず、我々は独自にSMARTのUser Interfaceモデリング機能を開発したが、この例のようにUser Interfaceのモデリングにはいくつかの未解決の問題点が存在する。3.2では、これらの問題について論じる。

3.2 User Interface モデリングの問題点

3.2.1 UI Model と UI Instance

あるModelを表すUser Interfaceのモデリング時、部品として貼り付けられるUser Interface部品は、Modelが実行状態にない限りはModelである。例えば、「リモコン」モデルのUser Interfaceに「電源ボタン」というUser Interfaceを部品と

して貼り付ける。このとき、「リモコン」は Instance ではないのだから、この貼り付けられた「電源ボタン」はまだ Model である。

Model から Instance が作られ、この Instance が実行状態になったとき初めて、貼り付けられている User Interface も Instance として機能するものになる。先の例では、「リモコン」の Instance として「リモコン 1」が作られてこれが実行状態になったときに、「電源ボタン」も Instance になる。Fig3.1 にその図を示す。

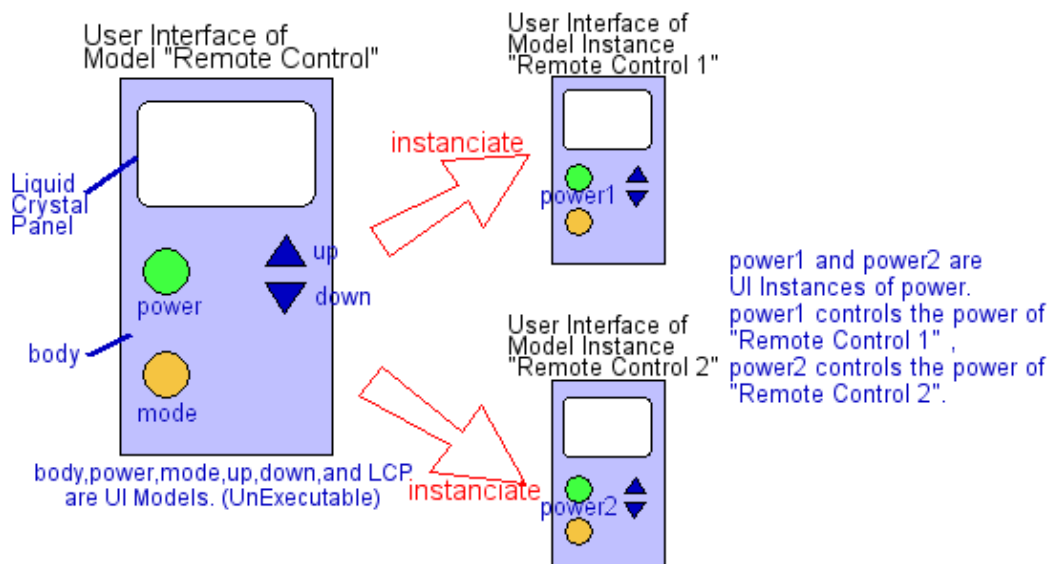


Fig 3.1: Instanciation Of Remote Control With UI

このように考えなければいけない理由は、1つの Model から作られる Instance は1つではないことにある。仮にモデリング時に貼り付けられた「電源ボタン」が Instance であるとすると、以下のいずれかのように考えなければならない。

1. 「リモコン」の Instance 「リモコン 1」と「リモコン 2」を作ったときに、この Instance 「電源ボタン」を共有する。
2. Instance 「電源ボタン」の Instance というものを2つ作り「リモコン 1」と「リモコン 2」に属させる。

ところが、1では「電源ボタン」を押したときの動作を「リモコン 1」と「リモコン 2」で区別することができず、2はオブジェクト指向の考え方に合致しない。

したがって、「電源ボタン」は Instance ではなく UI 部品の Model でなければならない。

このように考えた場合に、「リモコン」の User Interface の Composite Structure 図を描く Fig3.2 のようになる。なお、Fig3.2 には、3.2.3 で述べる User Interface の継承関係についても、Button の例を用いて示した。

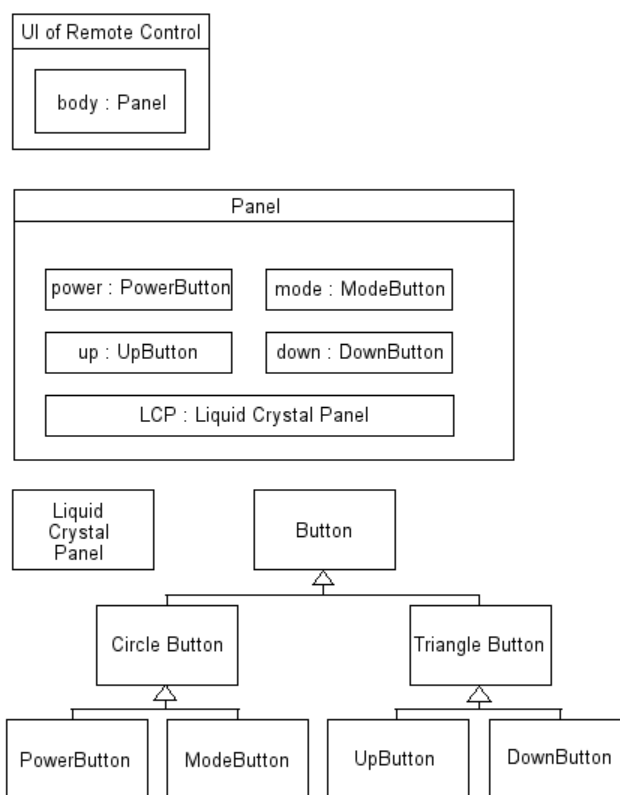


Fig 3.2: Composite Structure Of User Interface Of Remote Control and Buttons Hierarchy As User Interface Parts

本論文では、Model の User Interface に貼り付けられた User Interface 部品を UI Model と呼び、Model が Instance 化されたときに同様に Instance 化され実行可能状態になった User Interface 部品を UI Instance と呼ぶことにする。

UI Model と UI Instance は、基本的な外見は同じである。基本的な外見とは、User Interface 部品としての色や形状、大きさなどを指す。一方で、実行時にしか決められない外見もある。例えば、液晶パネルに表示する文字列は、UI Model の

段階では決めることができない。電源オン時と電源オフ時では色が変わるようなボタンの色も、実行時にしか決められない外見である。

また、UI Instance は UI Model によって定められた動作をするべきであるが、動作対象は UI Instance ごとに個別である。例を挙げると、Model 「リモコン」に貼り付けられた「電源ボタン」は「リモコンに対して電源が ON/OFF されたというイベントを発行する」という働きを持つが、「電源ボタン」が Instanciate されたとき、「リモコン 1」に属する「電源ボタン 1」は「リモコン 1 に対して」動作するべきである。

3.2.2 UIのプロトタイプ

3.2.1 で述べたように、UI Model と UI Instance は、外見についても動作についても完全に同じであるとはいえないが、一方で、UI Model に少しの変更を加えることで UI Instance の個性を決めることができることも事実である。SMART0.2 はクラス概念を持たない “Instance Editor” であったと結論付けられるが、この考えに従えば、SMART0.2 の UI Modeler では、“TextButton” や “TextLabel” といった UI Model が最初から存在しており、User Interface デザイン時には、この UI Model に表示文字列や背景色といったパラメータを与えて UI Instance を作り、これを User Interface 部品として貼り付けていたといえる。

以上より、UI Model と UI Instance の関係とは、UI Model は UI Instance の一種のテンプレートであり、これにパラメータを与えて UI Instance を生成する、プロトタイプと具体物の関係であるといえる。

3.2.3 User Interface の継承関係

User Interface にも継承関係が存在すると考えられる。例えば、「ボタン」という UI Model から、色をそれぞれ白と青にした「白ボタン」「青ボタン」というような “UI Sub Model” が考えられる。

一方で、これらの「白ボタン」や「青ボタン」は 3.2.2 で述べたような、プロトタイプにパラメータを与えて作ることもできる。元の UI Model からの変更が大きい場合は継承関係を用いるほうが適切だと考えられる。

3.2.4 論理部分と外見部分

3.1 で十字キーの例を用いて述べたように、SMART0.2 の User Interface では、外見的な部品としては 1 つであっても、論理的な動作がいくつか定義されるよう

な User Interface 部品を的確にモデリングできない。これを解決するには、「見た」と「動作」の2つを分けて定義する必要がある。

例として、3.1 で述べたゲームのコントローラーの十字キーを、Fig3.3 に図式して説明する。

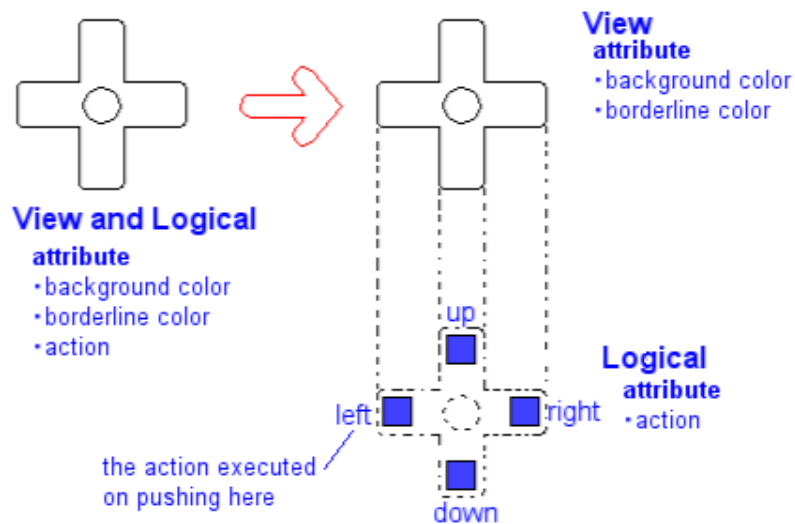


Fig 3.3: “All-in-one” User Interface And Separated User Interface

Fig3.3 において、左は SMART0.2 で実装されていた形式の User Interface であり、外見の情報（背景色・ボーダーライン色）と、この十字キーを押されたときの動作に関する情報（アクション）とが一つになっている。しかし、これでは 3.1 で述べたような問題が発生する。したがって、Fig3.3 の右に示したように、外見を表す部分と論理的な動作を定義する部分を別々にモデリングする必要がある。

第4章 SMART UML Modeler

SMART0.3のUML Modelerでは、SMART0.2におけるStatechart ModelerおよびUser Interface Modelerを、第2章で述べたModel図によって統合してクラス概念を導入し、さらに実行時のログをSequence図でトレースする機能を追加した。この章では、第2章と第3章で説明した概念をSMART0.3でどのように実装したかについて述べる。

4.1 Modelの実装

SMART0.3では、ModelとModel Instanceは同じクラスとして実装した。Modelクラスには、これがModelかModel Instanceかを示すbool値があり、これによってModelとModel Instanceを切り分けている。このように実装したのは、Modelと、そのModelのModel Instanceは同じ構造をしているからである。[UML2 03]には、Classifierについて”A classifier is a classification of instance. it describes a set of instances that have features in common.”と述べられており、この実装と合致することがわかる。

Modelから実行状態のModel Instanceを生成するときは、Modelのすべての構造をコピーし、Instance名をこのInstanceに与える。さらに、Modelの持つ属性に与えられたデフォルト値をその属性の値として代入する。つまり、SMART0.3では、Instanceを基本として構造を作っており、Modelによって表されるClassifierは、このInstanceの属性に値が入っていない状態だということができる。

4.2 User Interface

第3章で述べたように、SMART0.2のUser Interface Modelerの実装では様々なUser Interfaceのモデリングに対応しきれず、また、User Interfaceのモデリングと実行には解決の難しいいくつかの問題点がある。したがって、SMART0.3のUser Interfaceモデリング機能は、SMART0.2からあるべきUser Interface Modelerの実装に到達する中間段階のものとして実装することにした。

SMART0.2では、TextLabel、TextButton、ImageLabel、ImageButton、TransparentButtonの5つのUser Interface部品が用意されており、これらに表示文字列・表示画像や背景色・前景色、アクションファイルといった設定項目がある。これらの設定項目をユーザが指定することで、それぞれのUser Interfaceの個性を決めていた。

SMART0.3では、TextLabelのようなUser Interface部品はすべて最初から用意されている作り付けのModelであると考えた。この作り付けのModelは、例えばTextLabelであれば背景色・前景色・ボーダーライン色・表示文字列といった属性を持っている。ユーザがUser Interfaceをデザインする際には、3.2.3で述べたような継承関係を用いて、この作り付けのModelを継承した別のModel(Sub Modelと呼ぶことにする)を作る。このSub Modelでは、親Modelの各属性に具体的な値を入れておき、それによってこのUser Interface部品の個性が決まる。

ただし、3.2.4で述べたような、論理部分と外見部分の切り分けについては本年度は実装していない。これについては今後、より適切な概念を定義し、その上で実装すべき課題である。

4.3 実行状態

ユーザが、Modelを実行状態にするボタンを押すと、一旦すべてのInstanceを作り直す。これは、先に名前だけ存在していたInstanceに、ここまでのモデリングの結果を反映させるためである。

各Instanceの状態は、モデリングをしていたウィンドウとは別のウィンドウ(Instance Viewer)に表示される。SMART0.2にはClassがなく、モデリング対象がInstanceそのものであったため同じウィンドウにInstanceの実行状態を表示していたが、SMART0.3では1つのModelに対して複数のInstanceが存在しうるため、このような実装とした。

また、SMART0.3では、2.7で述べたように、ModelのPropertyとInstanceのSlotを同じクラスとして実装した。これは、実行状態を表現するために行った、UML2.0の拡張である。

4.4 ログの取得と解析

システムを実行すると、その実行状態を実行後に確認したいときがある。例えば、あるオブジェクトのある時点での属性の値を再現できれば、バグの発見に役立てることができる。

従来のデバッガのトレース機能は、エラーが発生したときにその場で実行を止め、そのエラーを再度発生させることでエラー発生までをトレースする。たとえば、Java 統合開発環境ソフトの Eclipse(<http://www.eclipse.org>) では、デバッグモードを用いるとエラーが起きた時点で実行を止める。その上で、そこまでのメソッド呼び出しの過程をリストする。このリストは、再現したい時点をクリックすれば再度そのメソッドからエラーまでを実行してくれる。

このようなことが可能なのは、一旦起こったエラーを再現できるからであるが、システムの中にはエラーの再現に困難を極めるものもある。たとえば、並列処理の場合がそれに当たる。したがって、実行中に常に何らかの形で実行ログを記録し、再現が不可能であってもエラーが起きたときの状況を確認できるようなデバッガが必要である。こういった考え方のデバッガは、replay/record debugger と呼ばれている。

以上から、SMART0.3 では、実行状態の記録をログとして保存し、それを Sequence 図として表示することを考えた。ログは次のような形式を用いた。

〈ログの種類〉: 〈ログの詳細〉

ログの種類としては、Table4.1 に示す名前で定義した。

message-send- <code>{call,return}</code> -event	他のオブジェクトの event を発生させる
message-send- <code>{call,return}</code> -activity	他のオブジェクトの持つ activity を起動する
event-activity-execute- <code>{begin,end}</code>	状態遷移中に effect として指定された activity の終了
transition- <code>{begin,end}</code>	状態遷移の開始・終了
<code>{begin,end}</code> -testcase	テストケースの開始・終了
write-attribute	属性の値の書き換え
read-testcase	属性の値の読み取り
setcurrent-state	オブジェクトの状態をセットする
assert- <code>{equals,true}</code>	アサーションが実行されたことを示す
iscurrent-state	
failed-by-meerror	Test 実行中の MissingElementError の発生

Table 4.1: Kinds Of Log

以上のような形で記録されたログの例を次に示す。

```
begin-testcase:testsuiteroot\温度\testcase1,2004/1/22/13:28
```

```
write-attribute:controller. 設定温度,controller. 設定温度 [0],24,24,24,  
testsuiteroot\温度\testcase1,testsuiteroot\温度\testcase1  
setcurrent-state:aircon. 電源 on,testcase1  
setcurrent-state:controller. 電源 on,testcase1  
message-send-call-event:push 上昇ボタン,testsuiteroot\温度\testcase1,  
testcase1,controller, []  
transition-begin:controller.transition1  
transition-end:controller.transition1  
message-send-return-event:push 上昇ボタン,testsuiteroot\温度\testcase1,  
testcase1,controller  
assert-equals:25,controller. 設定温度 [0],25,24,false
```

Fig4.1 に、このログを Sequence 図として出力したものを示す。

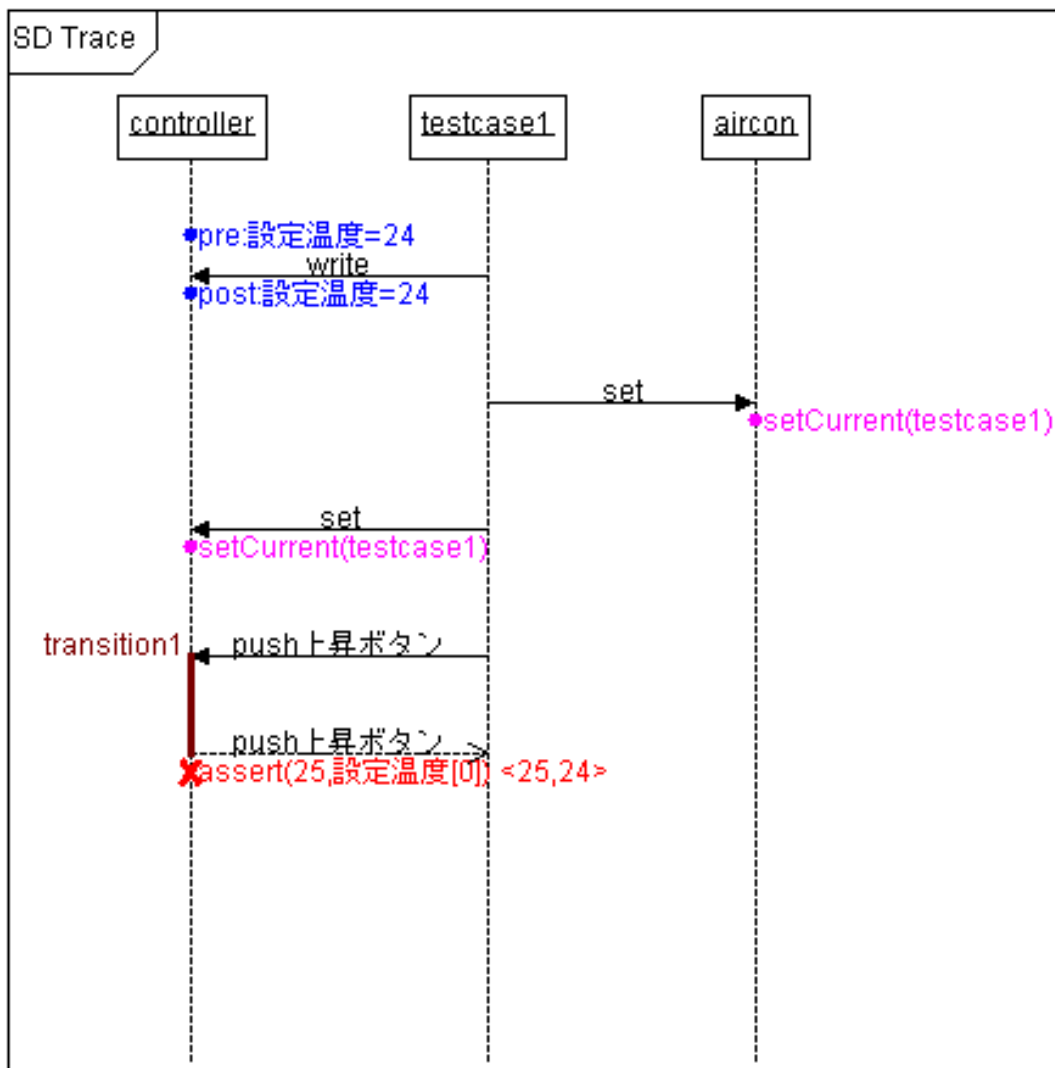


Fig 4.1: Sequence Diagram As Trace

Sequence 図によるトレースは、実行の様相を実行後に視覚的に確認するうえで有効である。例えば、Fig4.4 で、controller の属性「設定温度」がどこで書き換えられているのかはこの図から直ちに理解することができる。

4.5 Composite Structure のチェック機能

SMART0.3 では、Composite Structure 図をユーザが直接描くこともできるが、モデリングを進めていくうちに、Composite Structure 図に描いていない通信をモデリングしてしまう可能性がある。そこで、SMART0.3 では、4.4 で述べたログを

利用して、Composite Structure が正しくモデリングされているかどうかを確認し、誤りがある場合には修正候補を提示するようにした。

以下に例を挙げて示す。最初に、ユーザが Fig4.2 に示すような Collaboration を持つ Air-Conditioner with Remote Control の Model を作ったとする。

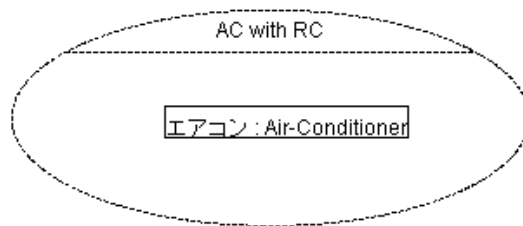


Fig 4.2: Composite Structure Of Air-Conditioner With RemoteControl

ところが、この Fig4.2 では、協調動作するはずの RemoteControl を書き忘れて
いる。この状態で、RemoteControl が Fig4.3 に示す StateMachine を持つとする。

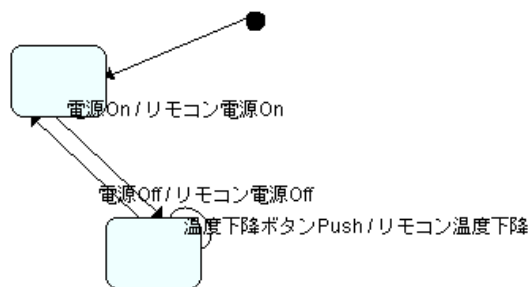


Fig 4.3: StateMachine Of RemoteControl

ここで、effect の「リモコン温度上昇」が次のような activity であるとする。

```
begin
controller. 設定温度 [0] := controller. 設定温度 [0]+1;
callEvent body. 温度設定 (controller. 設定温度 [0])
end
```

controller は RemoteControl のインスタンス、body は Air-Conditioner のインスタンスである。この activity から、RemoteControl と Air-Conditioner 間に通信があることがわかる。このような StateMachine が実行されたとき、次のようなログが記録される。

```
write-attribute:body. 設定温度, body. 設定温度 [0], 22, 21, 22, controller
```

ユーザがボタンを押したときこのログを解析し、その結果 Fig4.2 の Composite Structure が不十分であることを発見する。その後 Fig4.4 に示すような Composition Error と、その改善策の候補を表示する。

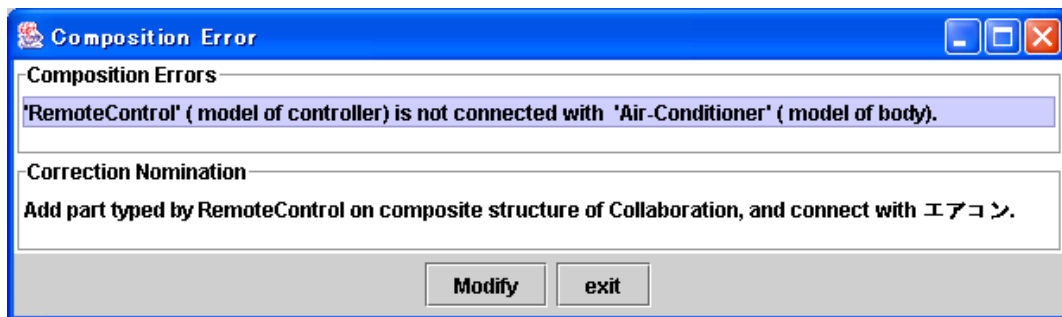


Fig 4.4: Composition Errors And Modification Nomination For Fig4.2

第5章 結論および今後の展開

5.1 結論

以上に述べてきたように、本研究では UML2.0 を基本として、User Interface も含めた Classifier の表現として Model を提案し、SMART0.3 の UML Modeler として実装した。さらに、User Interface のモデリングに関する諸問題を提起し、また、実行状態のログを記録したうえでそれをモデリングに生かすことを考え、それを実現するためのツールを作成した。

以下では、本年度の成果を踏まえて、コンポーネントベースの開発手法 Catalysis において、本論文で考えてきた Model をどのように生かせるかを考える。また、SMART0.3 において概念や実装の面で改良すべき点を述べ、今後の指針としたい。

5.2 モデリングの方法論

5.2.1 Component-Based Development - Catalysis -

Catalysis は、オブジェクト指向開発方法論の一種であり、UML1.X をやや拡張した表記法を用いている。この方法論を発案した人々が、Catalysis を考え出す中で定義した様々な概念の中には、UML2.0 のセマンティクスに影響を与えているものも多い。具体的には、Component、Collaboration、Refinement などがそれに当たるが、ここでは詳細を述べないこととする。Catalysis については [Souz 98] に詳しいほか、本論文の付録 B でもその概要を説明した。

5.2.2 Refinement と Model

本論文で述べてきた Model による構造的モデリングは、Catalysis における Type の Refinement の一種であると位置づけられる。たとえば、2.4 で用いた「リモコンつきエアコン」では、最初に「リモコンつきエアコン」という大まかな Type を決定し、その後詳細を徐々に決めていくという形を取った。その一方で、コンポーネントベースの開発において中心的な概念の 1 つである Action の Refinement

に関しては、現在の Model ではサポートしきれていない。[Souz 98] で “Normally we zoom actions and objects at the same time.” と述べられているように、Model 図において Classifier の Refinement を行ったときに、元となっている Classifier の持つ Sequence 図や StateMachine 図などが示す Behavior が、Refinement 結果の Classifier においてどのような Behavior に refine されているかをユーザが理解することを助けるツールが必要だろう。

5.3 ログに関する考察

4.4 で述べたように、ログを記録しそれをトレースすることは、バグの発見や実行状態の理解に役立つ。ここでは、ログについて、ログの記録形式の問題及びその活用方法について論じる。

5.3.1 ログの形式

実行状態のログは確かに有用ではあるが、実行時のあらゆる情報を逐一記録していくことは、記憶領域や実行速度の問題から現実的に無理であるし、また、あとでこの記録を読み出すときに複雑な処理を要する。したがって、どのような情報を記録していくかが重要なポイントとなる。

SMART0.3 では、4.4 に示したログの例のように、ごく簡単なテキストのみで、Sequence 図としてトレースすることが可能となった。このように簡単な実装が可能となったのは、その大半を UML2.0 のメタモデルに沿ったログを記録したからである。例えば、Transition や Activity などは、UML2.0 中でメタクラスとして定義されており、それらの構造や意味論を定義することにより、モデルの実行の意味論が中傷的に定義されているので、それに沿って記録をとれば比較的容易に情報を分析できる。ただし、2.6 で述べたように、UML2.0 には実行状態やインスタンスについては定義していないので、今後システムの実行状態を含めたメタモデルを作り、それに沿ってログを記録するべきである。

さらに、現在は単純なテキストとなっているログの形式を、XML(eXtensible Markup Language) 形式に変更することで、より形式的にログを残すことが必要である。

5.3.2 ログの活用

4.5 で述べた、ログによる Composite Structure のチェック機能は、ユーザが見つけていなかった不整合に対して、コンピュータがそれを suggest している。この方法では、不足していた要素をコンピュータが見つけたときに、それを強制的に作るのではなく、ユーザに一旦その要素を追加するかどうかを確認させる方式を取っている。XP(eXtreme Programming) ではプログラマ二人によりプログラミングを進める「ペアプログラミング」の実践を勧めているが、この言葉を借りれば、Composite Structure のチェック機能はいわばユーザとコンピュータの「ペアモデリング」であるといえる。

この方式では、実行結果を頼りにして、その実行結果すべてに該当するようなモデリング結果を探り出そうとする。つまり、「実行」という実験を元にして、この実験結果に不整合が起こらないようなモデリングをコンピュータが suggest する。実験結果が少ない場合、たしかにこの方式では要求に対するモデリングの漏れが生じる可能性もあるが、実験結果が大量であれば、それらの実験結果は実行結果に関するデータベースとしてこのように活用する価値がある。これは、一種のデータマイニングを利用したモデリングであるともいえるだろう。

今後の課題として、このログをさらにどのように活用することができるかを探り、それによるモデリングを考案していくことが必要である。

5.4 SMART の改良

SMART0.3 において、今後改良が必要と思われる点を以下に述べる。

5.4.1 User Interface の適切な実装

SMART0.3 では、3.2.4 で述べた User Interface の外見と動作の切り分けの問題や、3.2.3 で述べた継承関係の問題について、いずれも不完全な形で実装を行っている。User Interface の問題に関してはまだ問題自体の分析が不完全であるように考えられる。様々な User Interface のモデリングに対応していくには、これらの問題を分析した上で実装を行っていくべきである。

5.4.2 Instanciation 時の activity

Model Instance を作る際には、元となる Model Instance をコピーするが、このとき、Model の StateMachine に属する Transition の effect として、SMART Action

Language(SAL) で書かれた activity もコピーされる。ところが、この activity を記述するとき Instance の名前が SAL プログラム中に書かれることがあり、モデリングの時点で Instance の名前を知っていなければならない。

例えば、Model「リモコン」に Fig5.1 のような StateMachine が存在するとする。

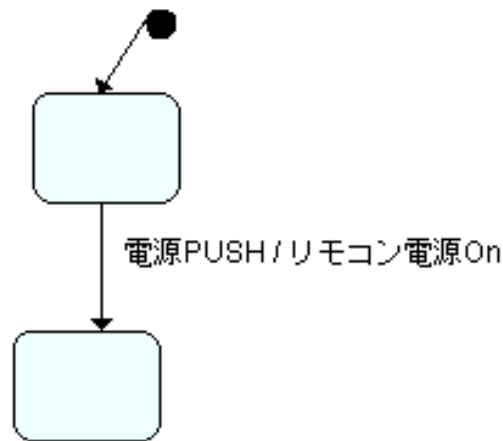


Fig 5.1: StateMachine Of Remote Control

ここで、Fig5.1 の Transition に付いている activity 「リモコン電源 On」は以下のようなものであるとする。

```
begin
controller. 設定温度 [0] := 18
callEvent aircon.power1
end
```

controller は「リモコン」の Instance、aircon は「エアコン」の Instance であるが、現在の SMART ではこのようにモデリングの段階で Instance の名前を activity 中に用いるしかない。将来的には、モデリングの段階で SAL 中に controller や aircon と書いていたところに、Instanciation 時に Instance ごとに適切な Instance 名を入れられるように改良していく必要がある。

5.4.3 Instance のコンストラクション

SMART0.3 の Instance のコンストラクションでは、2.5 の最後に述べたような「特にコンストラクタの指定がない」場合についてのみ実装している。つまり、2.1

で述べたようなコンストラクタを Model にまだ実装していない。今後、より複雑なモデリングのためには、コンストラクタをユーザが指定できるように実装し、さらに、Instance が別の Model の属性として用いられている場合にはそのコンストラクタが実行されるように改良を行っていくべきである。

5.4.4 Composite Structure 図と Sequence 図の関係

4.5 で述べたように、システムの実行後に記録したログを解析することで正しい Composite Structure が描かれているかをチェックできるようにした。より有用に Composite Structure を用いるには、4.4 で描いたような Sequence 図と Composite Structure 図の関係をj用いることができるようになることが望ましい。というのも、A.4 で述べるように Composite Structure を refine することによって Sequence 図が生成できるからである。このような関係を林研究室で潘が行っている Traceability によって明示することができれば、Composite Structure 図は 付録 B で論じる Catalysis に示された、次のような refinement の過程で価値があるものとなる。

1. UseCase(pre/postcondition を持つ)
2. UseCase の pre/postcondition を満たす abstract action ができる
3. abstract action を満たす collaboration が生成できる。このとき、collaboration 中に描かれた connector は、abstract action を分解した action を表す
4. connector に合う Sequence 図を描く

5.4.5 より複雑な Composite Structure の記述と実行

SMART0.3 では、Composite Structure の part の多重度が 1 であることを前提として実装を行った。つまり、SMART0.3 では、Fig5.2 のようなモデリングをサポートしていない。

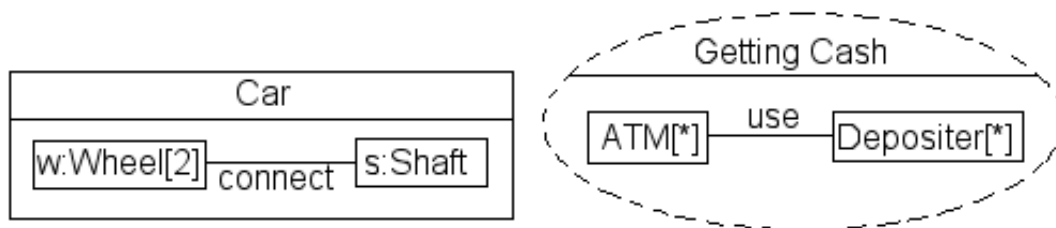


Fig 5.2: Unsupported Composite Structures : Using Multiplicity

ここで、[] 内に示しているのが多重度である。Car の例では、Shaft1 つに対して2 つの Wheel が存在することを示しており、Getting Cash の例では、任意の数存在する ATM に対して、任意の数の預金者がこれを利用することを示している。このようなモデリングをサポートすることで、同じ Model の Instance が複数存在する場合の並行動作をより明確に記述することができるようになる。

5.4.6 多機能先行モデリング

ビジネスシステムとは違い、家電などの組み込みシステムでは、たくさんの機能を持った「高機能版」を先に作りこみ、そのあとで価格を抑えるために機能を削り「廉価版」を作るようなモデリングがされることがある。この場合、「多機能版」モデルと「廉価版」モデルの関係をどのように定義するべきかは未知数である。

通常、オブジェクト指向型開発では、より abstract な Super Class を先に作る。その後この Super Class を特化させた Sub Class を作る。Sub Class は普通 Super Class よりも作りこまれ、多くの機能を持っていることを考えると、先に述べた組み込みシステムのモデリングはこの逆のアプローチを取っているといえる。組み込みシステムをターゲットとするシステムを目指すならば、このような逆方向のモデリングをサポートしていくことが必要である。

5.4.7 非同期メッセージのサポート

Fig2.6 にあるように、SMART0.3 の Sequence 図では、非同期メッセージをまだサポートしていない。しかし、これは同研究室で清水が実装を行っている concurrent SAL が実装された時点でサポートされる予定である。

謝辞

初めに、筆者が本研究を行うに当たり、理論から応用のための議論に至るまで、大変熱心なご指導、ご教授をいただきました神戸大学工学部情報知能工学科 林晋教授に心から感謝いたします。また、共同研究者である神戸大学自然科学研究科 潘沂冰氏、SMARTの実装において多くのご助力をいただきました、神戸大学自然科学研究科 森健司氏 及び 薛世宗氏をはじめ、平素より様々な面でお世話になりました林研究室のメンバーに感謝いたします。

参考文献

- [UML2 03] Object Management Group, “03-08-02.pdf”, <http://www.omg.org>(2003)
- [UML1 03] Object Management Group, “03-03-01.pdf”, <http://www.omg.org>(2003)
- [Souz 98] Desmond Francis D’Souza and Alan Cameron Wills, “Objects, Components, and Frameworks With Uml: The Catalysis Approach”, Addison-Wesley(1998),pp.3-44,pp.153-183,pp.213-272
- [Mori 03] 森 健司, “UML を用いた分散システム・モデリングツール”, 卒業論文, 神戸大学 (2003)
- [Wada 03] 和田 周, “UML2.0 - 新仕様の読み方 -”, The 2nd MDA Technology Forum(MTF)(2003)
- [Yama 02] 山田 正樹, “UML2.0 の最新情報”,<http://www.metabolics.co.jp/00Technology/UML2.html>(2001)
- [Yama 03] 山田 正樹, “開発プロセスから見た UML2.0 の意味”, The 2nd MDA Technology Forum(MTF)(2003)

付録 A UML2.0

UML2.0 は 2003 年 6 月にドラフト版として策定されたが、現在最終的な整合性チェックなどを行っており、正式なリリースは 2004 年度中と言われている。ここでは、UML2.0 について、UML2.0 の特徴及び本論文で用いているところを中心として説明する。なお、UML2.0 の解説としては [Wada 03]、[Yama 03] 及び [Yama 02] を参考にした。

A.1 UML2.0 の構成

UML2.0 は以下の 4 つから成り立っている。

- Infrastructure (下部構造)
- Superstructure (上部構造)
- Object Constraint Language (OCL)
- Diagram Interchange

Infrastructure は主に OMG の Meta Object Facility(MOF) との整合性を取るためのものであり、UML で用いられる記法に関する部分は Superstructure に記述されている。

A.2 図の種類

UML2.0 には、TableA.1 に示す 13 の図が定義されている。なお、太字で示したものは UML2.0 になって導入されたもの、下線で示したものは、概念的には UML1.X で存在していたが UML2.0 になって名称を変更されたものである。

Structure 図	Behavior 図
Class 図	Sequence 図
Object 図	<u>Communication 図</u>
Package 図	Interaction Overview 図
Composite Structure 図	Timing 図
Component 図	<u>State Machine 図</u>
Deployment 図	<u>Activity 図</u>
	<u>UseCase 図</u>

Table A.1: UML2.0 Diagrams

A.3 UML2.0の特徴

MOF との整合性 UML1.X では、MOF との不整合が問題とされていたが、Infrastructure の導入によりこの問題は対処された。

実装の分離 UML1.X では、メタクラスの中に実装に関するクラスが多く混ざっていた。たとえば、Operation というクラスは、その実装として Method というメタクラスとの関連を持っていた。しかし、UML2.0 では Method のような実装に関するメタクラスはすべて排除され、モデリングのための言語としての色彩を強めている。

パッケージのマージ パッケージのマージという強力な概念が初めて導入された。パッケージのマージとは、あるパッケージが他のパッケージを拡張するという概念である。FigA.1 のように表記し、その意味は FigA.2 に示したようなものになる。なお、この2つの図は [UML2 03] から引用した。

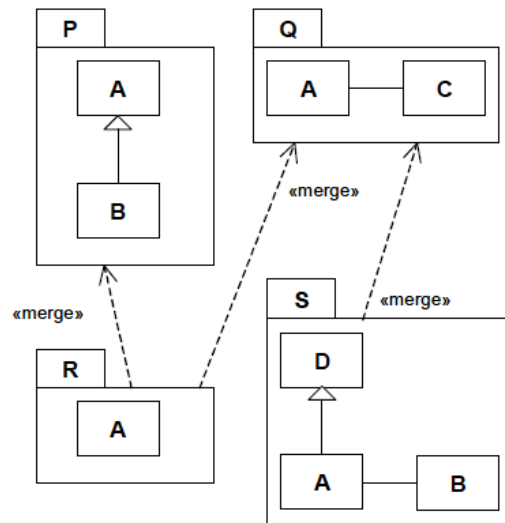


Fig A.1: Package Merging : Notation

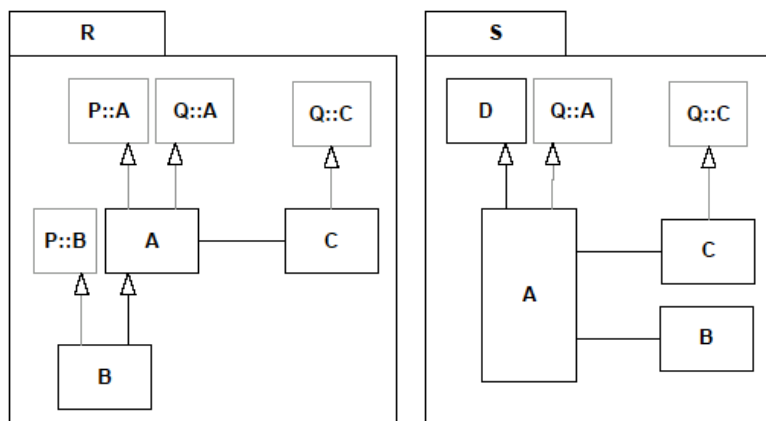


Fig A.2: Package Merging : Result

Precondition/PostCondition Behavior と Operation に対して Precondition 及び Postcondition が Constraint の形で張り付くようになった。Precondition とはこの Behavior や Operation が開始される時に満たされていない条件のこと、Postcondition とは Behavior または Operation の終了後に満たされていない条件を指す。この condition の導入により、Operation や Behavior

が正しく動作しているかどうかを検証する道筋ができたといえる。なお、Behavior は Activity、StateMachine、Interaction のスーパークラスとして UML2.0 になって初めて導入されたクラスであり、振る舞いに関するメタクラスが整理されたことがわかる。

Sequence 図の記述力強化 UML2.0 では、Sequence 図の記述にフレームと呼ばれる枠を使うアイデアが採用されている。これにより、UML1.X ではできなかった、既存の Sequence 図の使い回しや If 文要素の記述などが可能になっている。

FigA.3 に例を示す。この図では、“example” という Sequence 図の中で、reference(ref) というフレームを用いて他の Sequence 図 “sequence D” を参照し、さらに alternation(alt) というフレームを用いてその中に if 文に該当する Sequence 図を描いている。

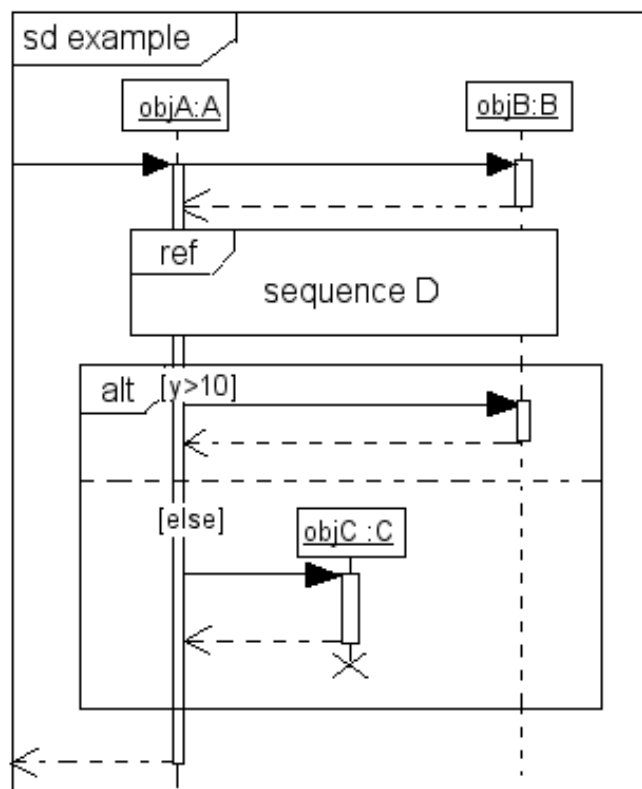


Fig A.3: Sequence Diagram with reference(ref) and alternation(alt)

なお、UML1.X で Sequence 図と相互変換できる立場にあった Collaboration 図 (UML2.0 では Communication 図) は、記法に大きな変化はなく、また、Interaction を表す標準的な記法は Sequence 図であることが明記されているため、Collaboration 図は Interaction の記法としてはややマイナーな位置づけになったといえる。

時間概念 UML に初めて時間概念が導入され、Interaction 図のうち、Sequence 図、Interaction Overview 図、Timing 図で時間制約を記述できるようになった。Interaction Overview 図を用いて、時間制約の記述例を FigA.4 に示す。

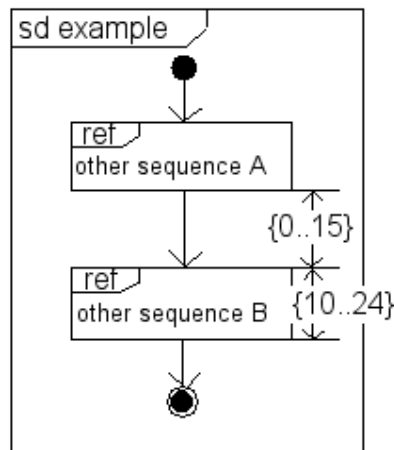


Fig A.4: Notation Of Time

Composite Structure クラスの内部構造や Collaboration のために導入された概念である。これに類するものは UML1.X にも Composite Object という形で存在していたが、Catalysis 等の影響を受け、Collaboration を含んで、より前面に押し出されてきた。これに関しては本論文に特に関係するので A.4 に詳しく述べる。

Activity の位置づけ UML1.X では、Activity は StateMachine の一部だったが、UML2.0 では Activity は Behavior の 1 つとして StateMachine と同等の位置づけになっている。したがって、UML2.0 の Activity 図では StateMachine 図によらない表記が可能になった。

A.4 Composite Structure

構造を持つ Classifier の内部構造を記述するために、UML2.0 では新たに Composite Structure 図が導入された。これにより、Classifier の内部にどのような部品 (Part) が存在し、それらが互いにどう協調 (Collaboration) してこの Classifier を構成しているかを表現できるようになった。この概念は、UML1.X の Composite Object を、より明確にしたものだと考えられる。

また、UML1.X で使われていた “Collaboration” という言葉が Composite Structure 図で使われるようになった。つまり、“Collaboration” という言葉は、UML1.X のように動的振る舞いを指すのではなく、オブジェクトどうしの協調関係を静的に表すものとなり、具体的な動的振る舞いについては Sequence 図などの Interaction 図で表現する。

記法

Internal Structure FigA.5 に、Composite Structure 図によってクラスの内部構造を表した例を示す。

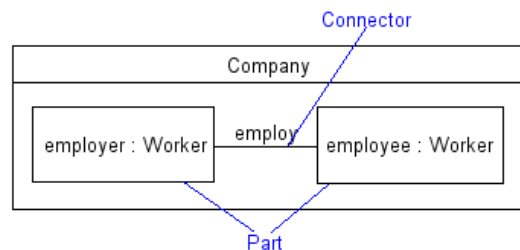


Fig A.5: Composite Structure Diagram : Class

ここでは、別に Worker クラスが存在し、その Worker がそれぞれ employer と employee としてこの構造に加わっている。ここで、employer や employee はオブジェクトを示しているのではなく、この構造中での役割 (Role) を表現した名前であることを注意が必要である。この記法上、employer と employee を Part と呼び、employer と employee の間の線を Connector と呼ぶ。Connector には employ という名前がついているが、これはオブジェクト間の特定の Message を示すものではなく、employer と employee の関係性を示している。

例えば、この図を refine した場合に FigA.6 のような Sequence 図を生成できる。

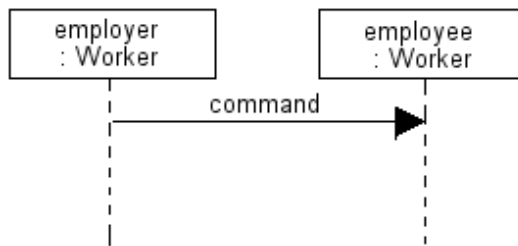


Fig A.6: Refinement Of FigA.5

FigA.6において“command”は具体的な Message であるが、これは FigA.5 での employ という Connector を通して送られる。このとき、command は employ という関係に即したものである必要がある。

Collaboration FigA.7 に、Collaboration を Composite Structure 図によって示した例を挙げる。

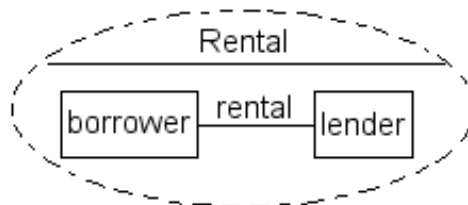


Fig A.7: UML Composite Structure Diagram : Collaboration

Collaboration で表現しようとするのは、あるクラスとあるクラスの関係であり、Collaboration は直接にインスタンス化されなくてもよい。つまり FigA.7 では、borrower と lender の関係 rental を示したいのであって、Rental というクラスが存在しなくてもよい。また、Collaboration を描くときには Borrower や Lender の内部構造には着目しない。

さらに、Collaboration を用いて他の Collaboration を表現することもできる。

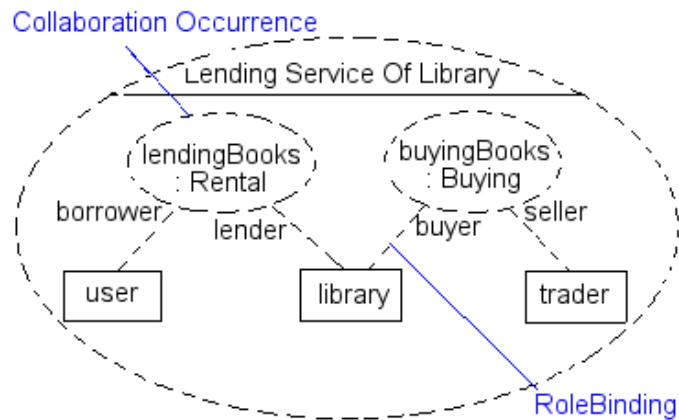


Fig A.8: Collaboration Using Other Collaboration

FigA.8 では、他で定義されている Rental や Buying といった Collaboration を Collaboration Occurrence という形で用いている。Rental は FigA.7 で示した Collaboration と同一である。破線で表した RoleBinding の脇には、元の Collaboration の part 名を Role として表示する。これは、RoleBinding を用いて Collaboration Occurrence とつながっている part の役割を示す。例えば、FigA.8 において、user という part は Rental という Collaboration の borrower に当たる。

A.5 Component

UML1.X にもコンポーネントの概念やコンポーネント図は存在していたが、Composite Structure と組み合わせることで強力にコンポーネントを描けるようになった。インタフェースは 2 種類に分かれ (requiredInterface と providedInterface)、このインタフェースによってコンポーネント外的な仕様を描き、Composite Structure を用いてコンポーネントの内部構造を描くことができる。コンポーネントとは、クラスの特化した形だと言えるが、この概念がどのように使えるものであるかについては定まった考え方がなく、今はまだ実験の段階だといえる。

付録B Catalysis

Catalysis は、UML を用いたオブジェクト指向・コンポーネント基盤開発方法論であり、その内容は [Souz 98] に述べられている。以下では、その Catalysis に関して概要を述べる。なお、図に関しては、UML1.X 及び UML2.0 には存在しない、Catalysis 特有の記法を用いているものも多いので、それらは B.6 でまとめて述べ、本文中では特に説明はしない。

B.1 Basic Concept

Catalysis は、Object と Action という 2 つの概念を基本にし、これを “Zoom-in/out” することが基本的なアイデアの 1 つになっている。

Object and Action これは特に Catalysis での新しい概念ではなく、一般的なオブジェクト指向開発で用いている意味と同義である。Object は情報と機能の塊を表し、Action はオブジェクト間のインタラクションを示す。

Zoom-in/out 1 つの Action や Object を抽象化・詳細化することで、それらを異なるスケールで見ることがある。より抽象度の高い Action は、より詳細化されたいくつかの Action によって構成され、Object もまた同様である。このように、ある Action や Object のより詳細な部分に着目するとき、Catalysis ではこれを “Zoom-in” という言葉で説明し、逆を “Zoom-out” と呼んでいる。

たとえば、Catalysis では、最初からある Action をある Object の仕事として割り当てることはしなくてもよい。Catalysis で Action を記述するときには、まず「何をするか」を書く。そのあとで、「どのようにするか」を考える。これも Zoom-in の一種である。

FigB.1 に、Zoom-in の例を示す。

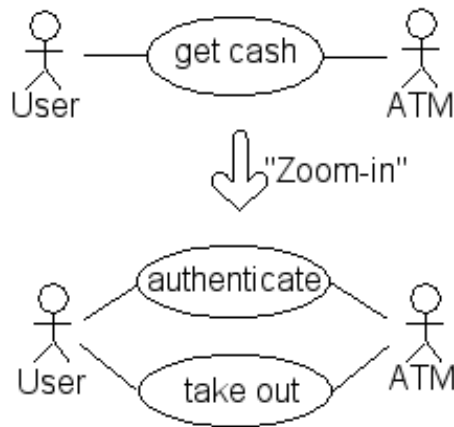


Fig B.1: “authenticate” and “take out” are zoomed-in actions of “get cash”

FigB.1 では、最初に “get cash” という Action を考え、「何をするか」を最初に定義している。その次に、これを Zoom-in してこの “get cash” をどのように実現するかを考え、2つの小さな Action を定義している。もちろん、この2つの小さな Action のそれぞれを、さらに小さな Action で構成することもできる。

このように、異なるスケールで Action や Object を設計することで、本質的な問題と細かい問題とを切り分けることができる。

B.2 Three Modeling Concepts

Catalysis は、Collaboration、Type、Refinement の3つの概念とともに、Framework と呼ばれるパターン化を核としている (FigB.2)。以下では、これら4点について説明する。

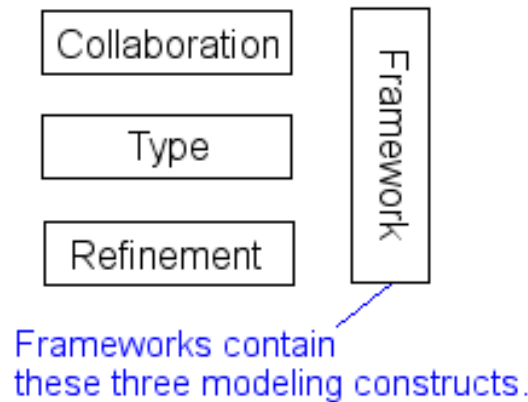


Fig B.2: Three Modeling Concepts And Framework

Collaboration Collaboration は、action 及びその action に関わる object の type の集まりである。Collaboration は refinement 可能であり、scoping や refinement の単位としても使われる。

Type Object の持つ、外部から見える振る舞いを記述するものである。一般的に Class は object の実装を表現するものだが、Type は実装を持たない。ゆえに、ある Type の仕様に対して、複数の異なる実装を持たせることができる。

Refinement 同じ物事について異なる詳細度レベルで記述した 2 つの Object や Action の関係を指す。B.1 で述べた Zoom-in, Zoom-out の概念がこれに当たる。Refinement には以下に示す例のようにいくつかのパターンが考えられる。

- Component design - Component specification
- Class implementaion - Type specification
- Particular sequence - Abstract action

一般的には、Catalysis では抽象レベルの高い要求を実装（コード）に変形していくが、必ずしもトップダウンの順になっていなくてもよい。

Framework 以上で説明した Collaboration、Type、Refinement によるモデルや設計のパターンは、別の場所で再び用いられることがあるものも多い。たとえば、「図書館から本を借りる」と「市役所に転入届を出す」は、高い抽象レベルにおいて構造的に似ている (FigB.3)。このように、たびたび現れる構造をパターンとし

て再利用するとき、このパターンを Framework と呼ぶ。Framework は、パッケージとして表現される。

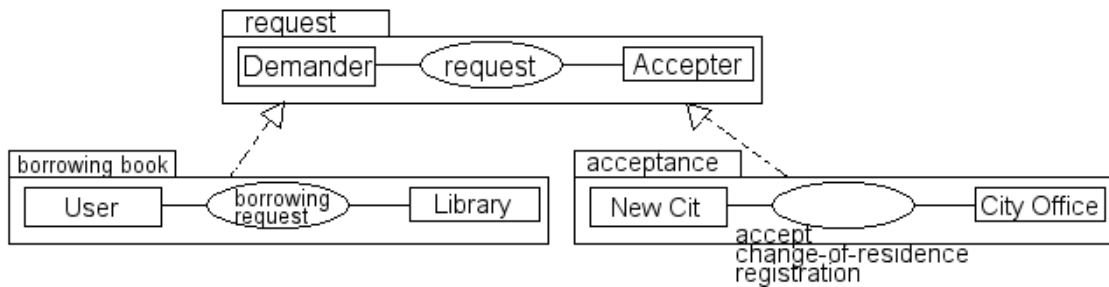


Fig B.3: Framework Example

B.3 Refinement の例

ここでは、ある要求を Sequence 図にしていくまでの Refinement の過程を、具体的な例を挙げて説明する。ただし、通常 Catalysis では事後条件などの表現に OCL(Object Constraint Language) を用いるが、ここではわかりやすい表現のために自然言語を用いるものとする。また、ここで挙げた過程は一例であり、必ずしもこのような順番にならない。

例に用いる要求を次に示す。

ユーザは、Internet Service Provider(ISP) から新しいアカウントとそのパスワードを取得する。

Abstract Action 最初の要求から、次のような Use Case 図が描ける。

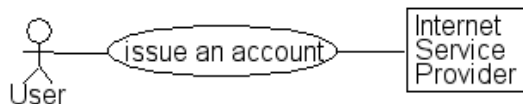


Fig B.4: Abstract Action

ここで、“issue an account” は一般的には Use Case と呼ばれるが、Catalysis ではこれを Abstract Action と呼ぶこともある。

UseCase の事後条件 次に、FigB.4 の事後条件を考えると以下のようなになる。

1. ユーザは新しいアカウントとパスワードを入手する。
2. ISP は新しいアカウントとパスワードを保存する。

Collaboration の Refinement FigB.4 はユーザ、ISP、及び “issue an account” の Collaboration であるが、これは FigB.5 のように refine される。

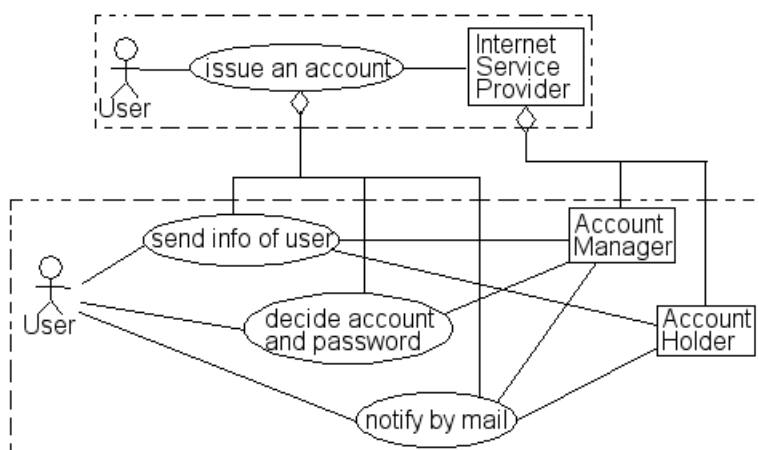


Fig B.5: Refinement Of Collaboration

B.6 で説明するように、UML の Class 図で使うような Aggregation の記号によって、Action や Type の Refinement 関係を記述することがある。FigB.5 では、“issue an account” という Action は “send info of user”、“decide account and password”、“notify by mail” という 3 つの Action に refine されている。これはまた、“issue an account” がこれら 3 つの Action によって構成されていることも示す。同様に、ISP は Account Manager と Account Holder という 2 つの Type に refine される。また、FigB.5 全体を指して、上部の破線で示した Collaboration が、下部の破線で示した Collaboration に refine されたこともこの図からいえる。

先に示した事後条件は次のように分解される。

1. ユーザの情報が ISP に送られる。
2. ISP は新しいアカウントとパスワードを登録する。
3. 登録されたアカウントとパスワードを通知するメールが User に届く。

Refinement:Role 割り当て 説明が冗長となるため、以下では FigB.5 に示された Action “decide account and password” について考えていく。

ここで、“decide account and password” で、誰がアカウントとパスワードを決めるのか考えてみる。一般に、アカウントもパスワードもユーザが決めることができるサービスもあれば、ともに ISP が決めてユーザに通知する場合もある。また、アカウントは ISP が決めるがユーザがパスワードを決めるという場合もある。したがって、誰がアカウントとパスワードを決めるか、という問題を後回しにして、FigB.5 を refine することを考える。

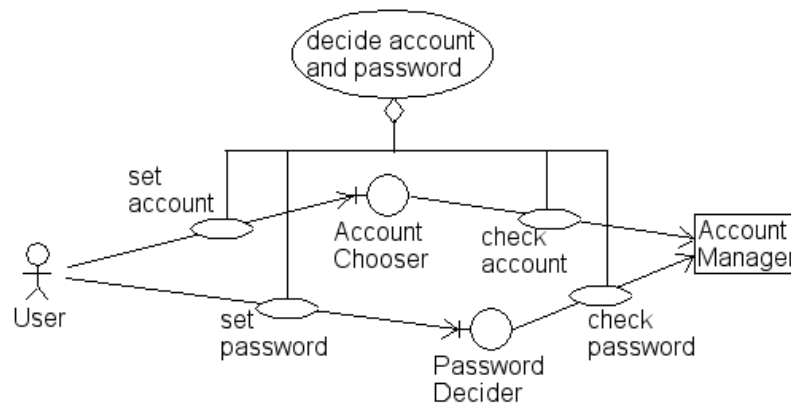


Fig B.6: Assignment Of Responsibility

Account Chooser や Password Decider は Role と呼ばれる。これは具体的な Type ではなく、行うべき仕事だけを割り当てられた仮の存在である。ここでは、Account Chooser は「適切なアカウントを選び出す」という仕事を与えられ、Password Decider は「パスワードを決定する」という仕事を与えられている。

このように、Catalysis では、重要なことを先に決め、詳細は後回しにする手法を度々用いる。これは、B.5 で述べる Abstraction の原則に基づくものである。

Type 割り当て 次に、FigB.6 の Role を具体的な Type に割り当てていく。

1つの割り当て方の例として、「アカウントもパスワードもユーザが決める」という方針になったとする (FigB.7)。

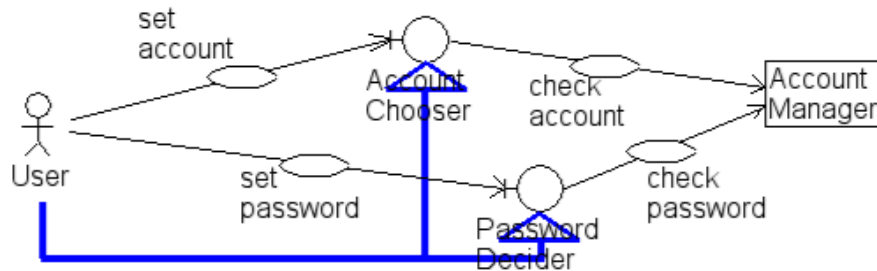


Fig B.7: Assignment Of Type : User decides account and password

これに従うと、FigB.6 は次のように refine できる。

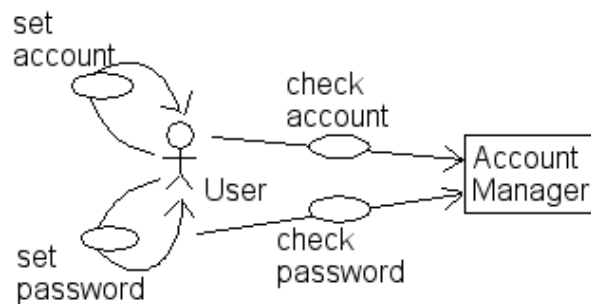


Fig B.8: Refinement Of FigB.6 : User decides account and password

他の場合を次に示す。アカウントもパスワードも ISP が指定する場合は FigB.9 および FigB.10 のように refine され、アカウントは ISP が決めるがパスワードはユーザが決める場合は FigB.11 および FigB.12 のように refine される。

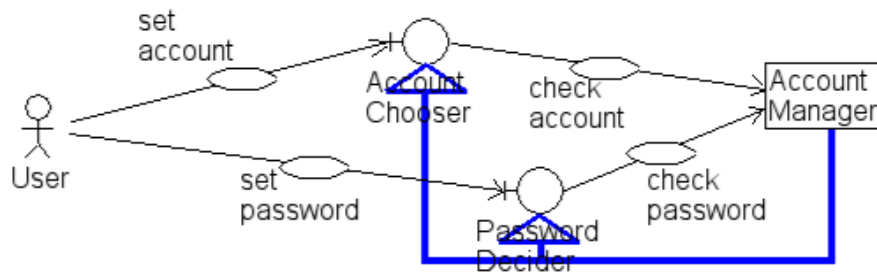


Fig B.9: Assignment Of Type : ISP decides account and password

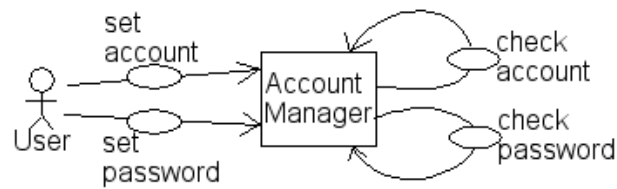


Fig B.10: Refinement Of FigB.6 : ISP decides account and password

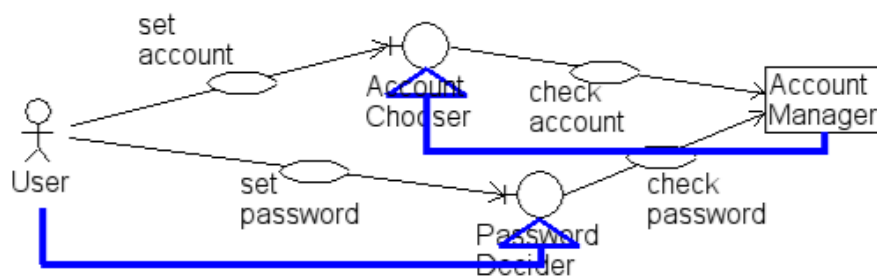


Fig B.11: Assignment Of Type : ISP decides account and User decides password

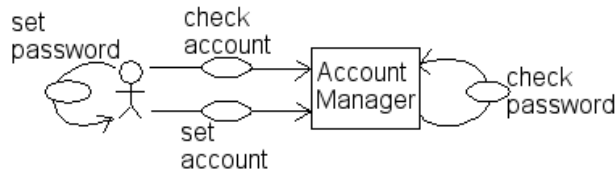


Fig B.12: Refinement Of FigB.6 : ISP decides account and User decides password

Sequence 図 FigB.8 から Sequence 図を描くと、 FigB.13 のようになる。

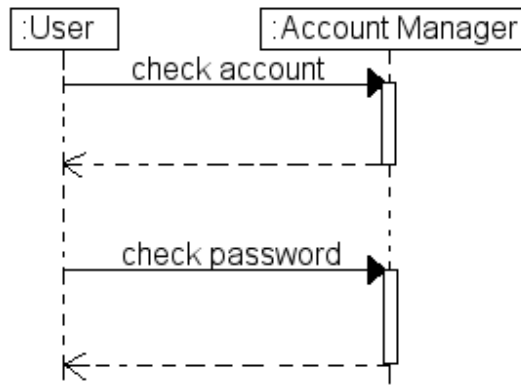


Fig B.13: Sequence Diagram As Refinement

B.4 Three Levels of Modeling

Catalysis では、以下の3つの段階に分けてモデリングを扱っている。

Domain/Business モデリングの対象となっている領域を表す。これをモデリングするには、対象となるビジネスのプロセスや、そのプロセスに関わる role や collaboration を理解する必要がある。

Component Specification Type の仕様の書き方を用いて、component の外部的振る舞いの仕様をモデリングする。内部の詳細についてはここではモデリングしない。

Internal Design Component の仕様に合うように Component の内部構造をモデリングする。この内部構造を描くときにも collaboration を用いる。このモデリングの結果、より小さな Component がモデリングされた内部構造に登場することもある。

B.5 Three Principles

Catalysis は次の 3 つの原則に基づいている。

Abstraction 詳細を隠蔽して行うプロセスを指す。これによって、次のような利益が生まれる。

- 詳細にとらわれなくて、要求と構造の決定を扱える。
- ビジネスルールやビジネスプロセスをコードに落とししていく階層化モデルを扱える。
- 方法論的な Refinement とコンポーネント構造を扱える。

Precision ドキュメントにおける厳密さを指す。

Pluggable Parts Class に限らず、Framework や仕様やパターンを再利用することを指す。

B.6 Catalysis での記法

Catalysis を論じている [Souz 98] では、UML1.X や UML2.0 で用いられていない記法を独自に扱っているが、その記法に特別な厳密性はないように考えられる。たとえば、UML の Aggregation の記号を、FigB.14 で示すような形で描くこともあれば、一方で FigB.5 のように通常の UML の記法が用いられることもある。Catalysis で用いられている記法についての詳しい解説は少ないため、詳述は難しいが、ここでは主に本論文中で用いた記法について説明する。

Collaboration Diagram FigB.1 に示した図は、[Souz 98] では “Collaboration Diagram” とコメントされているが、詳細な記述はない。形式としては、UML の UseCase 図と同様の形を取っている。

Action の refinement 関係の表現 B.1 で述べたように、Action は、その Action を実現するためのより小さな Action に refine されることがある。その関係を表現するための記法は、UML2.0 でもまだ提供されていないが、Catalysis では FigB.14 のように、UML の Class 図で使われるような Aggregation を用いて表現できる。なお、この図は FigB.1 で挙げた 3 つの Action の関係を例にとって示した。

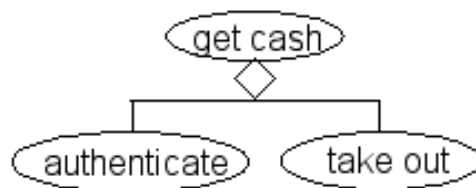


Fig B.14: Diagram depicted constituents

Directed Action UML1.X の Collaboration 図 (UML2.0 では Communication 図) では、Object 間のメッセージの流れを単純な矢印で表記した (FigB.15)。



Fig B.15: Collaboration Diagram Of UML1.X

Catalysis ではオブジェクト間で行われるアクションを “Directed Action” と呼び、FigB.15 と同じことを FigB.16 のように表現することがある。

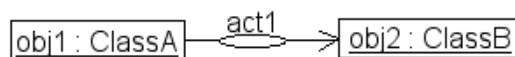


Fig B.16: Directed Action