JAIST Repository
https://dspace.jaist.ac.jp/

Title	オブジェクト指向方法論のための通信モデルに関する研究
Author(s)	赤木, 匡博
Citation	
Issue Date	2001-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1445
Rights	
Description	Supervisor:片山 卓也, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

オブジェクト指向方法論のための通信モデルに関する研究

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授

審査委員　渡部卓雄 助教授

審査委員　落水浩一郎 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

810002　赤木匡博

2001 年 2 月 15 日

Copyright © 2001 by Masahiro Akagi

要 旨

本論文ではオブジェクト指向方法論における通信の扱いに関して整理し、概念形成をする。また、形式的な動的モデル ObTS における通信モデルを定義し、通信モデルの独立的な考察を試みる。それを基に、CASE ツールにおいて通信モデルを柔軟に扱う方法を提案し、形式的仕様記述言語 ObCL とそのシミュレータ ObML に対し拡張を施す。特に本研究の成果物は、異なる通信モデルを含むシステムを記述することができ、表現能力の高いものになっている。さらに、イベントと通信の一元的な定義に基づいた開発プロセスを提案し、ObTS/ObCL を用いた設計の支援と、開発工程において通信モデルを考慮するための指針を示す。最後に仮想的なエレベータシステムの設計を適用例として掲載する。

目 次

1　はじめに　1

2　背景　4

2.1　オブジェクト指向開発 4
2.2　CASE ツールとシミュレーション 5
2.3　ObTS/ObCL/ObML 6
2.4　本研究の目的とアプローチ 7

3　通信モデル　8

3.1　オブジェクト指向における通信 8
3.1.1　イベントの定義 8
3.1.2　通信の定義 10
3.1.3　オブジェクトのリンク 10
3.2　通信モデルの概念 10
3.2.1　実行モデルと通信モデル 10
3.2.2　通信モデルのパラメータ化 11
3.3　イベント空間 12
3.3.1　発生からの存在時間と遅延時間、寿命 13
3.3.2　イベントバッファと待機領域 13

4　CASE ツールと通信モデル　14

4.1　ObCL/ObML 14
4.1.1　ObCL の通信モデル 14
4.1.2　フィールド 14
4.1.3　イベントクラス 15
4.1.4　ObCL の言語仕様 15

4.2 通信モデルとフィールド . . . 17
4.2.1 フィールドの解釈 . . . 17
4.2.2 フィールドプロパティ . . . 18
4.3 実装 . . . 19
4.3.1 ObCL 言語仕様の変更 . . . 19
4.3.2 動作例 . . . 21

5 通信を意識した開発方法論 25
5.1 オブジェクトモデルと ObTS . . . 25
5.2 開発プロセス . . . 26
5.3 イベントフローリスト . . . 28
5.4 オブジェクト関連図 . . . 28
5.5 フィールドの決定 . . . 30
5.5.1 フィールドリンク図 . . . 30
5.6 ObTS の作図 . . . 31
5.7 ObCL の記述 . . . 32

6 適用例 34
6.1 仮想的なエレベータのモデル化 . . . 34
6.2 設計分析 . . . 36
6.3 ObTS/ObCL 記述 (第 1 段階) . . . 41
6.3.1 並行オブジェクト . . . 41
6.3.2 イベントクラスとフィールドクラス (ObCL 記述) . . . 42
6.3.3 オブジェクトの設計 . . . 42
6.4 詳細化 (第 2 段階) . . . 52
6.4.1 フィールドプロパティ . . . 52
6.4.2 内部オブジェクト . . . 52

7 評価と考察 55

8 今後の課題 56

9 まとめ 57

図 目 次

2.1 概観 6

3.1 振る舞いの連携 8
3.2 イベント空間 12

4.1 イベントがフィールドを伝わる様子 15
4.2 異なる通信モデルを持ったフィールド 17
4.3 実行モデルと遅延時間・寿命 18
4.4 イベント寿命による動作例 21
4.5 イベント遅延時間による動作例 23

5.1 開発の流れ 27
5.2 オブジェクト関連図 29
5.3 フィールドリンク図 31

6.1 エレベータの外観 35
6.2 オブジェクト関連図 39
6.3 エレベータシステムのフィールドリンク 40
6.4 全体の ObTS 図 (細部省略) 41
6.5 ドアの開閉 43
6.6 リフトの動き 44
6.7 エレベータの階 46
6.8 コントローラー 48
6.9 ドアの制御部のクラス 53
6.10 すべての「呼」を処理する仕様 54

表 目 次

4.1 ObCL 記述の例 16
4.2 追加予約語 19
4.3 イベント寿命の記述例 22
4.4 イベント遅延時間の記述例 24

6.1 対話実行の様子 37
6.2 イベントクラスとフィールドクラス 42
6.3 ドアクラス記述 43
6.4 リフトクラス記述 45
6.5 かご室クラス記述 47
6.6 コントローラクラス記述 49
6.7 システム記述 51
6.8 呼をためるフィールド 52

第 1 章

はじめに

近年、ソフトウェア開発の大規模・複雑化はめざましい。その解決にもっとも有用とされているのがオブジェクト指向開発である。オブジェクト指向設計では多視点なモデル化を行うことによって対象を記述する。その一つに動的モデルがある。動的モデルではシステムの振る舞いについて扱う。その多くでは Statechart [5] が記述法として採用されている。

また、飛躍的に需要が延びている分野に組み込みシステムがある。組み込み機器の高機能化に伴い、組み込みシステムもまた大規模・複雑化の一途をたどっている。組み込みシステムのようなリアクティブシステムの開発では、古くから Statechart が設計や仕様記述として用いられてきた。そして、この分野にもオブジェクト指向開発は徐々に適用され始めている。

組み込みシステムなどの分野では年々、製品寿命が短期化するのに伴い開発期間が短縮されてきている。ライフサイクルの短い製品は、基本機能はそのままで付加機能を徐々に変更しながら繰り返し開発を行うものが多い。そのような開発にはオブジェクト指向がもたらす再利用性が有効である。

また LSI の集積技術はとどまるところを知らず、ソフトウェアまでも同じチップ上に載せるようになってきている。それはハードウェアができる前からソフトウェアを開発する必要があることを意味している。さらに掘り下げると、ハードウェアなどの環境が決定する前からソフトウェアを設計していく必要があることを示唆している。このような開発にはシミュレーションによるテストが有効である。

シミュレーションは誤りの早期発見にも効果を発揮する。プログラミング時のエラーだけでなく、設計の段階でも誤りは混入する。そのような誤りは設計工程で食い止めた方がよい。上流工程で発見できる誤りをできるだけ早い段階で取り除くことにより、開発コス

トが減少する。

製品開発において仕様書は重要である。開発の各工程で仕様書を作ることが望ましい。仕様は次の工程の前提であり、それをもとに開発が進められる。当然その仕様に誤りや矛盾が混入していれば、製品は当初の要望と異なったものができあがったり、正しく動かないことがありうる。仕様書の内容があいまいであったり記述が足りないために、下流工程であいまいな解釈を行い開発を続けることも同様である。これらを未然に防ぐために形式的アプローチが有効である。設計仕様を形式的に記述することは、誤りを早い段階で発見することにつながる。形式的仕様記述はその実行支援環境を用意できる。これは仕様書の段階でシミュレートできるということである。シミュレーションによって誤りや矛盾を発見し取り除くことができる。

さて、伊藤によって提案された ObTS [6] はオブジェクト指向方法論における動的モデルのための形式的仕様記述モデルである。ObTS の計算モデルは Statechart に基づく。

Statechart は状態遷移図に階層構造を与えることによって状態数爆発を抑え、また並行性を導入するなど表現力の高い記述法となっている。しかしオブジェクト指向開発への適用では、並行動作や階層の単位の意味が明確でないことや、大域変数を用いるために大規模な複雑さに対応できないなど、オブジェクト指向との親和性の低さが目立つ。そのため、動的モデルでの利用ではいくつかの主要なオブジェクトの単独の振る舞いを記述するにとどまる。全体の振る舞いを把握するためには、いくつもの補助的な記述を用意しそれらと注意深く照らし合わせることによって理解する必要がある。

そこで ObTS ではオブジェクト指向との親和性を高めるためのさまざまなアプローチが採られている (2.3 節)。さらに ObTS のための形式的仕様記述言語 ObCL でシステムを記述することによって、全体の振る舞いをシミュレーションによってテストできる。

しかし、オブジェクト間通信の概念に Statechart のイベント通信の計算モデルを適用しているため、実対象との対応が見えにくくなっている。その通信モデルは同期的であり、通信遅延がないなど理想的なセマンティクスが与えられている。これは非同期的なシステムの記述を困難にさせている。記述したとしてもそれを理解する実装者の負担は大きい。

また本研究を通じて、オブジェクト指向方法論におけるイベント (メッセージ) 通信の概念も明確でないことが浮彫りにされた。そこで本研究では、オブジェクト指向方法論の通信の概念を洗いだし、整理することを試みている。通信モデルに関わるいくつかの定義や提案をし、実際に ObCL に拡張を施した。またそれらを考慮した開発の指針も与える。

本論文の構成は次の通りである。第 2 章では本研究の背景技術とその問題点を述べ、そ

れらに対する本研究の目的とアプローチを示す。第 3 章ではオブジェクト指向でのモデル化、特に動的モデルにおいて扱われている通信の概念を明確にする。これは本研究の問題領域も特定している。その上で通信モデルの定義を行う。第 4 章では実際に ObCL/ObML に 3章の成果を適用した。これにより、複数の通信モデルを持つような広範囲のシステムの自然な記述を柔軟にできるようにする。第 5 章では ObCL/ObML を用いて、通信モデルを意識した開発を行うための指針を示した。第 6 章では実際に仮想的なエレベータを題材に適用例を示した。第 7 章で適用例をもとに本研究で提供した ObCL の拡張と、提案した方法論による開発アプローチの有用性を評価する。そして第 8 章で、本研究で浮彫りにされた通信モデルのさらなる概念形成のための指針と、今回追求できなかった残された課題について述べる。

第 2 章

背景

本研究の背景となる諸技術の紹介と問題点、そして各分野に対する本研究の目的とアプローチを述べる。

2.1　オブジェクト指向開発

オブジェクト指向では、対象をいくつかのオブジェクトとその相互作用によってモデル化する。その相互作用の媒介としてモデル化されるものがイベント (メッセージ) であり、イベントは極めて重要な位置づけにされている。

オブジェクト指向方法論において、対象システムの振る舞いのモデル化を行うものに動的モデルがある。一般にイベントは動的モデルにおいて状態と共にモデル化される。またその他のモデルでも、イベントやイベントの関わる概念は存在している。既存の方法論ではイベントの概念についての解説は行うが、独立的にモデル化されることはなく、イベントそのものの扱いや仕組みについて明確な定義を行わない。つまりイベントの一元的な定義を行うことがないまま開発が進められる。

また、オブジェクトのクラスの概念を用いてイベントクラスを作成するというアプローチもしばしば採られる。これは、ObCL のイベントクラス (4.1.3 節) の概念と違い、意味の不明瞭さを招く。なぜなら、そのイベントクラスとその他のクラスの関連は何によって示されるのか明確でない。ただの「意味の関係」だけを示したいのかも知れないが、動的モデルとの対応が取りにくく、また実装工程では異なる構造になることが明白である。このようなアプローチがとられるのはイベントに対する解釈の指針が未熟なためだと考える。イベントや通信モデルを別の枠組みとして捉えるという指向がない前提の上では、それらをオブジェクトとしてモデル化するのはオブジェクト指向としてはごく自然なアプ

ローチであると理解できる。

つけ加えると、オブジェクト指向言語には他のオブジェクトの操作のためにメソッドが提供されている。オブジェクト同士がメソッドを呼び出しあうことによって連携動作を実現している。これはオブジェクト間通信の実装である。ただ、メソッドは返り値があることや制御を奪うことを常に念頭に置いておかなければならない。オブジェクト指向開発では、実装段階がオブジェクト指向言語によるコーディングであることを仮定していることが多々ある。するとイベントとメソッドの対応が分かりにくくなることがある。

なお本論文では、オブジェクトという言葉はインスタンスオブジェクトの意味で用いる。

2.2 CASE ツールとシミュレーション

CASE ツールに対する絶対的な定義はないが、ここではコンピュータ上の設計支援ツールとする。

オブジェクト指向分析/設計のための CASE ツールは古くから製品化されている。以前はグラフィックエディタの機能程度しか持たない物ばかりだったが、最近では整合性チェックや、簡単なソースコードを出力するようなものが主流になっており、多く利用されるようになった。

本論文では、形式的仕様記述言語のための実行環境も CASE ツールの 1 つと見なして述べている。形式的 CASE は従来の CASE とは明らかに異なるアプローチであるが、コンピュータの上で実行することによって強力にユーザを支援することが可能である。

オブジェクト指向自体がシミュレーションのために発明されている。オブジェクト指向方法論の動的モデルは、対象の振る舞いのモデル化であるためシミュレートが可能であり、またシミュレートした方が理解を助ける。

シミュレーションを行うためには動的モデル、つまり振る舞いの記述について形式的に定義する必要がある。実際 Statechart などは様々な形式化が試みられている [3]。

動的モデルを形式的に支援可能な CASE ツールでは、ただ 1 つの計算モデルを定義し、それに基づき動作する。振る舞い仕様記述においてはその動作モデルが設計対象の実際の仕組みと異なっていると、仕様の意味は正しくても実装段階で動作原理を解釈しなおす必要が生じる。当然その解釈は形式的なものではなく、せっかくの形式的仕様の必要意義が失われることになる。

また、せっかくのシミュレーションならばテストのためだけでなく、設計者 (仕様記述の読者) の直感的理解を助けるものであったほうが CASE ツールとしての意義がある。そのためには実対象と対応のとりやすい動作をするほうが良い。

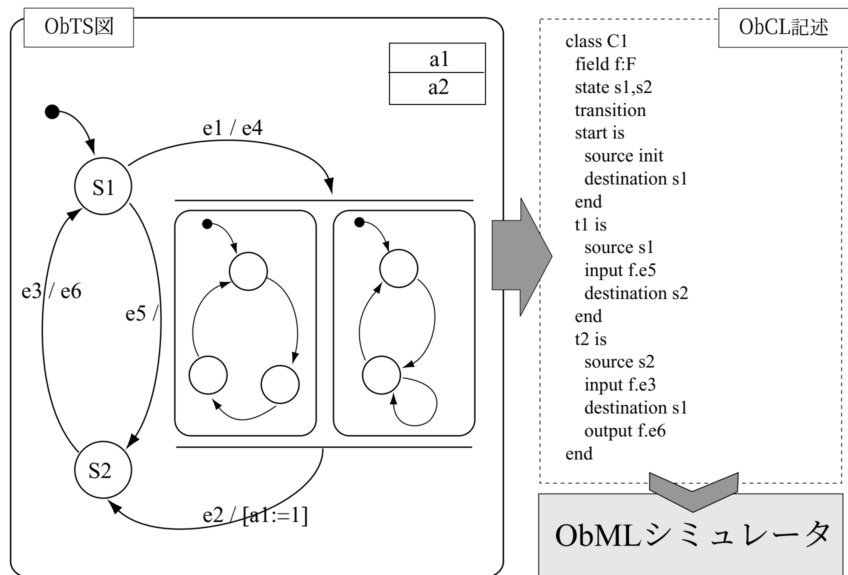

図 2.1: 概観

2.3 ObTS/ObCL/ObML

ObTS は動的モデルのための仕様記述モデルである。ObTS では階層構造をオブジェクトによって表現し、カプセル化の概念を導入してる。そのため変数 (属性) は局所的であり、複雑さを解消する。オブジェクトを扱うため、オブジェクトモデルとの親和性が高い。オブジェクト指向方法論において、他モデルとの親和性が高いことは生産性の向上が期待できる。

また ObTS の計算モデルは Statechart、特に CASE ツール Statemate で採用されているものを基に定義されている。オブジェクトの振る舞いは状態遷移図で示され、オブジェクト間の相互作用はイベント通信によってもたらされる。

ObTS をベースにした形式的仕様記述言語 ObCL が提案されている [4]。ObCL はその実行環境である ObML によってシミュレーションができ、仕様記述の動的解析が可能である。

ObCL/ObML はオブジェクトの振る舞いとシステムの振る舞いを統括的に扱う。対象システム全体の振る舞いをシミュレート出来るので、設計者や実装者の理解を助けることになる。しかし、オブジェクト間通信や全体の振る舞いを示すことによって、Statechar 式の計算モデルを持つことの弱点が浮彫りになる。現在の計算モデルはシュミレートすることが容易である反面、現実世界の並行動作における非同期的な通信との対応がわかりに

くい。実装段階で非同期通信の環境を選択したとき、実装者は同期的な仕様記述を見て、改めて頭の中で解釈を行いなおすことになる。これではあいまいさを除去したはずの形式的仕様記述の存在意義が薄れることになる。

また第 4 章でも ObCL/ObML について分析、解説する。

2.4 本研究の目的とアプローチ

本研究ではイベントと通信に関する概念を明確にし、一元的な定義を行う。これはオブジェクト指向開発の各工程、各モデルにわたって一意にイベント通信を扱うためである。また、動的モデルの通信モデルに関して議論を行いやすくする狙いもある。

さらに定義した通信の概念をもとに、通信モデルの概念形成を行う。通信モデルを独立的に扱うことによって、動的モデルに対して任意の通信モデルを適宜、与えることを可能にする。

本研究では実際に動的モデルのための形式的 CASE ツール ObCL/ObML に対して、通信モデルを変更するための機構を与える。これにより実対象に適した通信モデルを用いることができ、より自然なシミュレーションが可能となる。これは実装者の理解を助けるものであり、開発の品質向上にもつながる。

また、イベント通信に対して一元的な定義のもと設計を行い、あいまいさを削減した設計手法により、ObTS/ObCL の記述作業と親和性の高い方法論を提案し、通信モデルを意識したシステム開発の指針を示す。

第 3 章

通信モデル

この章ではオブジェクト指向方法論で扱われる「通信」について言及し、通信モデルの外観を整え、簡単な定義を行う。

3.1 オブジェクト指向における通信

通信を扱わないオブジェクト指向方法論はない。オブジェクト指向では、対象をもの(オブジェクト) の集合として捉え、個々のオブジェクトがそれぞれイベントのやり取りを行うことによって相互作用を表現し、その振る舞いをモデル化する。

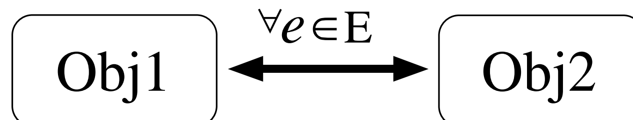

図 3.1: 振る舞いの連携

オブジェクト指向の概念レベルではオブジェクト間の相互作用はメッセージによってもたらされるという前提が多いが、本研究ではオブジェクトの相互作用の媒介を示すものは**イベント**という言葉で統一する。

3.1.1 イベントの定義

イベント (事象) という言葉はオブジェクト指向の概念において、特に一般的な概念を逸脱したような捉え方をされることはない。本研究でもそのような概念を覆すような要素

はない。むしろ、一般的なイベントの解釈を前提とした上で展開する。ただ、あらためて動的モデルにおけるイベントの厳密な定義を行いたい。

さて本研究においてイベントとは、オブジェクトとオブジェクトを繋ぐ唯一の概念である。また、イベントはオブジェクトではない。

基本的にイベントの持つ意味はあいまいである。抽象度もイベントによって異なる。どのようなイベント (の意味) をモデル化するかは設計者にすべて任される。しかしイベントの意味に関わらず、イベントの振る舞いは定義することができる。

また、イベントはメソッド呼び出しのような返り値を期待できるようなものは想定しない。つまり他のオブジェクトに対して何らかの値を要求し、その値を得るためには、要求と応答 (返り値) の少なくとも 2 つのイベントが必要となる。

イベントは発生する。また、発生しなければイベントは存在しない。イベントを発生させるのはオブジェクトのみである。この場合、外部イベントの発生も「アクター」オブジェクトとして捉える。つまり、あるイベント e について最初に確認できるようになったことを**発生**と呼び、確認できるようになった時点を**発生時点**とする。確認と言ったが、確認される前は真に存在していなかったという解釈でよい。発生したイベントがその後どうなるべきかはイベントの性質による。

イベントは取得される。イベントは遷移を発火させる。それはオブジェクトの動作のきっかけであり、唯一のものである。受理可能状態であれば取得され、不可状態であれば何も起らない。あるイベント e によってある遷移が発火したことを**取得**と呼び、その時点を**取得時点**とする。その後のイベントがどうなるかはやはりイベントの性質による。

また、イベントは取得によって存在しなくなりうるが、取得以外で存在しなくなることを**破棄**されたと呼ぶ。

イベントの取得と発生はオブジェクトが関知するが、破棄についてオブジェクトは関知しない。つまり、オブジェクトがあるイベントを “破棄させる” ということは起こり得ない。

概念レベルで現実をモデル化してみる。人がある事柄や現象を目撃しているとする。その事柄・現象もオブジェクトであり、そこで発生した “イベント” が人 (オブジェクト) によって取得されたことになる。人がそれを見ていなければ (状態)、そのイベントは発生したにも関わらず、オブジェクトによって取得されることはない。次の瞬間「見た」としても先ほどのイベントは存在しない。つまりそのイベントは瞬時に破棄されてしまっている。

また、本論文ではイベントに関して受理と取得という言葉を用いている。主に取得はオ

ブジェクトの立場での表現で、**受理**は遷移自体やイベントの立場での表現として用いている。しかし 2 つの相違は大きなものでなく、適宜読み替えても語弊は生じない。

3.1.2 通信の定義

オブジェクト間でのイベントのやり取りを指して、**通信**と呼ぶ。通信するのがオブジェクトであり、通信されるのがイベントである。イベントだけがあっても通信が行われているかは不明である。前節の言葉を用いると、発生と取得を確認できるとき通信が行われたことを示す。

またすべての通信はイベントが媒介であり、それ以外の通信の概念は持ち込まない。

3.1.3 オブジェクトのリンク

オブジェクト指向方法論では、オブジェクト同士の静的構造をリンクによって表現する。図的表現ではオブジェクトを表すいくつかのアイコンを結ぶ線である。意味の違いにより、線とオブジェクトの接点や、線の途中に何らかの図形をしるすことがある。

オブジェクト指向開発において、動的な側面と静的構造が何の関連もないことはあり得ない。リンクは主に関連、汎化、集約などがあるが、ここでは関連によって示されるオブジェクト間にはイベント通信による相互作用があるということだけ言及しておく。本研究ではイベントが媒介しないような関連は考えない。

詳しい議論は第 5 章でする。

3.2 通信モデルの概念

3.1.1 節で述べた、「イベントの性質」こそがまさに通信モデルだと言える。通信モデルはイベントがどう振る舞うのかを決定する。

3.2.1 実行モデルと通信モデル

動的モデルの形式化を行うとき、その計算モデルには実行モデルと通信モデルがあると考える。一般的にはどちらかに片寄ることや一緒くたに考えられることが多い。本研究においても厳密にその 2 つを分けた定義にまで追求されていない。しかし、通信モデルの議論と考察を行うに十分の概観を整える。

動的モデルにおいて重要な要素はオブジェクトとイベントの 2 つである。オブジェクトの振る舞いに関する計算モデルが**実行モデル**と考える。実は実行モデルでもイベントを扱う必要がある。しかし発生と取得のみに限定される。それ以外のイベントの扱いはすべて通信モデルによって定義される。

イベントのための計算モデルが**通信モデル**であると定義する。通信モデルはあるイベントがどのように振る舞うのかを決定するが、通信モデルだけではそのイベントが「通信」されることは保証できない。取得されないイベントも発生しうるからである。かといって実行モデルで通信を保証するという意味でもない。通信は保証されるものでなく、通信が起るかどうかという要素もシステムの振る舞いを示している。

オブジェクトだけが実行されても、複数のオブジェクトからなるシステムの振る舞いは表現できない。イベントだけがあっても、振る舞いは示されない。オブジェクトとイベントがあって初めて、システムの振る舞いはモデル化される。実行モデルと通信モデルのいずれも欠かせなく、また密接に連携している。

3.2.2 通信モデルのパラメータ化

後の考察で見えてくることだが、異なる通信モデルに対する考察は、通信モデルの内容を決めるいくつかのパラメータを抽出することであり、通信モデルの変更はその値を変動させることである。その方針を考慮した上で以後の議論を進める。

3.3 イベント空間

本研究では**イベント空間**という言葉と概念を提案する。イベント空間は現実対象との対応がとれないこともありうる。これは現実との 1 対 1 のモデル化であるオブジェクト指向に反する考えかもしれない。しかしイベントがオブジェクトでないと考えるならば、オブジェクトが存在する空間以外の枠組みでイベントを捉えることは理解を助ける。実際に本研究ではイベント空間は大前提であり重要な要素である。

イベント空間の考えは、イベントの振る舞いや性質を決定するのはイベント自身でなく、環境つまりイベント空間であると捉えている。

前節でイベントの発生と取得が通信モデルと実行モデルの接点であることを示唆した。図 3.2 で示す通り、イベントの発生と受理の時点を境界としたイベントが唯一存在する領域である。これを**イベント境界**と呼ぶ。このイベント空間を通信モデルが扱う領域であるとして考察する。

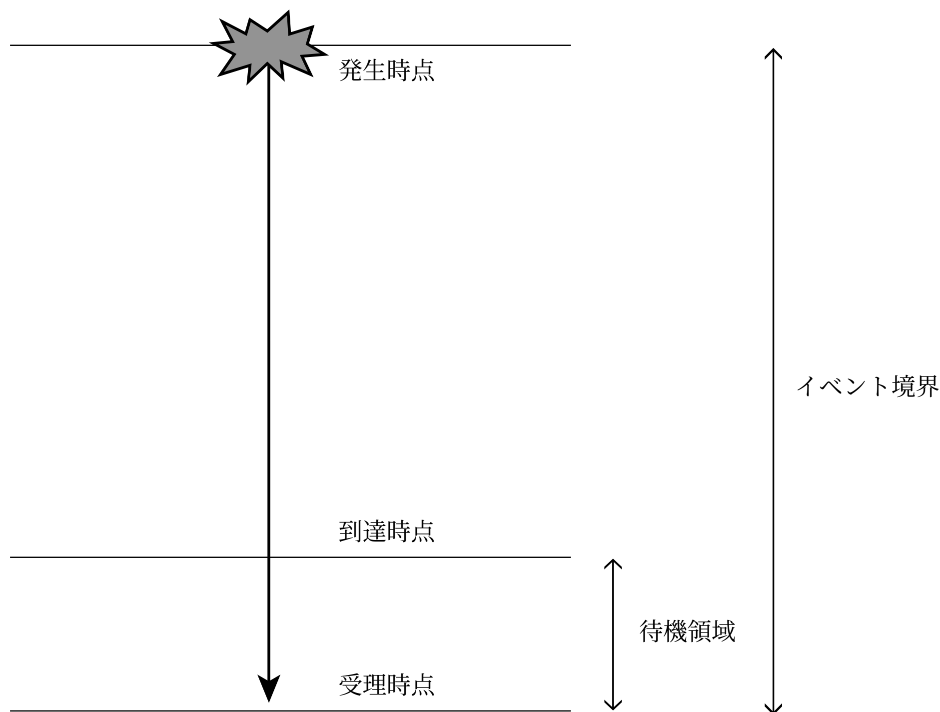

図 3.2: イベント空間

3.3.1　発生からの存在時間と遅延時間、寿命

図 3.2において、発生時点から受理時点への矢印には時間の概念が暗示されている。ここではイベント空間が持つ時間の概念の考察をする。本研究では実時間についての考察は行っていないので、本論文で扱う時間の概念はすべて仮想時間である。また、Statechart式の計算モデルにおけるステップは仮想時間である。

図に**到達時点**を示した。これはイベントがオブジェクトによって取得可能となった時点である。また取得可能となることを**到達**したと呼ぶ。これが意味するものはイベントの遅延もしくは通信の遅延の概念である。到達にかかる時間を**到達時間**、あるいはイベント空間のパラメータとして**遅延時間**と呼ぶ。つまり、遅延時間によって示された仮想時間が経過したのち、イベントはオブジェクトによって取得可能となる。それまでの間、オブジェクトは関知しない。

到達時点から受理時点の間にも時間がある。図では**待機領域**として示した。この間、イベントは取得可能である。この時間をイベントの**寿命**と呼ぶ。つまりイベントの発生からでなく、到達時点から受理時点までの時間がイベント寿命である。寿命によって示された時間が経過するとイベントは破棄される。

到達時間と寿命の和がイベントの**存在時間**となる。

3.3.2　イベントバッファと待機領域

オブジェクトがイベントバッファを持つ通信モデルが考えられる。

イベントバッファはイベント待機領域であると考える。つまり、イベントバッファに入ったイベントは取得されたと見なさない。取得されたように見えるかも知れないが、そのイベントはオブジェクトの状態をまだ遷移させていないことはわかる。これは取得されていないとする。

第 4 章

CASE ツールと通信モデル

ここでは第 3 章の成果を ObCL/ObML に適用する。

4.1 ObCL/ObML

本研究の視点から、形式的 CASE ツールとしての仕様記述言語とシミュレータである ObCL/ObML について分析、解説する。

ObCL では ObTS からさらに推し進め、再利用のためのクラスや通信範囲を限定するためのフィールドなどが導入されている。

4.1.1 ObCL の通信モデル

ObTS の計算モデルが Statechart に基づくことは 2.3 節で述べた。実際に ObML によって提供される通信モデルは次のようになる。

まず計算モデルとして、全体の動作をステップという仮想時間よって区切るという前提がある。また遷移にかかる時間は考慮せず、瞬間である。その上で、発生したイベントは 1 ステップ間有効であり、1 ステップ後には使用の是非に関わらず破棄される。また遅延時間というものは考えず、イベントは瞬時に到達するものと仮定される。この他は実行モデルに含める。

4.1.2 フィールド

ObTS では Statechart の通信モデルに基づきブロードキャスト通信を行う。ObTS 図では表されないが、ObCL には通信範囲の限定のために**フィールド**が導入されている [4]。

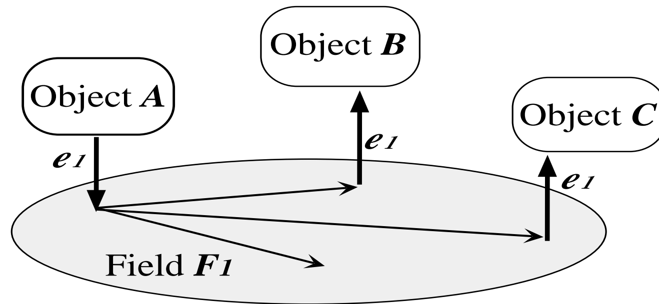

図 4.1: イベントがフィールドを伝わる様子

これにより、関連の深いオブジェクト同士の通信としてイベントが意味付けられており、通信の複雑さに対応している。なお本論文で以後でてくるフィールドという言葉はすべてここで述べたものを示す。

ObCL ではイベントは必ずフィールドに属することになる。またオブジェクトも 1 つ以上のフィールドが定義され、フィールドを介してイベントの通信を行う。

4.1.3　イベントクラス

イベントはイベントクラスのインスタンスとして表される。イベントクラスはイベント独自のクラスであり、オブジェクトのクラスとは異なる枠組みで捉えられる。

一般的なオブジェクト指向開発におけるアプローチとして、イベントをオブジェクトとしてとらえ設計することがあることを 2.1 節で指摘した。ObCL でもイベントに操作を持たせるなど、オブジェクト指向で捉えられている。しかし、オブジェクトとは別の枠組みで捉えるため不明瞭さがないことがわかる。これはオブジェクトを状態遷移図で表し、イベントはそれらの遷移を発火させるものであるという、明確な意味付けがされているためである。そのため自然なアプローチとしてこのような言語仕様に落ち着く。

これは本研究の題材として ObCL がもっとも親和性の高いものであることを示している。また、本研究の動機の発端でもある。

4.1.4　ObCL の言語仕様

ObCL 記述は主に、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、そしてシステム記述部からなる。オブジェクトクラスは遷移の一つひとつを 1 つのまとまりとして記述し、ObTS 図との対応が取りやすい。図的表現を思い浮かべやすいこと

は仕様記述者の負担を減らす。ObTS で表されたオブジェクトはオブジェクトクラスのインスタンスとして記述される。図 4.1 に記述例を示す。

この例に現れるイベントはすべてイベントクラス ATTRIBUTED のインスタンスだが、属性をもたないイベントクラス GENERIC_EVENT が用意されている。あらたにイベントクラスを宣言しないときは、通常すべてのイベントは GENERIC_EVENT のインスタンスとして宣言する。新たなイベントクラスは GENERIC_EVENT を継承しなければならない。また “--” 以降はコメント。

サンプルコード

```
event ATTRIBUTED     -- イベントクラス
  inherit GENERIC_EVENT  -- イベントクラスの継承
  attribute val:Int      -- イベント属性の宣言
end                  --

field F1             -- フィールドクラス
  event e1,e2:ATTRIBUTED  -- 属するイベントの宣言
end                  --

class C1             -- オブジェクトクラス
  field f1:F1          -- 属するフィールドインスタンスの宣言
  attribute n:Int      -- オブジェクト属性の宣言
  state s1,s2          -- 状態

  transition           -- 遷移記述開始
  start is             -- 初期遷移
    source init
    do n := 2            -- (属性の初期化)
    destination s1
  end                  --

  t1 is                -- 遷移その 1
    source s1            -- 遷移元
    input f1.e1          -- 入力イベント
    when f1.e1.val = n   -- 遷移条件
    destination s2       -- 遷移先
  end                  --

  t2 is                -- 遷移その 2
    source s2
    input f1.e1
    do f1.e2.val := 1    -- 遷移動作
    destination s1
    output f1.e2         -- 出力イベント
  end                  --
end                  -- オブジェクトクラスここまで

system SAMPLE        -- システム記述
  object c1:C1         -- オブジェクトインスタンスの宣言
end                  --
```

表 4.1: ObCL 記述の例

4.2　通信モデルとフィールド

個々のフィールドに対してフィールドプロパティを与えたモデルの提案と、その仕組みを提供する。

4.2.1　フィールドの解釈

本研究ではフィールドをイベント空間として解釈する。1 つのフィールドが 1 つのイベント空間であると考える。すると、システム記述にいくつものフィールドがあるということは複数のイベント空間を持つということを表している。1 つのイベント空間には 1 つの通信モデルが考えられる。つまりこの考えにより、異なる通信モデルを複数持ったシステムの記述が柔軟にできる。

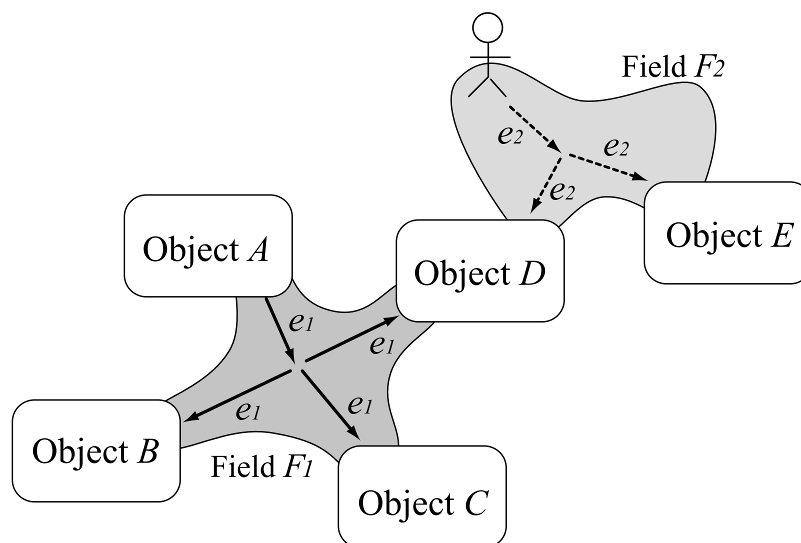

図 4.2: 異なる通信モデルを持ったフィールド

図 4.1 と図 4.2 の違いは、異なる通信モデルを持つことだけではない。図 4.2 はフィールドがイベント空間であり、イベントはフィールドの中にのみ存在している。図 4.1 にあるフィールドからオブジェクトへの矢印のような、所属のわからない空間がないことを示す。また、外部イベントはユースケース図のアクターのアイコンで表した。実際、外部イベントを発生させる**外部オブジェクト**とユースケースのアクターの概念は「モデル化の対象システムの外部からの、対象システムへの作用」という点で 1 対 1 に対応するものである。

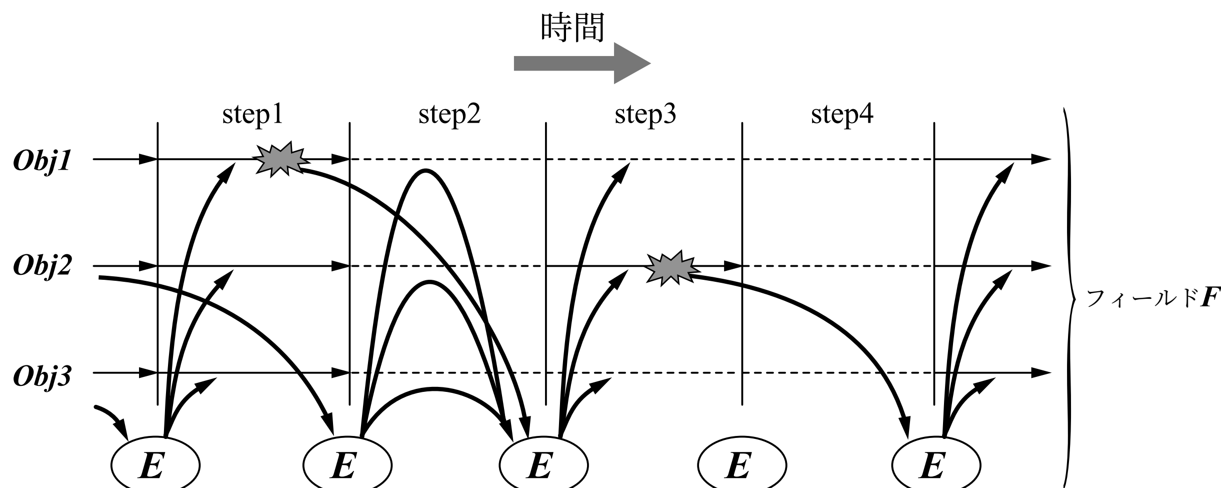

図 4.3: 実行モデルと遅延時間・寿命

4.2.2 フィールドプロパティ

本研究で実際に提供した、個々のフィールドに与える「通信モデルの一部」のパラメータを**フィールドプロパティ**と呼ぶ。

フィールドプロパティはいくつかのパラメータからなり、個々のフィールドに対して任意に与えられる。今回用意したパラメータは**イベント遅延時間**、**イベント寿命**、**DeadEvent**である。DeadEvent の説明は後で述べる。遅延時間はイベントがオブジェクトによって取得可能になるまでの時間であり、寿命はイベントが取得可能になってから破棄されるまでの時間を表す。破棄されるまではそのイベントはいずれかのオブジェクトに取得されるまで受理可能である。

これらはある種のバッファの存在を意味するが、このバッファは無限バッファを仮定している。無限なのはバッファの長さである。つまりイベント寿命が 2 であっても、過去 2 ステップ以内に到達したイベントは無限個バッファリングされることになる。

図 4.3 は遅延時間が 1 ステップ、寿命が 2 ステップのプロパティを持つフィールド F について表している。また、イベントバッファは 1 つのフィールドに 1 つとしたモデルである。横軸は時間の経過を表し、フィールド F に属する並行動作する 3 つのオブジェクトがある。横矢印は各オブジェクトの状態の遷移を表し、点線はそのステップにおいて遷移しなかったことを示す。E はイベントの集合でこの図ではイベントバッファのようなものとして捉える。縦線はオブジェクトの実行の区切りである。

曲線矢印はイベントの流れを表している。E より出ていく矢印が取得可能なイベントで

あり、取得されないこともある。遷移個所から E に入る矢印は、その遷移によって発生したイベントである。ここでは 1 ステップ遅延していることが表されている。また、分かりやすく発生を図示した。step2 ではイベントが取得されなかったことが表されている。このイベントは次のステップでも取得可能である。仮にいずれかのオブジェクトに取得されたとすると、矢印は E に戻ることはなく、次のステップにそのイベントはない。

次のステップ step3 では step1 で発生したイベントと step2 で受理されなかったイベントが取得可能である。ここでは step1 で発生したイベントが受理されたことが表されている。なぜなら step1 で発生したイベントが取得されなかった場合は、もう 1 ステップの間、取得可能であるため矢印は E に戻ることになる。一方 step2 で受理されなかったイベントは、このステップで取得されなければ破棄され、矢印が戻るようには表されない。

最後に step4 に注目すると、E にイベントがないために各オブジェクト共、遷移が起きていない。従来の ObML では外部イベントが発生しない限り硬直状態が続くが、ここでは step3 で発生したイベントが 1 ステップの後、取得可能となるためそれを受理した遷移が起る。

4.3 実装

本研究では ObCL/ObML に拡張を施すことにより、通信モデルを考慮した開発を支援しやすくした。実際には、前節で定義したフィールドプロパティの記述とそれを個々のフィールドに与える仕組みを提供した。

4.3.1 ObCL 言語仕様の変更

ここでは ObCL にあらたに用意した言語仕様とその機能を紹介する。

fieldproperty	import
delay	deadevent
life	Indef

表 4.2: 追加予約語

フィールドプロパティ　フィールドプロパティは ObCL におけるクラスの記述と類似するが、いわゆるクラスとは異なり、インスタンスが生成されることはない。ここに記述さ

れるものは通信モデルに関するいくつかのパラメータであり、任意の値を与えることにより異なる通信モデルを表現する。またそれ単体では意味を持たず、フィールドに対して適用することによってイベントの振る舞いを定義する。

fieldproperty *FP*
  delay [0-9]+
  life [1-9][0-9]* | Indef
  deadevent true | false
end

FP はフィールドプロパティの任意の名前。delay はイベント遅延時間で、続けて記述されるのは 0 以上の整数。省略時は 0 になる。life はイベント寿命で、続けて記述されるのは 1 以上の整数ならびに "Indef"。Indef は無限を表し、イベントが破棄されることはなく、イベントがなくなるのは受理された時だけである。省略時の値は 1。ここでイベント寿命の下限が 1 であることに注意する。寿命 0 はイベントの存在を許さない意味になる。実際、寿命はカウントされ 0 になったとき破棄されると考える。

dead-event deadevent[1]に続けて記述されるのは bool 値 (true あるいは false)。true であれば、イベント e が破棄された時 deadevent(target=String "e") が発生する。実は dead-event は第 3 章でのイベントの定義からは外れている。dead-event の存在意義は CASE ツールの便宜に終始する。実行モデルとしてイベントの入力のみが遷移を発火させるとすると、こういった時間付きの遷移を起こすにはそのためのイベントが必要となる。このようなイベントを暗黙のイベントと呼ぶことにする。概念的にはイベントが発生したと見なさないこともできる。

インポート 先に記述したフィールドプロパティを個々のフィールドに適用するために、インポートを用いる。

field *F*
  import *FP*
  ⋮
end

[1]厳密に決めてないが、DeadEvent はパラメータ名、deadevent は ObCL における宣言子、dead-event は実際のイベントを指す言葉として使っている

import に続く、FP は fieldproperty によって定義された任意のフィールドプロパティ名が入る。これにより、このフィールドクラス F の各インスタンスは FP に示されるフィールドプロパティを持つことになる。

4.3.2 動作例

以下に簡単な例を示し、フィールドプロパティの変更による振る舞いの変化を解説する。

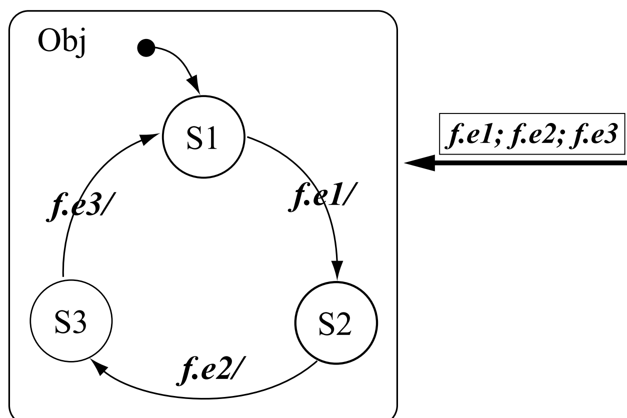

図 4.4: イベント寿命による動作例

図 4.4 はオブジェクト Obj が状態 S1 の時、同時にイベント f.e1、f.e2、f.e3 が発生した様子である。この時、イベント寿命が 2 ステップであると、f.e1 は当然受理され、次のステップでは f.e2 が残っており受理される。しかし、2 ステップたったイベントは破棄されるため、3 ステップ目に f.e3 はなく遷移は起らない。

表 4.3 は図 4.4 についての ObCL 記述である。フィード f のクラス F に、life 2 であるフィールドプロパティFP がインポートされている。イベント遅延時間は書かれていないのでデフォルトの “0” である。

```
fieldproperty FP                  -- フィールドプロパティ
  life 2                             -- イベント寿命
end                               --

field F                           -- フィールド
  import FP                          -- フィールドプロパティのインポート
  event e1,e2,e3:GENERIC_EVENT
end                               --

class C1
  field f:F
  state s1,s2,s3

  transition
  start is
    source init
    destination s1
  end

  t1 is
    source s1
    input f.e1
    destination s2
  end

  t2 is
    source s2
    input f.e2
    destination s3
  end

  t3 is
    source s3
    input f.e3
    destination s1
  end
end

system SAMPLE
  object obj:C1
end
```

表 4.3: イベント寿命の記述例

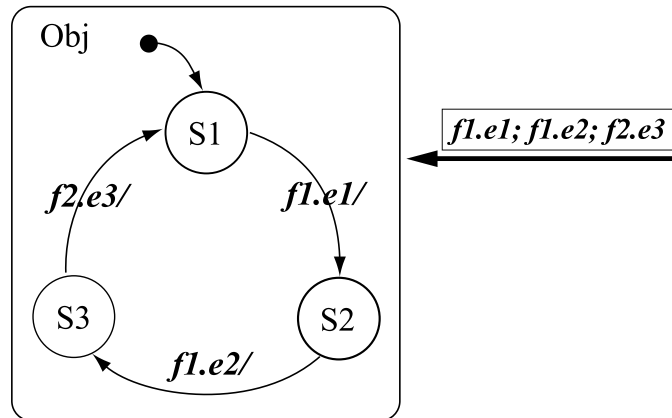

図 4.5: イベント遅延時間による動作例

次は似た例でイベント遅延時間について解説する。図 4.5 はオブジェクト Obj は 2 つのフィールド f1、f2 に属する。f1 のプロパティは先ほどの f と同様に、イベント寿命が 2 ステップであり、イベント遅延時間が 0 ステップである。フィールド f2 は寿命は 1 ステップであり、イベント遅延時間が “2” ステップとする。この時、状態 s1 においてイベント f1.e1、f1.e2、f2,e3 が発生した時、当然先程と同様 2 回の遷移が起きる。しかしこの時、イベント f2.e3 は遅延によってまだ到達していない。2 ステップ経た後、つまり 3 ステップ目で受理可能となる。よって 3 ステップ目においても遷移が起り、再び状態は s1 に戻る。ここで 3 つのイベントが同時に発生したことに注目したい。もしイベント f2.e3 が、2 つのイベントより 1 ステップ先に発生していたとすると、到達するのは f1.e2 が取得されるステップであり、次のステップでは寿命が 1 であるために破棄されており遷移は起らないことになる。

表 4.4 は図 4.5 についての ObCL 記述である。フィード f1 のクラス F1 に、life 2 であるフィールドプロパティFP1 がインポートされている。ここは先ほどのフィールド F と同様である。次にフィールド f2 のクラス F2 に対しては、delay が 1 であるフィールドプロパティFP2 をインポートしている。ここではイベント寿命を記述していなくて、値はデフォルトの “1” が設定される。

```
fieldproperty FP1               -- フィールドプロパティ
  life 2                           -- イベント寿命
end                             --

fieldproperty FP2               -- フィールドプロパティ
  delay 2                          -- イベント遅延時間
end                             --

field F1                        -- フィールド
  import FP1                       --フィールドプロパティのインポート
  event e1,e2:GENERIC_EVENT
end                             --

field F2                        -- フィールド
  import FP2                       -- インポート
  event e3:GENERIC_EVENT
end                             --

class C1
  field f1:F1
  field f2:F2
  state s1,s2,s3

  transition
  start is
    source init
    destination s1
  end

  t1 is
    source s1
    input f1.e1
    destination s2
  end

  t2 is
    source s2
    input f1.e2
    destination s3
  end

  t3 is
    source s3
    input f2.e3
    destination s1
  end
end

system SAMPLE
  object obj:C1
end
```

表 4.4: イベント遅延時間の記述例

第 5 章

通信を意識した開発方法論

ここで述べる方法論は動的モデル ObTS/ObCL を対象とし、その側面に特化したものになる。よってこれだけですべてに対応する方法論でもなく、他の方法論と相反するものでもない。むしろ相互補完することによって、よりよい効果が発揮できるものと期待できる。実際ここでは、主に設計工程初期の設計分析から設計仕様記述を完成させるまでを述べる。それ以前はユースケースなど既存の手法で十分であり、また最適だと考える。よってすでに要件定義などは済んでおり、またある程度分析ができているものとする。

他にも機能モデルなどには全く言及していないが、必要に応じて既存の適切な方法論を選択する必要がある。

また、現場の開発者は通信モデルのバリエーションを認識していることは少ない。自分の経験や想像できる一つの通信モデルを暗黙の内に仮定し、設計している。そのような場合、通信モデルを分析することすらままならない。そこで本研究ではある程度の指針を与える必要がある。

ここでは本研究の主旨であるイベントとその通信をモデル化するためのガイドラインを提供し、ObTS/ObCL を用いた開発への導入とフィールドプロパティ抽出のプロセスを示す。

5.1　オブジェクトモデルと ObTS

オブジェクトモデルは一般的にオブジェクトクラスを対象に議論する。一方、ObTS に現れるオブジェクトはすべてオブジェクトインスタンスであることは注意したい。つまり、クラス図よりもむしろインスタンス図などの方が ObTS とは関連が深いことになる。しかし、ObCL ではオブジェクトクラスについて記述を行い、インスタンス化や継承の概

念と機構を提供している。そのため、クラス図との親和性は大きい。

ただ、オブジェクトモデルはその柔軟性の代償として常にあいまいさが付きまとう。これはオブジェクトモデルと動的モデル間の移行性を損ねている。特に形式的に動的モデルの設計を行う際、両モデルの対比においてそのあいまいさと厳密性の落差が大きなコストとなる。

クラス図において、クラス間のの代表的なリンクには関連、集約、汎化があるがここでは関連に関してイベント通信を意識した設計を行うことにより、あいまいさの削減とObCL 記述の効率を上げることを試みる。この成果として 5.4節のオブジェクト関連図がある。

また、集約は内部オブジェクトと一致することもあり、汎化はオブジェクトクラスの継承と対応する。ObCL ではイベントクラスにも継承が用意されている。そのため、クラス図を用いてイベントクラスの継承をモデル化してもよい。しかし、そこからオブジェクト等への関連をリンクするべきでない。ましてや、イベント同士の関連をリンクすることは混乱を引き起こすため推奨できない。

5.2　開発プロセス

図 5.1に当開発手法における、流れの外観を示した。オブジェクトモデルとの分担、連携による設計はあっても良いが、ここでは ObTS/ObCL のための閉じた開発プロセスが採れるものになっている。

まず、オブジェクトの抽出が必要となる。これは既存方法論におけるオブジェクトモデルの指針を参考にするよりない。しかし本研究の立場として、いくつかの判断条件がある。1 つはオブジェクトは必ず状態を持つ。状態は 1 つであってもかまわないのでほとんどのものがオブジェクトとなりうる。それでも対象システムの視点から明確に状態が抽出できないのなら、対象にとって意味のないオブジェクトをあげている可能性がある。次はオブジェクトは必ず他オブジェクトと通信をする (第 3 章)。5.4 節で言及するが、オブジェクトは通信によってのみ関連づけられる。通信をしないオブジェクトは対象にとって意味がない。また、オブジェクトはオブジェクトクラスであってもよい。

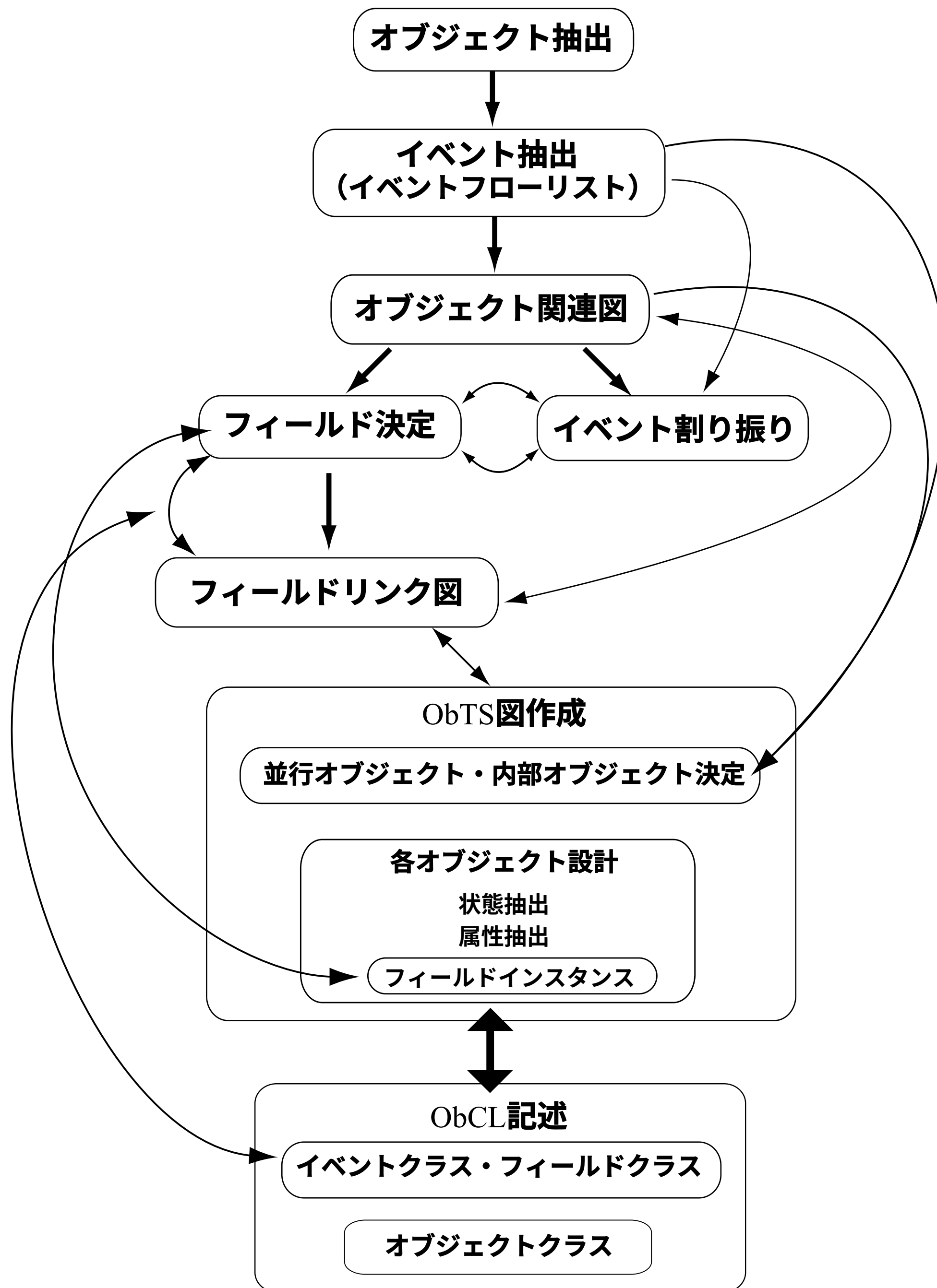

図 5.1: 開発の流れ

5.3　イベントフローリスト

ここではイベントの抽出を行う。何度も述べているが、オブジェクトの関連は通信によってのみ表される。どのオブジェクトからどのイベントが発生し、そのイベントがどのオブジェクトによって取得されるのかを、次のイベントフローリストを用いて分析する。

ここでは**イベントフローリスト**を提案している。特に形式的なシンタクスを定義するつもりはなく、そういった議論は追求しない。

$$Object1 \xrightarrow{e} Object2 \quad (e : event1)$$

または

$$Object3 \xleftarrow{E} Object4 \quad (E : \{event2, event3, event4\})$$

上記はそれぞれ、2 つのオブジェクト間の通信を記述している。2 つのオブジェクト名を記述し、その間を矢印で結ぶ。矢印の意味は尻がイベントの発生であり、矢先が取得である。矢印の方向は左右どちらでも良い。方針としてはすべての矢印の方向を右か左に揃えるものと、注目するいくつかのオブジェクトを左にもってきて矢印の向きを変えるものが考えられる。矢印の上にイベントを記述する。イベント名をそのまま記述しても良い。ここではイベント記号 e、E を用いて、その後の括弧内にイベント名を記述した。こうすることで列挙の際の簡潔さが期待できる。e は単一のイベントを示し、E はイベントの集合を表した。括弧の中には “{}” を用いて列挙により集合を記述した。イベント集合を用いるので、ある 2 つのオブジェクトの組合わせの出現は、基本的に矢印の 1 方向につき 1 度である。ただ、あえて 2 つのイベント集合に分けることは制限しない。その時のイベントの分類はフィールドを意識したものが良い。

イベントフローリストではすべての通信を列挙する。すべてとはつまり、あるイベント E について多数のオブジェクトが取得する可能性があるのならそのすべてを記述する。またあるイベントについて多数のオブジェクトから発生するのなら、そのすべてを記述する。ただ、同じイベントが他のオブジェクトから発生するようなモデル化は、本当に正しくモデル化できているか注意が必要である。

5.4　オブジェクト関連図

イベントフローリストを基にすべてのオブジェクトの関連を図示する。

ObTS はイベント通信とそれに伴うオブジェクト間の協調した振る舞いの概念を含んでいる。しかし ObTS 図では並行動作するオブジェクトの集合としてシステムが記述される。そのためオブジェクト間のリンクなど、図的表現はなされていない。遷移図に示されるイベントからどこととどこがイベントをやり取りするのか解読するのみである。

ObTS 図の補助的な図式として**オブジェクト関連図**を提案する。以下のオブジェクト関連図は ObTS 図で表されないイベントの流れによるオブジェクト間の関連を示す。

他の方法論で厳密に「オブジェクト関連図」という言葉は確認できていないが、もし他に提案されていても本論文で使うオブジェクト関連図はここで述べたものとする。概略は UML のコラボレーション図や OMT のイベントフロー図と同様である。ただし四角形のアイコンで表されるものは、先に抽出し ObTS 図に描かれうるオブジェクトまたはクラスであり、コンポーネントなどの概念でない。オブジェクト間のリンクにそっていくつかの矢印が描かれる。矢印の方向の意味は 5.3節で述べた通り。

ここではコラボレーション図のようなイベントへのナンバリングは必要ない。フィールドとイベントが抽出出来れば良い。シナリオはイベントシーケンス図を用いて示せばよい。

また、オブジェクト関連図は UML などのクラス図の「関連」の概念も含む。本研究ではオブジェクト間のリンクはイベントによってのみ関連づけられるという考えなので、この図だけですべてが表現できている。クラス図との対応を考えると、イベントの方向は「ロール」である。すべてのイベントを記述するので、そのリンクが双方向の関連なのか片方向の関連なのかが必ず図示されている。

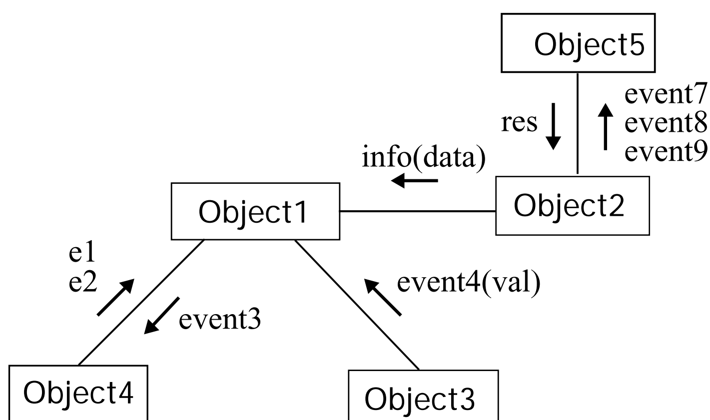

図 5.2: オブジェクト関連図

5.5　フィールドの決定

ここでイベントの括りについてもう一度見直し、フィールドとそれに属するイベントを決定する。ここでは、フィールド名は ObCL で実際に使うものをつける。オブジェクト関連図はフィールドの抽出のために用いることができる。

以下にフィールド決定のための指針を与える。これは設計仕様の直感的な分かりやすさと、フィールドプロパティの変更時に対応がしやすくなるようなものを考えている。

- 外部イベントと内部イベントではフィールドを変える
- 物理的な通信路の違いは異なるフィールドで表す
- イベントの意味的な違いを考えフィールド分けを行う
- 時には意味によるフィールドの括りも必要である
- それが外部イベントなのか、外部イベントを検知したという内部イベントなのかを明確にする
- あるいはそれら 2 つの抽象イベントなのかを考える
- 外部オブジェクト (アクター) の名前は外部イベントのフィールド名に現れる

フィールドもイベントと併せた列挙を文書化しておくとよい。その際、フィールドインスタンスも対応づけて記述する。また、イベントの属性の型も決定する。フィールドインスタンスは特にこの段階で決定する必要はない、ObCL を記述する段階で決定しフィードバックすると良い。具体的な記述は第 6 章のエレベータの開発例を参考。

5.5.1　フィールドリンク図

決定したフィールドとオブジェクト関連図を基にフィールドリンク図を作成する。ここで提案する**フィールドリンク図**は基本的にオブジェクト関連図と同様の構造をとる。オブジェクト関連図はオブジェクト間の相互作用が見て取れるが、ObCL 記述のためにはオブジェクトとフィールドの所属関係が図示されていると理解を助ける。

リンクは 1 本とは限らない。つまり、同一オブジェクト間に複数のフィールドが存在することはありえる。

またフィールドリンク図は無向グラフである。イベント通信の方向は 5.4節で示している。その上 ObCL においてフィールドに方向はなく、イベント通信の方向の制御や制限

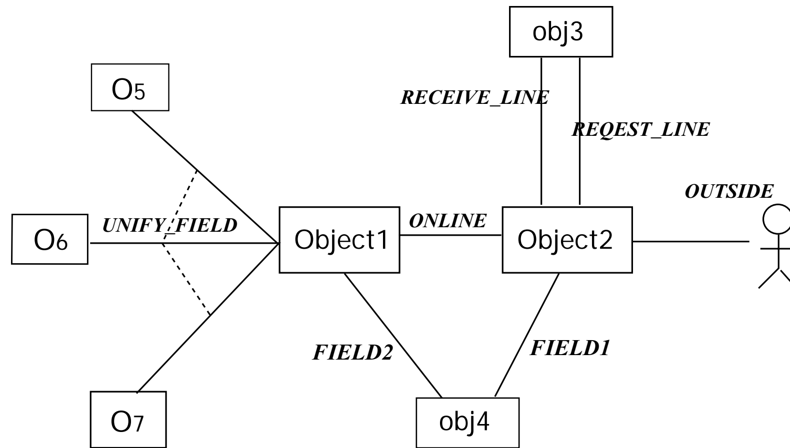

図 5.3: フィールドリンク図

もない。そのためフィールドリンク図に矢印は現れない。実際、双方向に通信が行われるフィールドがあっても良い。

同じフィールドに属するオブジェクトの記述は、できるだけオブジェクトとリンクの接続を同じ個所で表し、同じフィールドを表すリンクを点線で繋ぎフィールド名をつける。

5.6 ObTS の作図

先の設計分析をもとに、ObTS 図を記述する。

並行オブジェクトと内部オブジェクト　イベント通信によって関連づけられたオブジェクトは基本的にはすべて並行オブジェクトである。ただし、中には委譲の関連しかもっていないオブジェクトを独立させてしまうこともある。そのような場合には内部オブジェクトに直すと良い。このときイベントの意味が希薄である可能性が指摘できる。オブジェクト間のイベントがうまく抽出できないときには、内部オブジェクトで十分でないか分析する。

また内部オブジェクトは設計仕様を詳細化していくときに都合が良い場合がある。ひとつの意味でくくれるような状態遷移の集合は内部オブジェクトに階層化することで、イベント通信も簡略化することができ、可読性が向上することもある。

状態の抽出　分散システムの研究分野の議論において、あるメッセージが送られたら、それに対する処理が終り応答が返されるまで制御が奪われるような、一種の同期通信があ

る。これは本研究の通信モデルの範囲ではあつかいにくいため、このための通信モデルはつくれない。

このような通信を記述するときは状態を 2 つに分けて、応答待ち状態をつくる。応答待ち状態では応答イベント以外のイベントを受理しないことにより実現する。

フィールドとイベント　ObTS 図のイベント名にフィールドを含めた方が良い。実際に 1 つのイベント名として機能するので不都合は生じない。

先の分析でそのオブジェクトが属するフィールドは決定されている。実際にはオブジェクトはフィールドインスタンスに属するので、インスタンス名を決定する。ここでインスタンス名の適切さや簡潔さを思慮する必要がある。インスタンス名はクラス名と同名 (小文字で表す) のことも多いが、クラス名と異なる名前をつけた方がわかりやすいことも多々ある。そのようなときはおよそ、イベント名の一部として考えると的確な名前が決定できる。

決定したインスタンス名は「フィールドの決定」工程にフィードバックする。

属性の決定　オブジェクトの属性の抽出は、そのオブジェクトが常に把握しておかなければならない項目の決定である。

本研究でイベント寿命を 1 ステップ以上に変更できる記述を用意した。これはイベントバッファとみなして扱えることを先で述べた。これによって、属性をバッファのように扱う必要な無くなった。

5.7　ObCL の記述

ObTS 図を基に ObCL を記述する。ObTS と ObCL の対応は直感的にわかりやすく、スムーズに記述できるが、いくつかの考慮すべき点とフィールドプロパティ指定の指針を示す。

イベントクラス・フィールドクラス　フィールドクラスはすでに決定したものを基にそのまま記述できる。イベントの属性についても「フィールドの決定」工程において決定しておいた。

遷移規則　ObTS 図では遷移のラベルは記述されないが、ObCL 記述の遷移規則には遷移識別子が必要である。振る舞いの側面から、遷移識別子は重要なものではない。しかし、

遷移名は具体的かつ簡潔に名前をつたほうがよい。逆に意味がありすぎると状態の意味が目立たなくなったり、イベントを受理しての遷移であることを見失いがちになるため気をつける必要がある。例えば遷移先の状態名が “stop” であるとき stopping だとか、stoper といった名前をつけるべきでない。具体例は次章のエレベータの記述例を参考。

フィールドプロパティ　はじめはフィールドプロパティを指定しなくてもよい。適度に抽象化された仕様は通信もまた抽象化されている。そのとき通信モデルは理想状態で記述できる。本研究ではデフォルトのフィールドプロパティを遅延時間 0、寿命 1 にしてある。これはもともとの ObCL の通信モデルの値だが、これは理想化された通信モデルのパラメータ値である。設計の詳細化にともない通信モデルも具体化していくと良い。

第 6 章

適用例

組み込みシステムの代表例として挙げられるエレベータは、スケジューリング部分の手続き的な要素を除けば、分かりやすいリアクティブシステムであり、より良い ObTS の適用対象である。しかし、エレベータの仕組みはその構造やイベントの発生個所などいたって分散的であるため、通信モデルを柔軟に変化させてモデル化することが有用であることを示す。

本研究と直接関係のない個所もあるが、方法論の適用の例と ObTS の適用例としてその全体を掲載する。ObTS/ObCL による例題は過去にいくつも記述されてきたが、その設計の工程を文章で残したものはない。特にここに掲載するものは第 5 章の指針に沿って設計された、その工程の記録である。

6.1 仮想的なエレベータのモデル化

図 6.1にエレベータの外観を示すが、以下のような言葉の定義と仮想的な要件仕様を前提に進める。

かご室 実際に人が乗り、各階に運ぶために移動する箱。また室内はかご室内

フロアボタン 各階にある、かご室を呼ぶためのボタン

呼 フロアボタンからの呼び出しや室内ボタンによる行き先指示の総称

リフト ここではかご室を実際に動かすモータを含む装置のこととする

コントローラ エレベータの運行の制御を行う。

スケジューリング いくつかの呼の中から次に向かう (止まる) 階を決定する。ここではごく簡単なものを考える

エレベータ またはエレベータシステム。ここでの記述対象であるエレベータ全体

- 呼がある階には必ずいつか行く
- 制約 ドアが開いているときにかご室は移動しない
- 制約 かご室が停止状態でなければドアは開かない

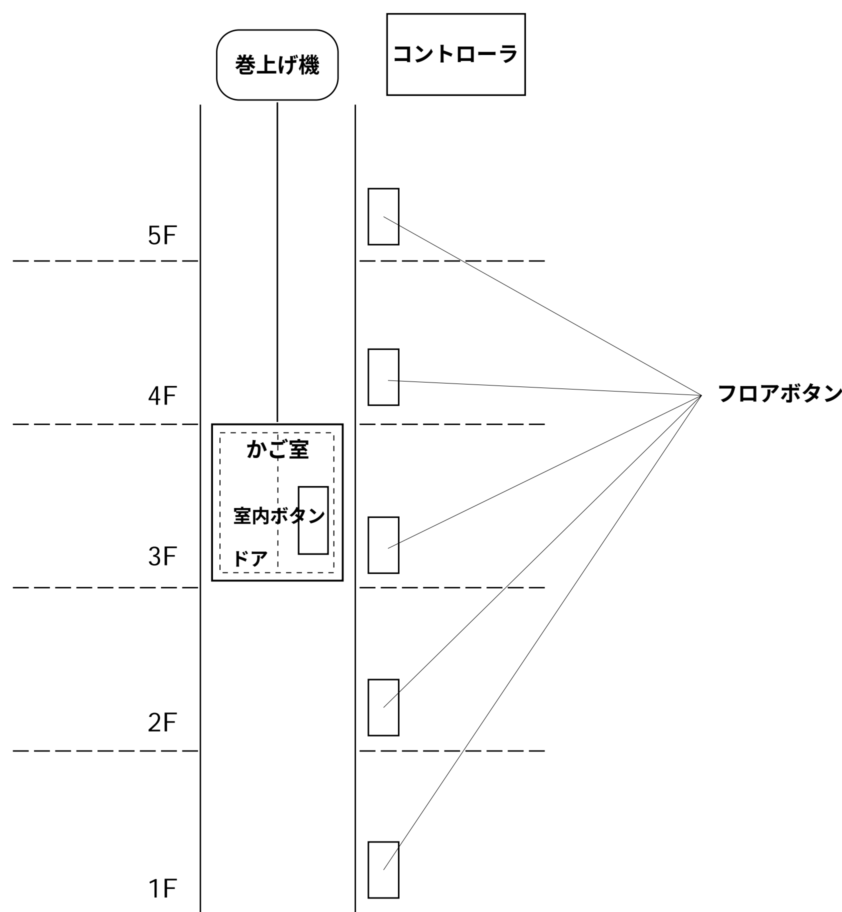

図 6.1: エレベータの外観

また、図 6.1のような概略図を描いてみることも大切である。例えば、当たり前だと思っていたイメージが他人と異なっていることに気づくかも知れない。あるいは組み込みシス

テムなどでは物理的なパーツが 1 つのオブジェクトとして切り出せることもあり、設計分析を助けることもある。

ここで実際に設計しなければならないのは、制御のための組み込みシステムであるコントローラになる。しかし本節では、制御すべき対象も含めたエレベータ全体を 1 つのシステムとしてモデル化を行う。これはシミュレーションとしての分かりやすさが得られ、また制御対象とのインタラクションも正しくシミュレートするためである。

なお本論文ではある程度整ったものを掲載しているが、実際には何度もの試行錯誤の上で設計されたものである。

6.2 設計分析

段階的に設計していく過程を追うことにする。まず階数は 5 階を想定し、1 つのエレベータについて設計する。また、フロアボタンは上下ボタンの区別をつけない。

オブジェクトの抽出　エレベータを構成するオブジェクトを抽出する。図 6.1におおよそのオブジェクトが出現していることが分かる。

— コントローラ
— かご室
— リフト
— ドア
— 室内ボタン
— フロアボタン

ここで、かご室をオブジェクトとして挙げたが本来は必要のないオブジェクトである。現実世界ののモデル化ならばむしろ中心的オブジェクトだと考えられるが、設計対象はコントローラであり、かご室はコントローラによって制御されるようなものではない。かご室の実際の移動はリフトの動作によるものであり、室内ボタンも独立したオブジェクトとして挙げている。

かご室をオブジェクトとして挙げたのは、シミュレーションの分かりやすさのためである。実際、最終的に ObCL を記述しシミュレートした結果、ObML の対話型実行では表 6.1のような状態表示をした。各ステップの 1 行目にかご室オブジェクト box の状態である、現在階が見て取れる。これは直感的に分かりやすい表示である。

また、かご室を現在階を把握するためのセンサーだと考えても良い。

```
Step:1
  ELEVATOR_box:State first,(ObjectID=Int 0;ObjectName=String "box")
  ELEVATOR_door:State closed,(ObjectID=Int 1;ObjectName=String "door")
  ELEVATOR_lift:State stop,(ObjectID=Int 2;ObjectName=String "lift")
  ELEVATOR_controler:State standby,(ObjectID=Int 3;ObjectName=String
:fb.come(n=Int 3)
fb.come(n=Int 3)
Step:2
  ELEVATOR_box:State first,(ObjectID=Int 0;ObjectName=String "box")
  ELEVATOR_door:State closed,(ObjectID=Int 1;ObjectName=String "door")
  ELEVATOR_lift:State stop,(ObjectID=Int 2;ObjectName=String "lift")
  ELEVATOR_controler:State moving_up,(ObjectID=Int 3;ObjectName=String
  move.up($transinfo=String "CONTROLLER:controler:push_fb3";$inputinfo
:
Step:3
  ELEVATOR_box:State first,(ObjectID=Int 0;ObjectName=String "box")
  ELEVATOR_door:State closed,(ObjectID=Int 1;ObjectName=String "door")
  ELEVATOR_lift:State up,(ObjectID=Int 2;ObjectName=String "lift")
  ELEVATOR_controler:State moving_up,(ObjectID=Int 3;ObjectName=String
  box.up($transinfo=String "LIFT:lift:work_up1";$inputinfo=String
:
Step:4
  ELEVATOR_box:State second,(ObjectID=Int 0;ObjectName=String "box")
  ELEVATOR_door:State closed,(ObjectID=Int 1;ObjectName=String "door")
  ELEVATOR_lift:State up,(ObjectID=Int 2;ObjectName=String "lift")
  ELEVATOR_controler:State moving_up,(ObjectID=Int 3;ObjectName=String
  floor.current(n=Int 2;$transinfo=String "BOX:box:upto2nd";$inputinfo
:
```

表 6.1: 対話実行の様子

イベントの抽出　先に挙げたオブジェクトの間にどのようなイベントがやり取りされるか分析する。

まず分かりやすいのが「呼」である。各階のフロアボタンと室内ボタンよりコントローラへ呼が送られる。

$$\text{コントローラ} \overset{e}{\longleftarrow} \text{フロアボタン} \quad (e : \text{呼 (階数)})$$

$$\text{コントローラ} \overset{e}{\longleftarrow} \text{室内ボタン} \quad (e : \text{呼 (階数)})$$

またコントローラはリフト、ドアを制御する。

$$\text{コントローラ} \overset{E}{\longrightarrow} \text{リフト} \quad (E : \{\text{停止, 上昇, 下降}\})$$

$$\text{コントローラ} \overset{E}{\longrightarrow} \text{ドア} \quad (E : \{\text{開, 閉}\})$$

逆にリフト、ドアからコントローラへは動作が完了したことを知らせる応答が必要である。この応答がないと先に提示した制約などが厳密に記述できない。注意深く設計すれば応答イベントを使わずに制約を満たすことができるが、実際の組み込みシステムには必要となることが予測できる。またそういった制約があることを明示することにもなり、明瞭な仕様になる。

応答があるということは、ドア、リフトには簡単な制御システムが入っていることを仮定している。仮に実装が、簡単な機器と動作検知のためのセンサーに分かれたものになっても問題はない。

$$\text{コントローラ} \overset{e}{\longleftarrow} \text{リフト} \quad (e : \text{停止応答})$$

$$\text{コントローラ} \overset{E}{\longleftarrow} \text{ドア} \quad (E : \{\text{開放, 閉切完了}\})$$

最後にかご室だが、かご室はリフトによってワイヤーで吊り上げられている。リフトの動作とかご室の動作は完全に一致する。そこで、ここではかご室への作用はコントローラからリフトへの作用と同じものとして設計する。また、かご室は現在階をコントローラに知らせる。

$$\text{コントローラ} \overset{E}{\longrightarrow} \text{かご室} \quad (E : \{\text{停止, 上昇, 下降}\})$$

$$\text{コントローラ} \overset{e}{\longleftarrow} \text{かご室} \quad (e : \text{現在階})$$

オブジェクト間の関連　イベントの抽出に基づきオブジェクトの関連を記述する。すると図 6.2のようになる。

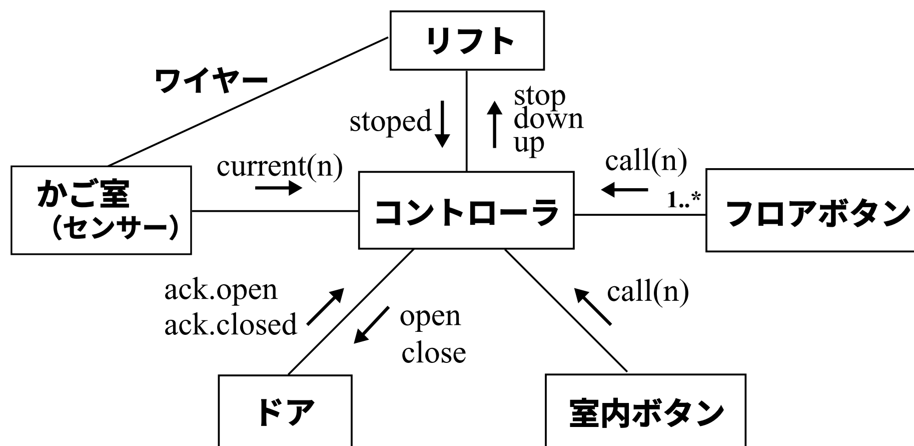

図 6.2: オブジェクト関連図

フィールドの決定　図 6.2よりフィールドを決定する。また、各フィールドに属するイベントも割り振る。

ここでフロアボタンと室内ボタンを外部オブジェクトとしたい。なぜなら、コントローラにとっては呼が重要であり、呼イベントの発生は「人が呼び出し/行き先ボタンを押した」という現象にまで抽象できる。もっとも実際のボタン (オブジェクト) は押されたら点灯し、かご室が到着したら消えるという状態があり、オブジェクトとしてモデル化が可能である。しかしそのような設計を行うのは少々煩雑であり、ここでは両オブジェクトを作らない。よって第 5 章の方針に従い、以下のように両ボタン名はフィールド名として表した。

また応答という意味で、ドアの応答フィールドとリフト停止応答のフィールドを同じにした。さらに “call” は ObCL の予約語であるため、呼イベントの名前を go と come に変更した。

- fb : FLOORBUTTON
  - come(n:Int)
- ib : INSIDEBUTTON
  - go(n:Int)
- move : CONTROL_LIFT

  - stop
  - down
  - up

- door : CONTROL_DOOR
  - close
  - open

- ack : ACK
  - stopped
  - open
  - closed

- floor : FLOOR
  - current(n:Int)

以上にフィールドと属するイベントを抽出した。これと併せて、図 6.3にフィールドリンク図を描く。

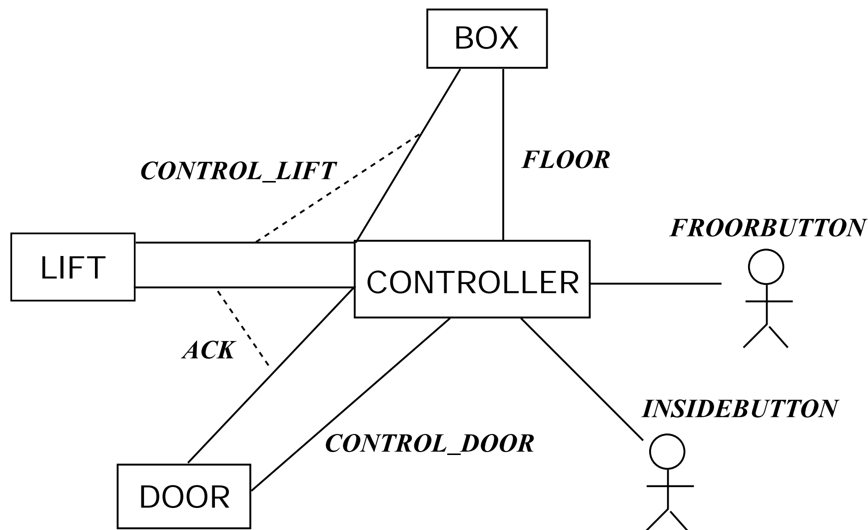

図 6.3: エレベータシステムのフィールドリンク

6.3 ObTS/ObCL 記述 (第 1 段階)

設計分析が一通り終わったので、ObTS/ObCL による繰り返し設計に移る。

手順としては ObTS 図を描いて、大体の設計を一通り行い。それをもとに ObCL 記述を行った。しかし本論文では対応が見やすいように、オブジェクトごとに ObTS 図と ObCL 記述を交互に掲載する。

6.3.1 並行オブジェクト

エレベータは現実問題なので、オブジェクトのすべてが並行オブジェクトであることは少し考えればわかる。

まず始めに図 6.4に全体の ObTS 図を抽象的に書いたものを示す。これは実際には各オブジェクト内には詳細に状態遷移図が書かれているものである。各オブジェクトの状態遷移図は以降に掲載する。

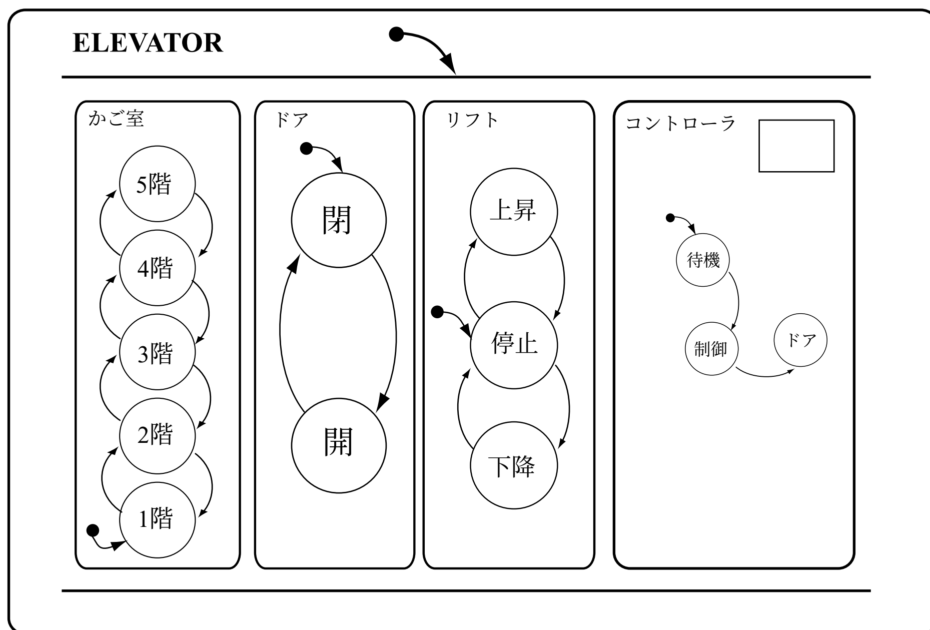

図 6.4: 全体の ObTS 図 (細部省略)

6.3.2 イベントクラスとフィールドクラス (ObCL 記述)

先の分析により、ここの工程は簡単に終わる。表 6.2 にイベントクラスとフィールドクラスの ObCL 記述を載せる。呼イベントや current イベントの属性である階数は Int 型としている。

```
--Int 型属性つきイベント
event EVENT_WITH_INT
  inherit GENERIC_EVENT
  attribute n:Int
end

-- 呼び出しボタン (各階&エレベータ内のボタン)
field CALLBUTTON
  event call:EVENT_WITH_INT
end

-- リフト制御
field CONTROL_LIFT
  event up,down,stop:GENERIC_EVENT
end

-- ドア制御
field CONTROL_DOOR
  event open,close:GENERIC_EVENT
end

-- 応答
field ACK
  event stopped,open,closed:GENERIC_EVENT
end

-- 現在の階
field FLOOR
  event current:EVENT_WITH_INT
end
```

表 6.2: イベントクラスとフィールドクラス

6.3.3 オブジェクトの設計

各オブジェクトについて設計する。

ドア　ドアの状態は基本的に開いた状態と閉じた状態である。詳細な設計を進めていくと、開きつつある状態や閉じつつある状態を考える必要があるが、ここでは単純に 2 つの状態を持つものとする。

フィールドはコントローラからの制御のための CONTROL_DOOR と応答のための ACK

フィールドがある。ここで制御の対象を簡潔に表現するために、フィールドインスタンス名を "door" とした。

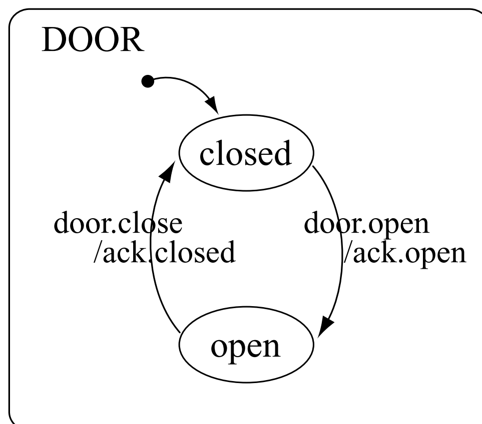

図 6.5: ドアの開閉

```
-- ドア
class DOOR
  field door:CONTROL_DOOR
  field ack:ACK
  state closed,open

  transition
  start is
    source init
    destination closed     -- 初期状態は「閉」
  end
  work_door1 is            -- 「ドアを開く」遷移
    source closed
    input door.open          -- 「ドア開け」
    destination open
    output ack.open          -- 「開きました」
  end                      --
  work_door2 is            -- 「ドアを閉じる」遷移
    source open
    input door.close         -- 「ドア閉じろ」
    destination closed
    output ack.closed        -- 「閉じました」
  end                      --
end
```

表 6.3: ドアクラス記述

リフト　リフトの状態は停止状態、巻き上げ中、巻き下ろし中の 3 つがある。フィールドはコントローラからの制御である CONTROL_LIFT がある。実際のフィールドインスタンス名はエレベータを実際に動かす作用であるという意味で "move" とした。また停止応答のための ACK フィールドがあり、停止状態へのすべての遷移には出力イベントとして ack.stopped が記述される。

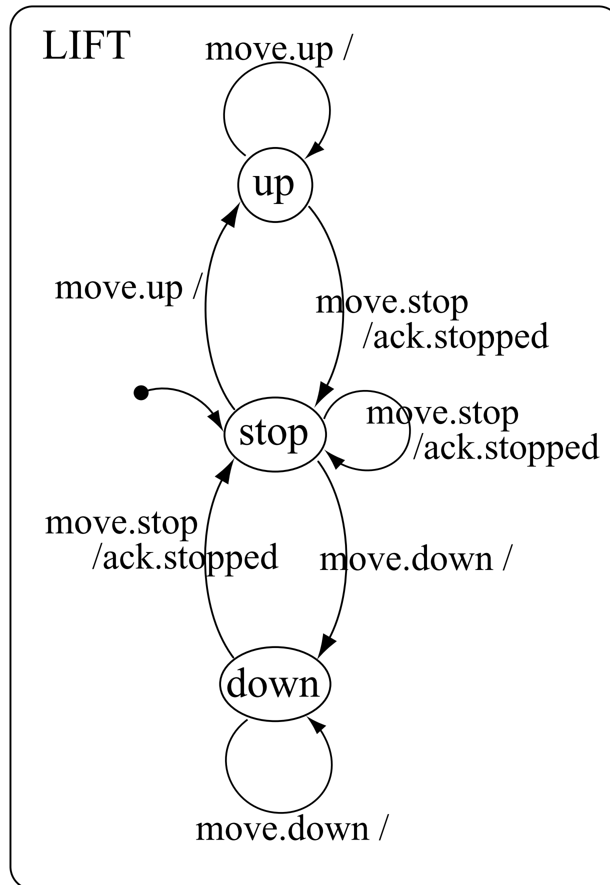

図 6.6: リフトの動き

```
-- リフト
class LIFT
  field move:CONTROL_LIFT
  field ack:ACK
  state stop,up,down

  transition
  start is
    source init
    destination stop
  end

  work_up is
    source stop
    input move.stop
    destination stop
    output ack.stopped
  end

  work_up1 is
    source stop
    input move.up
    destination up
  end
  work_up2 is
    source up
    input move.up
    destination up
  end
  work_up3 is
    source up
    input move.stop
    destination stop
    output ack.stopped
  end

  work_down1 is
    source stop
    input move.down
    destination down
  end
  work_down2 is
    source down
    input move.down
    destination down
  end
  work_down3 is
    source down
    input move.stop
    destination stop
    output ack.stopped
  end
end
```

表 6.4: リフトクラス記述

かご室　かご室オブジェクトはシミュレーションの分かりやすさのためだと先で述べた。エレベータの動きが分かりやすいように、かご室オブジェクトはすべての階を状態として記述する。対象の階数を変更するときに若干のコストがかかるが、ここでは分かりやすさを優先した。

先に述べたように、リフトと同じフィールド move より作用を受ける。イベント move.stop については無視する。これは階の変化のためには停止作用は必要ないためである。

また、各階 (状態) に移るごとにどの階に来たのかを知らせる。これは floor フィールドの current イベントである。

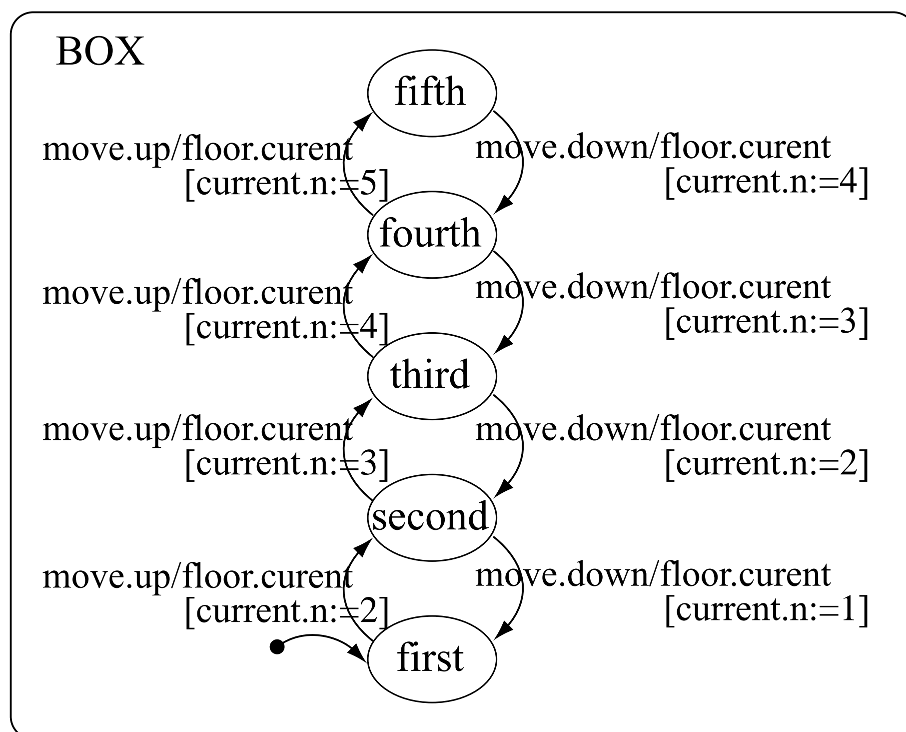

図 6.7: エレベータの階

```
-- エレベータの現在位置
class BOX
  field move:CONTROL_LIFT
  field floor:FLOOR
  state first,second,third,fourth,fifth

  transition
  start is
    source init
    destination first
  end
  upto2nd is
    source first
    input move.up
    do floor.current.n := 2
    destination second
    output floor.current
  end
  upto3rd is
    source second
    input move.up
    do floor.current.n := 3
    destination third
    output floor.current
  end
  upto4th is
    source third
    input move.up
    do floor.current.n := 4
    destination fourth
    output floor.current
  end
  upto5th is
    source fourth
    input move.up
    do floor.current.n := 5
    destination fifth
    output floor.current
  end
  downto4th is
    source fifth
    input move.down
    do floor.current.n := 4
    destination fourth
    output floor.current
  end
  downto3rd is
    source fourth
    input move.down
    do floor.current.n := 3
    destination third
    output floor.current
  end
  downto2nd is
    source third
    input move.down
    do floor.current.n := 2
    destination second
    output floor.current
  end
  downto1st is
    source second
    input move.down
    do floor.current.n := 1
    destination first
    output floor.current
  end
end
```

表 6.5: かご室クラス記述

コントローラ　コントローラはリフトとドアを制御する。コントローラは始めに挙げた制約を考慮して設計する。特に考慮しなければいけないのは応答待ち状態を記述することである。応答以外のイベントを受け付けないことにより、制約を満たす。

属性を考える。コントローラが常に把握しておかなければいけないのは、現在階と次に向かう階である。これより、属性 currentfloor と where を記述する。

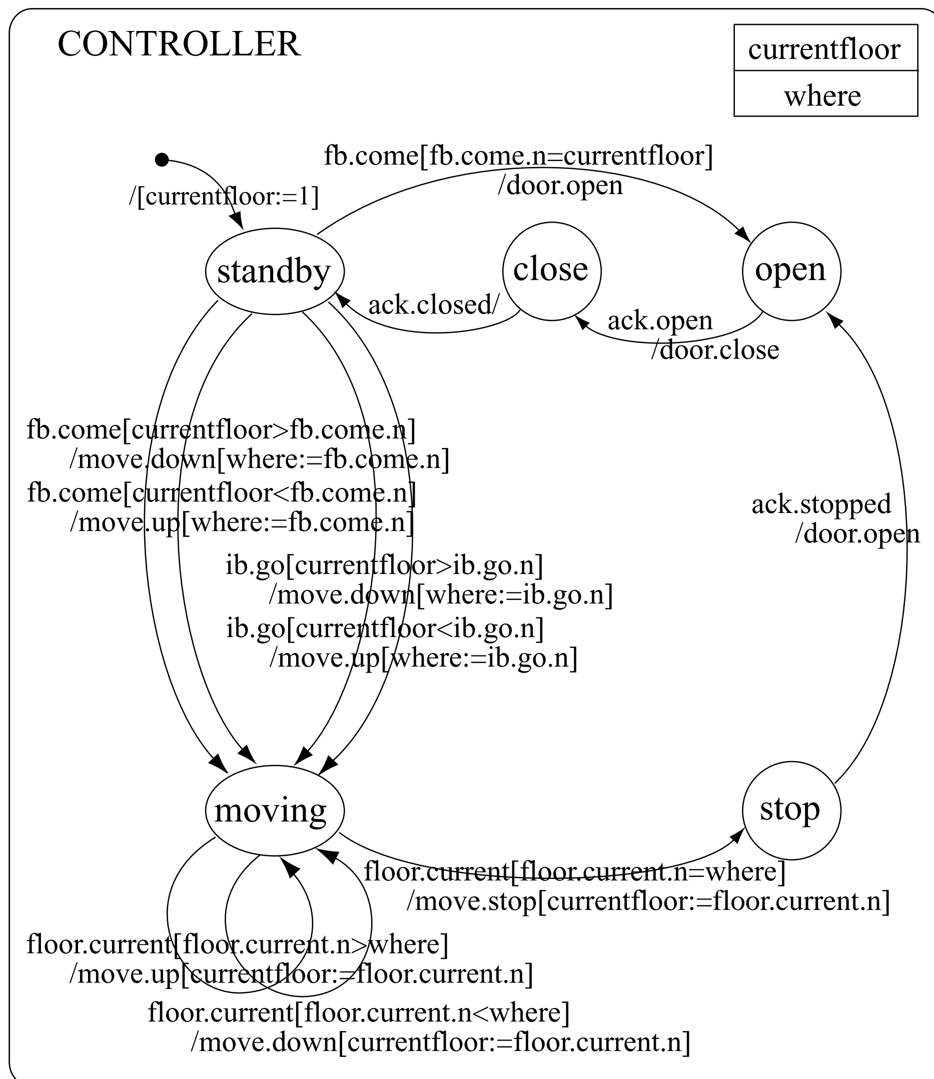

図 6.8: コントローラー

```
-- コントローラー
class CONTROLLER
  field fb:FLOORBUTTON
  field ib:INSIDEBUTTON
  field move:CONTROL_LIFT
  field door:CONTROL_DOOR
  field floor:FLOOR
  field ack:ACK

  attribute currentfloor:Int
  attribute where:Int

  state standby,moving,stop,open,close

  transition
  start is
    source init
    do currentfloor := 1
    destination standby
  end
  push_fb1 is
    source standby
    input fb.come
    when fb.come.n = currentfloor
    destination open
    output door.open
  end
  push_fb2 is
    source standby
    input fb.come
    when fb.come.n < currentfloor
    do where := fb.come.n
    destination moving
    output move.down
  end
  push_fb3 is
    source standby
    input fb.come
    when fb.come.n > currentfloor
    do where := fb.come.n
    destination moving
    output move.up
  end
```

表 6.6: コントローラクラス記述

コントローラクラスの続き

```
    push_ib1 is
      source standby
      input ib.go
      when ib.go.n < currentfloor
      do where := ib.go.n
      destination moving
      output move.down
    end
    push_ib2 is
      source standby
      input ib.go
      when ib.go.n > currentfloor
      do where := ib.go.n
      destination moving
      output move.up
    end
    situation1 is
      source moving
      input floor.current
      when floor.current.n = where
      do currentfloor := floor.current.n
      destination stop
      output move.stop
    end
    situation2 is
      source moving
      input floor.current
      when floor.current.n < where
      do currentfloor := floor.current.n
      destination moving
      output move.up
    end
    situation3 is
      source moving
      input floor.current
      when floor.current.n > where
      do currentfloor := floor.current.n
      destination moving
      output move.down
    end
    door1 is
      source stop
      input ack.stopped
      destination open
      output door.open
    end
    door2 is
      source open
      input ack.open
      destination close
      output door.close
    end
    door3 is
      source close
      input ack.closed
      destination standby
    end
  end
```

エレベータシステム　以上でエレベータのすべてのオブジェクトを記述した。
　最後にシステム記述 (表 6.7) により各インスタンスを生成する。

```
system ELEVATOR
  object total:
         {
          controler:CONTROLLER;
          lift:LIFT;
          door:DOOR;
          box:BOX
         }
  transition
    start is
      source init
      destination total
    end
end
```

表 6.7: システム記述

　これらを ObML 上でシミュレートすると、はじめに示した表 6.1の様な実行結果が得られる。

6.4　詳細化 (第 2 段階)

ここまでのエレベータは 1 つの呼しか受け付けないため現実的でない。そこで次はすべての呼に対応するエレベータの記述を行う。

6.4.1　フィールドプロパティ

ここですべての呼に対応することについて考察する。エレベータのボタンを押すのは非決定的かつ非同期的は現象である。つまり、go イベントと come イベントがいつどれだけ発生しても、コントローラオブジェクトがすべてを取得する必要がある。このとき、フィールド FLOORBUTTON と INSIDEBUTTON のフィールドプロパティによりイベント寿命を無期限 “Indef” にすることで対処できる。

表 6.8 に示した通り、フィールドプロパティKEEP_ON をつくり、それぞれにインポートした。また、ここではイベント遅延時間は考慮する必要がない。

```
fieldproperty KEEP_ON
  life Indef
end

-- 各階のボタン
field FLOORBUTTON
  import KEEP_ON
  event come:EVENT_WITH_INT
end

-- エレベータ内のボタン
field INSIDEBUTTON
  import KEEP_ON
  event go:EVENT_WITH_INT
end
```

表 6.8: 呼をためるフィールド

6.4.2　内部オブジェクト

途中まで記述するとドアの制御を端々で行うことがわかるため、再利用性を高めるためにドアの制御部を内部オブジェクトとして独立させる。図 6.8と比較されたい。このオブジェクトのために control:CONTROL フィールドが作られた。これは委譲の終了制御のための ok イベントを持つ。少々本質的でないが、再利用性と可読性の向上がめざましいので採用する。

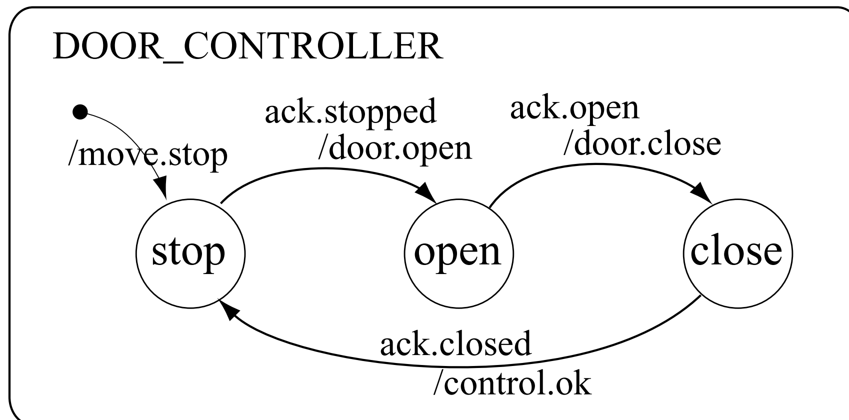

図 6.9: ドアの制御部のクラス

このクラスは実際には次の図 6.10に小文字の door_controlle として何度も状態として出現している。door_controlle に遷移すると、実際には内部オブジェクトなので委譲される。内部オブジェクトから control.ok イベントが発生すると遷移が起り、委譲が終わる。

図 6.10では fb.come と ib.go イベントが入力である遷移をまとめて一つの遷移として表している。これはただの簡略のためだけであり厳密でない。イベント名は call としているがこれは ObCL の予約語であるため使えない。実際の ObCL 記述には 2 つのイベントと 2 つの遷移として正しく記述している。

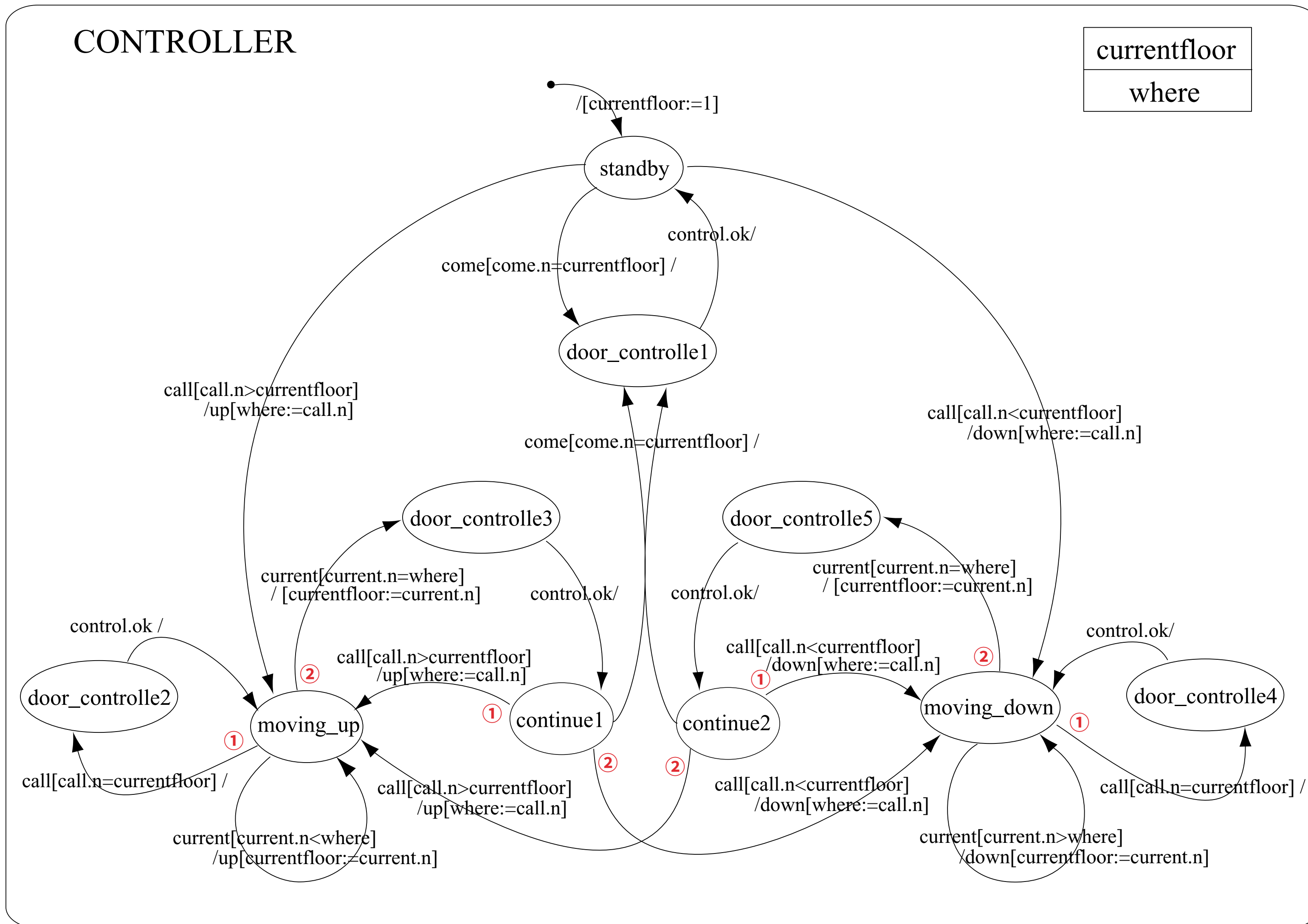

図 6.10: すべての「呼」を処理する仕様

第 7 章

評価と考察

方法論については主観的な意見にとどまり議論できないが、実際に指針にしたがってエレベータを設計した上では、円滑な設計ができたと考える。今回は ObTS と通信モデルの考察を用いた開発手法の明文化を試みたが、いままでの設計においても無意識にこのようなプロセスを経ていたと考える。そのため、当方法論は抵抗無く導入できるものである。また、オブジェクトモデルなどを用いなくとも動的モデルを中心とした設計を行える開発手法となったため、オブジェクト指向の知識はまだないが、Statechart は昔から使ってきているような現場への適用が円滑に行えるのではないかと考える。そのような場合、機能モデルに相当する部分は無理にオブジェクト指向開発手法に従わなくとも、熟れ親しんだ従来の手法が適用できると考える。

本研究において、ObCL/ObML への拡張を行ったフィールドプロパティについて評価する。第 6 章において最終的にフィールドプロパティを用いた設計を行った。ここではイベント寿命を無期限に設定し、イベントバッファをもつコントローラを設計した。実際にエレベータは非同期的に呼を受け付ける必要のあるシステムであるので、これは有効に機能した。もしフィールドプロパティがなければ、バッファとしてオブジェクトに配列型の属性を設け逐次的な作業を記述する必要がある。これでは煩雑な記述となるため、フィールドプロパティが開発効率と可読性を向上させた評価となる。

第 3 章で通信が起るかどうかは保証できないことを述べた。そのこここと同様に、発生したイベントが必ず受理されるという制約を ObCL で記述することはできない。しかし、このような制約についてテストしたいとき次のように確認できる。まず、フィールドプロパティにおいてイベント寿命を Indef に設定する。これにより望むべき処理についてシミュレートが終わったあとも何らかのイベントが残ったならば、そのシステムは制約を満たさなかったことが確認できる。

第 8 章

今後の課題

通信モデルの分離を目指し、問題領域を通信モデルに限定したあまり、実行モデルに依存した通信について追求できなかった。具体的には通信におけるオブジェクトの優先度などがある。本研究の成果では実行モデルとイベントとの接点は発生と取得のみとしている。このイベント境界と通信モデルの関わり合いについて更なる議論が待たれる。今後の課題としたい。

どのような時に、どういったフィールドプロパティを設定したらいいかの議論と指針の提供ができるが、今回はそこまで追求できなかった。いくつかの代表的な通信を選択し、それらに対して「通信モデルパターン」として、フィールドプロパティの既定値を提供する必要がある。

イベント寿命は時間について有限のバッファであったが、長さが有限のバッファを扱う通信モデルも考えられる。ネットワーク通信などではバッファがいっぱいになったら次からのイベントが破棄されるような通信モデルが必要となるときもある。

第 9 章

まとめ

本研究では、まずオブジェクト指向開発における通信の定義を行った。特に動的モデルのイベントを中心的に捉えたもので、オブジェクトがイベントを発生し、オブジェクトがイベントを取得するという簡潔な定義である。それに基づき通信モデルの概念を形成した。基本はイベントの振る舞いを扱う計算モデルというものである。イベントを扱い考察するためにイベント空間を提案した。そしてイベント空間が唯一イベントの存在する領域であり、通信モデルで扱う範囲とした。またオブジェクトの振る舞いは実行モデルによって定義される。そして、通信モデルの考察はパラメータ抽出であり、通信モデルの変更はパラメータの値の変動であるとした。そこから今回、イベント遅延時間、イベント寿命などのパラメータを考えた。

これらの概念を用いて、ObCL/ObML に対して拡張を施した。具体的には、各フィールドをイベント空間とみなし、それぞれ独立した通信モデルを持つことが可能となるものである。これを扱う ObCL 記述としてフィールドプロパティを提案し、実装した。

実際に通信モデルを念頭に置いた開発を行うことは難しいため、そのための方法論を提案した。ObTS/ObCL を中心にとらえた開発プロセスの指針であり、具体的にイベントフローリスト、オブジェクト関連図、フィールドリンク図といった設計分析手法を提案している。また本研究の成果を適用した、イベントと通信の意味が一元的な開発プロセスでもある。これらを用いて ObTS/ObCL の記述が円滑に行えることを、実際に仮想的なエレベータの設計に適用し実践した。結果は円滑な設計が行えたが、フィールドプロパティに関しては十分な例題とはならなかった。

通信モデルのパラメータはまだ多く抽出さておらず多くの議論ができなかったが、通信モデルの議論は始まったばかりであり、今後の発展が期待できる。

謝辞

本研究を進めるにあたり、多大な御指導を賜った片山卓也教授に深く感謝致します。また日頃から有益な助言など御支援頂いた伊藤恵助手に心より感謝致します。

キャノンソフトウェア株式会社の久保秋真氏には大変お忙しい中、本研究に対する議論、助言を頂きました。心より御礼申し上げます。また本論文をまとめるにあたって、さまざまな面で協力頂いたソフトウェア基礎講座の皆様に感謝致します。

参考文献

[1] 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. In 大蒔和仁(編), FOSE '95, レクチャーノート/ソフトウェア学, No.15, pp. 51-60, 日本ソフトウェア科学会, 近代科学社, 1996

[2] 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, コンピュータソフトウェア, Vol. 14, No. 2, pp. 22-37, 1997

[3] Michael von der Beeck, A Comparison of Statecharts Variants, LNCS 863, Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 128-148, 1994

[4] 久保秋真, 伊藤恵, 片山卓也, ObTS に基づくオブジェクト指向仕様記述言語とその支援環境, 日本ソフトウェア科学会 第 14 回 全国大会論文集, pp. 589-592, 1997

[5] David Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, pp. 231-274, 1987

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

[7] 伊藤恵, オブジェクト指向方法論のための動的モデルの形式化の研究, 北陸先端科学技術大学院大学 博士論文 (1998)

[8] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, Object-Oriented Modeling and Design, Prentice Hall, 1991. (オブジェクト指向方法論 OMT, 羽生田栄一監訳, トッパン)

[9] Grady Booch, Object-Oriented Analysis and Design with Applications, Second Edition, Addison-Wesley, 1994, (Booch 法：オブジェクト指向分析と設計 第 2 版, 山城 他 訳, アジソン・ウェスレイ)

[10] OMG Unified Modeling Language Specification, version 1.3, OMG, 1999

[11] John J. Marciniak Editor-in-chief, Encyclopedia of Software Engineering, (ソフトウェア工学大事典, 片山卓也/土居範久/鳥居宏次 監訳, 朝倉書店)

[12] Edited by Jan Van Leeuwen, Formal Models and Semantics, (形式的モデルと意味論, 廣瀬健/野崎昭弘/小林孝次郎 監訳, 丸善)

[13] 米澤明憲, 柴山悦哉, モデルと表現, 岩波書店