ソフトウェア工学 89−17
（１９９２．１２．１５）

形式的仕様記述言語ＬＯＴＯＳの記述スタイルについて

内藤 晋　　佐伯元司
東京工業大学 工学部

あらまし

ソフトウェアの仕様を効果的に記述するためにパターン化された手法が記述スタイルである。本論文では、形式的仕様記述言語ＬＯＴＯＳを用いて、この記述スタイルについて述べる。対象をとらえる「構造的」「機能的」「動作的」の３つの観点から、基本的な４つの記述スタイル 1) オブジェクト指向型, 2) 機能指向型,3) イベント指向型, 4) 状態指向型, を図や記述の具体例を交えて紹介する。さらに、記述スタイル間に階層的関係を想定し、大きな対象システムを記述する際の方法について述べる。最後に、「酒屋の在庫管理問題」をそれぞれの記述スタイルを用いて記述し、考察をおこなう。

和文キーワード：　仕様記述言語, ＬＯＴＯＳ, 記述スタイル.

Specification Styles For Formal Specification Language LOTOS

Susumu Naitou　　Motosi Saeki
Faculty of Engineering, Tokyo Institute of Technology

Abstract

Specification styles are techniques for structuring software specifications.In this paper, we report specification styles in formal specification language LOTOS. From three viewpoints – structural, functional and behavioral views – ,we introduce four basic specification styles – 1)Object oriented style, 2)Function oriented style, 3)Event oriented style, 4)State oriented style – with figures and concrete examples. We assume a hierarchy in specification styles and explain a method for descri--bing large systems. We describe “an Inventory Control System for liquor shops” by using each specification style and discuss them.

English Keywords: Specification Language, LOTOS, Specification Style.

1 まえがき

ソフトウェアの仕様を形式的に記述するための言語として、様々な仕様記述言語が提案されている。仕様を形式的に記述する利点は、

- 意味に曖昧性がないため、コミュニケーションの手段として使用できる。
- 構文チェックにより、仕様書の未定義および二重定義部分を検出できるのに加えて、検証系が用意されている場合には、無矛盾性のチェックなど意味的なチェックが行なえる。
- 実行可能の場合、プロトタイプとしても使用できる。

などである。

しかし、形式的仕様記述言語は、独特の構文規則・意味モデルによって構成されているので、その言語への習熟なしで読解／記述を行なうことは困難である。

形式的仕様記述言語を使って仕様を記述する際の手法は，種々考えられるが、そのうちの効果的なものをまとめてパターン化することが可能と思われる。このパターン化された手法に従って、仕様化を進めることにより、誤りの少ない形式的仕様を比較的短い時間で書きあげることができる。また、同じパターンを用いて記述された仕様の構造や記法は類似しているはずである。複数の作業者が分担して仕様を記述する場合では，この各自が作り上げる仕様の構造、記法の統一は重要である。本稿では、このパターンを記述スタイルと呼び、それに関する考察をおこなう。

Vissers らは通信システムの分野で 4 つの Specification Style を考案したが [1]、実際のモデル化手法との関連が薄く、一般のシステムに対しても十分な効果を発揮するのは難しいと思われる。本稿で提案する記述スタイルは一般的なモデル化の観点である「構造的観点」、「機能的観点」、「動作的観点」の三つの観点から構成される。

本研究では、形式的仕様記述言語の例として、ＬＯＴＯＳを選択する。ＬＯＴＯＳを使用する理由は、二つある。

1. ＩＳＯで国際標準として認められた言語であり、多くの研究がなされている。
2. 操作的な意味論が与えられており、それにしたがって実行させるツールも開発されている。

異なる手法で記述を行なう際、実際に実行して評価することができるのは大きな利点である。

2 章では、ＬＯＴＯＳの基本的な構文規則についての説明をおこなう。3 章では、各種記述スタイルを簡単な例をまじえて紹介する。4 章では、それぞれのスタイルの階層的な関係についてふれる。5 章で評価および考察を加える。

2 形式的仕様記述言語ＬＯＴＯＳ

ＬＯＴＯＳでは、対象を外部から見た時に観測されるイベント（アクション）の順序を規定することにより、対象を記述する。ＬＯＴＯＳの記述構造は以下のようなものである。

```
specification[a,b,c]:noexit

  抽象データ型の記述言語ＡＣＴＯＮＥ
  に基づいたデータ型定義

behaviour
  < behaviour-expression >
where
  process Proc[a,b]:noexit :=
    < behaviour-expression >
  where
    process SubProc[a]:noexit :=
      < behaviour-expression >
    endproc
  ...
  endproc
  ...
endspec
```

図１：ＬＯＴＯＳの基本構造の例

対象システムは複数の「プロセス」から成っていると考える。「プロセス」は環境（自分の外部にあるものすべて）と交信する「ゲート」（図中の a, b, c）を持ち、これを介して環境とデータのやりとりをする。プロセスは、反応を起こすゲートを< behaviour-expression >部に規定することによって記述される。

以下に、ＬＯＴＯＳを構成する基本的な構文の説明を載せる。詳しい説明は、参考文献 [5] を参照のこと。

2.1 逐次実行

発生するイベントの時系列的順序を規定するものとして、以下の二種類の構文がある。

2.1.1 アクション・プレフィクス

Ｂが動作式で、ａがイベントのとき、

a ; B

と書くことは、イベント a の発生後に、動作式Ｂで表現される動作を行なうことを意味する。

2.1.2 順次合成（Enabling Operator）

二つのプロセスが動作式Ｂ１とＢ２によって与えられるとする。イネーブリングオペレーター

$B1 \gg B2$

は、Ｂ１を終了してからＢ２を実行することを意味する。

2.2 選択（Choice）

Ｂ１とＢ２が既存の動作式の時、チョイスオペレーター [ ] を用いた動作式

B1 [ ] B2

は、プロセスがＢ１とＢ２のいずれかを選択し、動作することを意味する。どちらが選択されるかは、環境との交信によって決まる場合もあるし、非決定的な場合もある。

また、動作式Ｂの前に、[...] → Ｂと書き、[ ] のなかに条件式を書く。すると、条件式が満たされない限り、Ｂが実行されることはない。これを利用して条件つき選択を表現することができる。

$([x > 0] \rightarrow \text{Ev1};\ldots)\ [\ ]\ ([x \leq 0] \rightarrow \text{Ev2};\ldots)$

2.3 くりかえし

プロセス名ｐと実ゲート引数 $g1,\ldots,gn$ を用い、

$p[g1,\ldots,gn]$

と書くことを、プロセス・インスタンシエーションと呼び、他で定義されたプロセスを呼び出すことを意味する。自分自身を呼び出すことも可能である。この再帰呼び出しによって、くりかえしを記述することができる。

2.3.1 並列合成

二つのプロセスが動作式B 1とB 2によって与えられるとする。プロセスの並列合成

$$B1\ |[g1,\ldots,gn]|\ B2$$

は、二つのプロセスB 1、B 2が特定のゲート $g1\ \ldots\ gn$ に関するイベントを同期して（同時に）行なうことを意味する。

並列合成のオペレーターは他にも二種類ある。
B1 ||| B2 は、[ ] 内のゲートが無い場合の略記であり、二つのプロセスが完全に独立して動くことを意味する。
B1 || B2 は、[ ] 内にB 1とB 2のすべてのゲートが入っている場合の略記であり、二つのプロセスが完全に同期して動くことを意味する。

2.4 データの受渡し

一つのプロセスのデータを他のプロセスに渡す方法は、主に二つある。

2.4.1 通信

前出の並列合成を使う方法である。「二つのプロセスが一つのゲートに関して同期をとり、同時に同じイベントを起こす」ということは、別の観点から見ると、「二つのプロセスが一つのゲートを介して、互いに通信を行なった」と考えることもできる。この通信に際して、データの受渡しをすることができる。

通信を行なうプロセス間のゲートに関して同期をとり、「ゲート名+データ」という構造的アクションを実行する。この時、送信側のデータの頭に！マークをつけ、受信側の変数の頭に？マークをつける。すると、送信側のデータの値が受信側の変数に代入される。

```
(inp ?x:Nat;gate !x;exit) |[gate]| (gate ?y:Nat;outp !y;exit)
```

左側の inp に入力したデータは gate によって右側に渡され、outp から出力される。

2.4.2 accept

前出の順次合成を使う方法である。プロセスが終了する際、次に実行されるプロセスにデータを渡すことができる。

プロセスを終了するイベント exit に、伝達したいデータを引数として持たせる。

```
process Proc1[...]:exit:=
   ...; exit(data1,data2)
endproc
```

データを受けとる側は、accept を用いて変数にデータを入力する。

```
Proc1[...] ≫ accept x:type1,y:type2 in Proc2[...](x,y)
```

2.5 イベント隠蔽

Bが動作式の時、以下の hiding オペレーター

hide $g1,\ldots,gn$ in B

は、Bの起こすイベントの中で、ゲート $g1,\ldots,gn$ を巻き込むイベントを内部アクションとする。内部アクションは外部から観測不能であり、環境の介在なしに自発的に起こる。

3 各種の記述スタイル

3.1 分類

本研究では、四つの記述スタイル (1) オブジェクト指向型、(2) 機能指向型、(3) イベント指向型、(4) 状態指向型を提案する。

これらの記述スタイルを、「記述対象のどの面に注目し、それをどう表現するか」によって分類する。ここでは、対象システムをとらえる際によく用いられている三つの観点 (1) 構造的観点、(2) 機能的観点、(3) 動作的観点にしたがって記述スタイルを分類する。

構造的観点は、対象の持つ構造的な側面に注目する。対象を実体を持った構造体の集合として考え、個々の構造を記述する。この観点に分類されるものはオブジェクト指向型である。

機能的観点は、対象の持つ機能的な側面に注目する。対象が、入力された情報をいかに処理して出力するかを記述する。この観点に分類されるものは機能指向型である。

動作的観点は、対象の持つ動作的な側面に注目する。対象が、その時その時にどのような動作を行なうかを記述する。この観点に分類されるものは、動作指向型であり、さらにイベント指向型、状態指向型に分類される。

3.2 オブジェクト指向型

オブジェクト指向型は、対象システムを通信しあうオブジェクトの集まりとしてとらえる手法である。まず、与えられた仕様の中からシステムの物理的構成要素であるオブジェクトを抽出し、各オブジェクトの構成を記述するスタイルである。

- 抽出したオブジェクトは、ＬＯＴＯＳのプロセスに対応させる。
- オブジェクトとオブジェクトを連結する通信網は、ＬＯＴＯＳのゲートに対応させる。
- オブジェクト間の通信網を流れるデータやメッセージは、ＬＯＴＯＳのデータに対応させる。
- オブジェクトの内部状態や、保存する必要のあるデータは、ＬＯＴＯＳのプロセスに、パラメータという形で記憶させる。

以下に、典型的なオブジェクトの相関図と、それをオブジェクト指向型によって記述したものを例としてあげる。

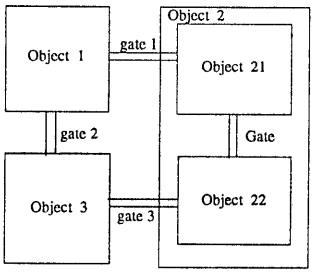

図２：オブジェクト相関図

```
behaviour
  (Obj1[gate1,gate2](初期データ)
  |[gate1]| Obj2[gate1,gate3])
  |[gate2,gate3]| Obj3[gate2,gate3]
where
  process Obj1[gate1,gate2](data: datatype): noexit :=
    (メッセージ１受信)；(メソッド１送信)；
    Obj1[gate1,gate2](newdata)
  [] (メッセージ２受信)；(メソッド２送信)；
    Obj1[gate1,gate2](newdata)
  []  ...
  endproc
  process Obj2[gate1,gate3]: noexit :=
    hide Gate in
    Obj21[gate1,Gate] |[Gate]| Obj22[Gate,gate3]
  where
    process Obj21[gate1,Gate]: noexit :=
    ...
    endproc
  ...
  endproc
endspec
```

図中の Obj2 のように、一つのプロセスを同期した複数のプロセスに分割して考える場合、内部の同期に使われているゲート（図中の Gate）は、hiding オペレーターを使って外部から隠蔽する。この手法は、次の機能指向型にも使用される。

個々のオブジェクトは独立しているので、他のオブジェクトを直接、プロセス・インスタンシエーションで呼び出すことはない。個々のオブジェクトは、

(他のオブジェクトからのメッセージを受信) ⟶
(他のオブジェクトへメソッドを送信) ⟶
(再帰呼び出しによるくりかえし)

という動作パターンが、そのオブジェクトで受信可能なメッセージパターン分だけ、[ ] で連結された記述となる。

3.3 機能指向型

機能指向型は、システムをデータを変換する機能の集まりとしてとらえ、機能とその間のデータの流れによって記述するスタイルである。

データフロー図との対応では以下のようになる。

- データフローを意味する記号⟶は、ＬＯＴＯＳのゲートに対応させる。
- データを処理する「プロセス（バブル）」は、ＬＯＴＯＳのプロセスに対応させる。
- データを一時的に保存する「ファイル」は、ＬＯＴＯＳのパラメータ付プロセスに対応させる。データはプロセス内に、パラメータという形で保存される。
- 外部の構造物を意味する「エンティティ」に対応するものは、ＬＯＴＯＳの冒頭の specification 部に定義される環境へのゲートである。

プロセスはデータをうけとり、処理して出力する。長期的なデータ保存をおこなうことはないので、パラメータを持つことはない。出力データは抽象データ型のオペレーションによって求められる。プロセスは常に活動状態にあるので、処理が終わった後、自己再帰をおこなう。

ＤＦＤの場合、図３のようにデータを処理するプロセスをさらに詳細化して、複数のプロセスの集合体とみなす階層的な記述が行なわれる。この場合は、サブプロセスの連結状態をメインプロセスの行動部に書き、後にサブプロセスの記述を行なう。その例を以下に示す。

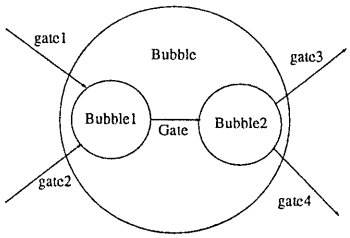

図３：ＤＦＤ

```
process Bubble[gate1,gate2,gate3,gate4]: noexit :=
  hide Gate in
  Bubble1[gate3,gate4,Gate]
  |[Gate]| Bubble2[gate1,gate2,Gate]
where
  process Bubble1[gate3,gate4,Gate]: noexit :=
    Gate ?dataA:datatype; (* データ受信 *)
    gate3 !out1(dataA); gate4 !out2(dataA); (* データ送信 *)
    Buble1[gate3,gate4,Gate]
  endproc
  process Bubble2[gate1,gate2,Gate]: noexit :=
  ...
  endproc
endproc
```

ＤＦＤのファイルに対応するプロセスは、保存するデータをパラメータという形で記憶している。プロセスは、データを読み出すためのゲート（read）と、外部からデータを書き込むためのゲート（write）を持つ。

ファイルは、どこのプロセスからもアクセスできるように、behaviour 部に書く。

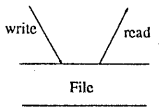

図４：ＤＦＤのファイル

```
behaviour
 (Proc1[gate1,...]...)
 |[read,write]| File[read,write](初期データ)
where
 ...
 process File[read,write](data:type1):noexit:=
  read !data; File[read,write](data) (* 読み出し *)
 []
  write ?x:type1; File[read,write](x) (* 書き込み *)
 endproc
```

3.4 動作指向型

システムが環境に対してどのような動作を行なうかに注目して記述するスタイルが動作指向型記述法である。この手法にはシステムを、起こり得るイベント系列としてとらえる手法と、状態遷移機械としてとらえる手法がある。

3.4.1 イベント指向型

イベント指向型記述法は、仕様の中からイベントを抽出し、その時系列的順序を記述するスタイルである。以下に、イベントのＬＯＴＯＳへの対応を示す。

- これ以上細分化できないイベントは、ＬＯＴＯＳのイベントに対応させる。
- 小さなイベントをまとめて、一つの大きなイベントにしたものは、ＬＯＴＯＳのプロセスに対応させる。

イベント指向型による記述を行なう際には、下のようなアクションツリー図やイベント相関図を使うと理解しやすい。

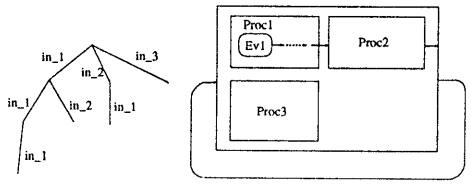

図５：アクションツリー図（左）とイベント相関図

アクションツリー図は、プロセス内でのイベントの流れを表現する。枝は一つのイベントに対応し、接点はイベントの接続に対応する。[ ] で複数のイベントが結ばれている場合、接点は複数の枝への分岐点となる。

イベント相関図は、プロセス間の実行の流れを表現している。四角はプロセス、丸はイベントである。矢印はプロセス（またはイベント）の流れで、次のプロセス（またはイベント）を指している。

これ以上細分化できないイベントは、ＬＯＴＯＳのイベントに対応させる。例えば、「１を入力する」というイベントは、in_1 というようにする。

さらに、記述を読みやすくするために、小さなイベントをまとめた大きなイベントをＬＯＴＯＳのプロセスに対応させる。図５の「合計が３になるまで入力する」という大きなイベントは、以下のようにプロセス化できる。

```
process Sum3[in_1,in_2,in_3]: exit :=
  in_1; (in_1; in_1; exit
         [] in_2; exit)
[] in_2; in_1; exit
[] in_3; exit
endproc
```

実際に記述をおこなう場合、まず仕様の< behaviour-expression >部に、全体のイベントの流れを記述するメインプロセスを記述する。

```
behaviour
  MainProc[Ev1,Ev2,...]
```

上記のようにまとめたプロセスを、メインプロセスのサブプロセスとして記述する。イベント指向型のＬＯＴＯＳ構造は以下のようになる。

```
behaviour
  MainProc[Ev1,Ev2,...]
where
  process MainProc[Ev1,Ev2,...]: noexit :=
    Proc1[Ev1,...] ≫ Proc2[Ev2,...] ≫
    ...≫ MainProc[Ev1,Ev2,...]
  [] Proc3[Ev3,...] ≫ ...
  ...
  where
    process Proc1[Ev1,...]: exit :=
      Ev1; ...; exit
    endproc
    ...
  endproc
endspec
```

個々のプロセスは、イベントの順次接続である ；、プロセスの順次合成である ≫、選択である [ ] によって記述される。

3.4.2 状態指向型

状態指向型は、システム全体を一つの状態遷移機械とみなし、その状態遷移を記述してゆくスタイルである。

以下に、状態遷移図のＬＯＴＯＳへの対応を示す。

- 各々の状態は、ＬＯＴＯＳのプロセスのパラメータに対応させる。
- 状態変化を起こすイベントは、ＬＯＴＯＳの「ゲート＋変数」を使って表現する。
- 保存の必要があるデータは、ＬＯＴＯＳのプロセスのパラメータに記憶させる。

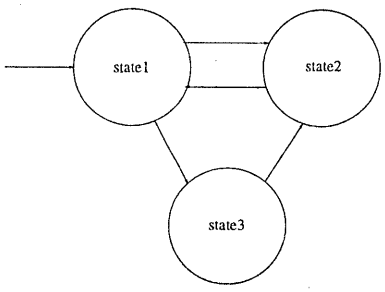

図６：状態遷移図の例

仕様の< behaviour-expression >部には、状態遷移機械を表すプロセスおよび状態パラメータの初期値を記述する。

```
behaviour
  StateMachine[...](state1)
```

状態遷移機械プロセスの内部には、状態パラメータの値に応じて、各々の状態が行なう動作の記述をする。

```
process StateMachine[...](s:state):noexit:=
  [s = state1]→ （状態遷移を起こすイベント）;
    ([遷移条件１] → ...;StateMachine[...](state2))
  [] ([遷移条件２] → ...;StateMachine[...](state3))
[] [s = state2] → ...; StateMachine[...](state1)
[] [s = state3] → ...; StateMachine[...](state2)
endproc
```

状態遷移機械は、イベントが生じることによって内部状態を変化させる。

状態間を越えて保存したいデータは、プロセスにそのデータ用のパラメータを設定し、記憶させる。

状態遷移機械には、最終的に停止するものと無限に作動するものの二種類がある。停止する可能性のあるものは、プロセスを exit で定義し、無限に作動するものは noexit で定義する。

複数の状態遷移機械を組み合わせて一つの状態遷移機械にする方法もある。

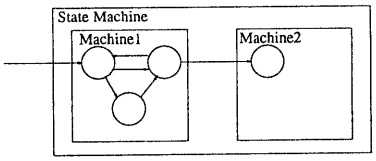

図７：状態遷移機械の結合

```
behaviour
 StateMachine[...](1)
where
 process StateMachine[...](n:MachineNumber):noexit:=
   [n = 1] → Machine1[...](state1);
             accept m:MachineNumber in StateMachine[...](m)
 [] [n = 2] → Machine2[...](state1);
             accept m:MachineNumber in StateMachine[...](m)
   ...
 where
   process Machine1[...](s:state):exit:=
   ...;exit(2)  (* 次の機械の番号 *)
   endproc
 ...
```

メインとなる状態遷移機械が持つパラメータは内部の状態遷移機械を識別するためのものである。内部の個々の状態遷移機械は、有限時間内に終了するので exit で定義する。

3.5 まとめ

以上で紹介した４つの記述スタイルは、上位レベルが並列オペレータを用いて構成される並列構成のオブジェクト指向型と機能指向型、逐次実行や選択などのオペレータを用いて構成される直列構成のイベント指向型と状態指向型の二つにわけられる。

並列構成のオブジェクト指向型と機能指向型は、システムをいくつかの部分にわけて、その並列的な合成で全体を記述しようというものである。それぞれの部分は、さらに小さな部分の合成であると考え、これを繰り返して対象を詳細に記述する。対象を分割して考えることにより、対象の複雑性は減少し、個々のプロセスはモジュール化しやすい構造を持つ。

オブジェクト指向型と機能指向型の違いは、対象のどの側面から分割を行なうか、という点にある。オブジェクト指向型の場合、システムを構成するのはシステムを物理的に構成しているオブジェクトであるが、機能指向型の場合、それは機能の一部を担うプロセスである。

直列構成のイベント指向型と状態指向型は、対象が行なう動作を時系列に従って記述しようというものである。

イベント指向型は、単純な一本道の動作をするシステムを記述する際には有用であるが、複雑な条件分岐をともなったシステムを記述するには適さない。また、サブプロセス内部で生じるイベントは、すべて親プロセスから継承したものである。従って、親プロセスには、そのサブプロセス内部で生じるすべてのイベントを記述しておかねばならない。一つのプロセスが扱うイベントの種類が数十のオーダーにもなると、繁雑になって読みにくくなる。

状態指向型は、イベント指向型が苦手とする複雑な条件分岐を記述するのに適している。そのかわり、状態遷移図を使うので、イベント指向型にくらべると全体の動きが見通しにくい。

4 記述スタイルの階層的な関係

複雑な大規模システムの仕様を記述する際には、システムを部分システムに階層的に分割して記述を行なっていく。各階層レベルでの記述は同じ記述スタイルで行なわれる場合もあるが、異なる記述スタイルがとられることもある。本節では、どの階層的な組合せが可能かを考察する。

まず、並列構成に属する二つの記述スタイルは、直列構成の二つのスタイルよりも上位に位置する。

オブジェクト指向型は対象からオブジェクトを抽出し、機能指向型は対象を構成する機能を抽出する。この場合、「対象をオブジェクトに分割した後、個々のオブジェクトから機能を抽出する」ことが自然に行なえるのに対して、「対象を機能的に分割した後、個々の機能からオブジェクトを抽出する」ことは考えにくい。よって、オブジェクト指向型は機能指向型より上位に位置すると考える。

直列構成のイベント指向型と状態指向型の間には、明確な上下関係はない。イベント指向型は一本道の動作を記述するのに適し、状態指向型は複雑な分岐を記述するのに適しているので、場合によって使い分けるのが良い。

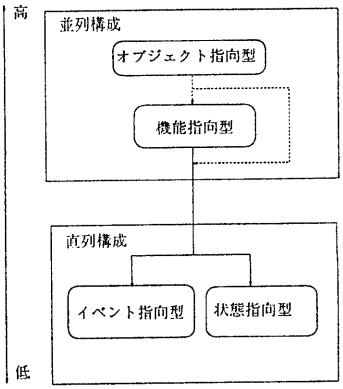

図８：記述スタイルの階層的関係

5 評価と考察

各記述スタイルに基づいて、「酒屋の在庫管理問題」[3]を記述した。その記述のメインとなる部分を以下に載せる。

5.1 オブジェクト指向型

抽出したオブジェクトは４つ。

- 受付係⟶Receptionist
- 積荷票⟶StockLists
- 在庫不足リスト⟶LaclLists
- 注文番号⟶Numbers

オブジェクト相関図は以下の通り。

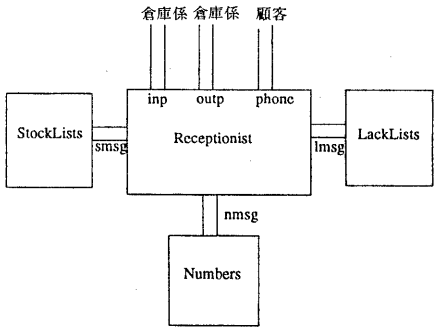

図９：在庫管理問題のオブジェクト相関図

```
behaviour
   MainProcess[inp,phone,outp]
where
   process MainProcess[inp,phone,outp]:noexit :=
      hide lmsg1,lmsg2,smsg1,smsg2,nmsg1,nmsg2 in
      ((Receptionist[inp,phone,outp,smsg1,smsg2,
                     lmsg1,lmsg2,nmsg1,nmsg2]
      |[smsg1,smsg2]| StockLists[smsg](L0))
      |[lmsg1,lmsg2]| LackLists[lmsg](K0))
      |[nmsg1,nmsg2]| Numbers[nmsg](0)
   where
   ...
```

5.2 機能指向型

作成したＤＦＤを以下に載せる。

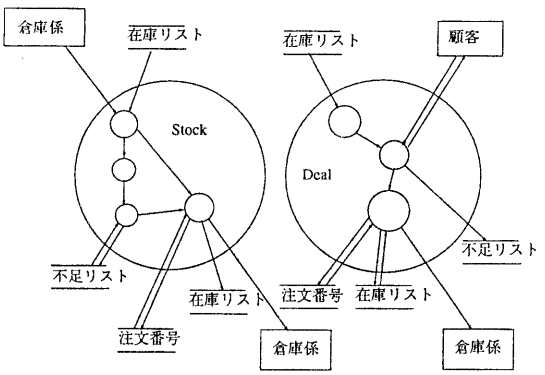

図１０：在庫管理問題のＤＦＤ

```
behaviour
  Receipt[inp,outp,receive,send]
where
  process Receipt[inp,outp,receive,send]:noexit :=
    hide stockread,stockwrite,
    lackread,lackwrite,numread,numwrite in
    (((Stock[inp,outp,stockread,stockwrite,
              lackread,lackwrite,numread,numwrite] |||
    Deal[outp,receive,send,stockread,stockwrite,
          lackread,lackwrite,numread,numwrite])
    |[stockread,stockwrite]| StockFile[stockread,stockwrite](L0))
    |[lackread,lackwrite]| LackFile[lackread,lackwrite](K0))
    |[numread,numwrite]| NumFile[numread,numwrite](0)
where
...
```

5.3 イベント指向型

作成したイベント相関図を以下に載せる。

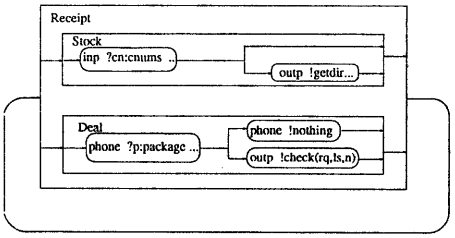

図１１：在庫管理問題のイベント相関図

```
behaviour
  Receipt[inp,phone,outp](L0,K0,s(0))
where
  process Receipt[inp,phone,outp]
                 (ls:loadlist,lack:lacklist,n:Num):noexit :=
  (Stock[inp,outp](ls,lack,n) ≫
   accept nld:loadlist,nla:lacklist,ne:Num in
   Receipt[inp,phone,outp](nld,nla,ne) )
  []
  (Deal[phone,outp](lack,packs,n) ≫
   accept nld:loadlist,nla:lacklist,ne:Num in
   Receipt[inp,phone,outp](nld,nla,ne) )
  where
  ...
```

5.4 状態指向型

作成した状態遷移図を以下に載せる。

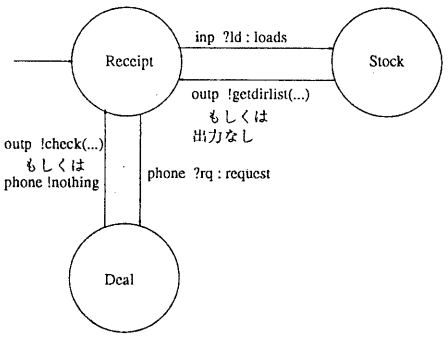

図１２：在庫管理問題の状態遷移図

```
behaviour
 Machine[inp,outp,phone](Receipt,L0,K0,0)
where
 process Machine[inp,outp,phone]
   (s:state,ls:loadlist,lack:lacklist,n:Num):noexit :=
  [s = Receipt]→
    (inp ?ld:loads;Machine[inp,outp,phone](Stock,add(ld,ls),lack,n))
    []
    (phone ?rq:request;Machine[inp,outp,phone](Deal,rq,ls,lack,n))
  [s = Stock]→
    ([pick(lack,count(ls)) ≠ K0] → outp !getdirlist(lack,ls,n);
     Machine[inp,outp,phone](Receipt,getload(lack,ls),lackrest(lack,ls),...))
  [] ([pick(lack,count(ls)) = K0]→
     Machine[inp,outp,phone](Receipt,ls,lack,n))
  [s = Deal]→
    ([pick(pl(rq,K0),count(ls)) ≠ K0] → outp !check(rq,ls,n);
     Machine[inp,outp,phone](Receipt,loadrest(rq,ls),lack,...))
  [] ([pick(pl(rq,K0),count(ls)) = K0] → phone !nothing;
     Machine[inp,outp,phone](Receipt,ls,add(rq,lack),n)
endproc
```

5.5 記述結果の考察

評価のポイントとしては、以下のようなものが考えられる。

1. 書きやすさ
   - 記述のサイズ、プロセス数
   - 論理的なバグ数
2. 変更のしやすさ

以下に、各スタイルの記述サイズ（メインとなる部分の文字数）、プロセス数、1プロセス当たりの平均のゲート数の表をのせる。

スタイル名	記述サイズ	プロセス数	平均ゲート数
オブジェクト指向型	2349byte	4	3.75
機能指向型	3571byte	11	4.45
イベント指向型	1231byte	3	2.33
状態指向型	977byte	1	3

表1：記述スタイルとサイズ

「酒屋の在庫管理問題」は、扱う対象の規模が小さな問題に属する。扱う動作式は短く、複雑な分岐も存在しないので、直列構成のイベント指向型や状態指向型を用いても、全体を記述するのはさほど困難ではない。

逆に、並列構成のオブジェクト指向型と機能指向型は、問題の簡略化に寄与しない無駄な分割を行なっているので、プロセス同士の同期関係を記述しなければならない分、直列構成の二つにくらべて記述のサイズは大きくなってしまった。

次に、変更のしやすさについて考える。ここでは、これまでの売上の総計を報告する機能をシステムに追加させることを考える。

- オブジェクト指向型
  「売上の総計」は長期保存の必要のある情報である。売上票というオブジェクトを追加してそこに記憶させる。また、受付係のオブジェクトの内部に、売上をたずねられた時の対応動作をチョイスオペレーターを使って連結し、出庫時に売上金を売上票に記録する動作を加える。
- 機能指向型
  「売上の総計」ファイルを追加し、出庫時に売上金を記録させる。Stock,Dealとオペレータで結合し、売上金を報告するプロセスを設定し、記述する。
- イベント指向型
  「売上の総計」を記憶するパラメータをメインプロセスに追加し、売上の報告をするイベント列をチョイスオペレーターを使って連結する。出庫時に売上金を記録するイベントを Stock,Deal の内部に追加する。
- 状態指向型
  「売上の総計」を記憶するパラメータをメインプロセスに追加する。出庫時に売上金を記録させる。売上報告という状態を一つ用意し、Receipt の状態で売上をたずねられたら、売上報告状態に遷移して報告を行ない、Receipt にもどる。

プロセス数の増加やプロセス間の通信イベントの発生を考慮すると、システムの大きさがこの程度の場合は、動作指向のイベント指向型、状態指向型を使って全体を記述する方が変更の点でも手軽である。

6 あとがき

本研究では、形式的仕様記述言語ＬＯＴＯＳを用いて、4つの記述スタイルを紹介し、記述スタイル間の階層的な関係を検討した。
今後の研究課題としては、以下のようなものが考えられる。

- ＤＦＤに制御用の信号を組み合わせ、それをＬＯＴＯＳで表現する。
- 記述サイズ以外の、記述スタイルを比較するための客観的な評価基準を考える。
- 紹介した4つ以外の記述スタイルを考案する。
- 記述スタイルを効果的に組み合わせた記述法を確立する。

謝辞

本研究を進める上で、多大な協力をいただいた本学の川瀬智氏に深く感謝いたします。

参考文献

[1] C.A.Vissers,G.Scollo,and M.v.Sinderen,'Architecture and Specification Style in Formal Descriptions of Distributed Systems',Proc.IFIP WG6.1 Symposium on Protocol Specification,Testing and Verification VIII ,Atlantic City,USA,June1988,pp189-204

[2] Information processing systems - Open sysytems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour. ISO 8807.

[3] 二村良彦, 雨宮真人, 山崎利治, 淵一博. 新しいプログラミング・パラダイムによる共通問題の設計. 情報処理, Vol.26, No.5, pp458-459

[4] 大蒔和仁, 二木厚吉. 図書館の問題とエレベータの問題のＬＯＴＯＳによる仕様記述. 情報処理学会ソフトウェア工学研究会資料, 64-12(1989)

[5] 高橋薫, 神長裕明, 白鳥則郎. ＬＯＴＯＳ言語の特質と処理系の現状と動向. 情報処理, Vol.31, No.1, pp35-46

[6] 大蒔和仁, 二木厚吉. 形式仕様記述言語 ＬＯＴＯＳ の試用経験. 情報処理, Vol.31, No.10, pp1400-1413.

[7] 来間啓伸, 大槻繁. 集合に基づく形式的言語を使ったソフトウェア仕様の記述形態について. 情報処理学会研究報告, Vol.92, No.10, pp97-104.

[8] 二木厚吉. ＩＳＯにおける形式記述技法の標準化動向. 情報処理, Vol.31, No.1, pp3-10.