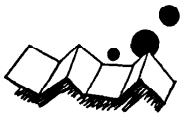
解　説



論理型並列プログラミング言語 Concurrent Prolog∏による在庫管理システムの記述†

大木　優††　二村良彦†††　竹内彰一††
宮崎敏彦††　古川康一††

1. はじめに

Concurrent Prolog(以降, CP と呼ぶ) は論理型並列プログラミング言語の1つであり，イスラエルの E. Shapiro により設計された．論理型プログラミング言語を並列に実行できるという特徴から第五世代コンピュータ・プロジェクトの中期における核言語の基本となっている．CP は論理型プログラミング言語の中で並列実行を基本とする論理型プログラミング言語であり，共有論理変数により通信を実現する簡潔な言語である．また，同期などのプロセス制御に関して従来のプログラミング言語と比較して十分強力な表現力を持っている[1]．

CP のプログラムでは，各ゴールは論理変数を介して通信しながら並列に実行する．各ゴールはゴールごとに並列に実行されるので，ゴール間で同期を取る場合には読み出し専用標記 (Read only annotation) を使って同期を取る必要がある．CP は共有論理変数をチャネルとし，プロセスでオブジェクトを実現することによって，オブジェクト指向プログラミングが実現できる．オブジェクト指向プログラミングは，CP を使ってプログラムを作成する際の有効なプログラミング・スタイルの1つである[2]．

本論文では，CP のオブジェクト指向プログラミングを使って在庫管理システムの共通問題[3]のプログラムを記述する．さらに，問題をモデル化するために，従来の機能分割に基づく方法 (以降，部分機能分割法と呼ぶ) とオブジェクト指向の方法[4),5] (以降，構成要素分割法と呼ぶ) の2つの方法を使用する．この2つの方法は次に示す方法である．

(1)　構成要素分割法

現実にあるものを単位としてモデル化し，その後，分割したものについて機能を考える．倉庫をモデル化する場合，倉庫の中の実際のものをモジュールの単位とする．例えば，倉庫の中にあるコンテナ自体を1つのモジュールとして考える．

(2)　部分機能分割法

システム全体の機能を考えて，機能を分割しながら，機能単位にモデル化する．(1)と同様に，倉庫をモデル化する場合，まとまった機能を実現する処理をモジュールの単位とする．例えば，倉庫の中のコンテナを管理する処理を1つのモジュールとして考える．

2つのモデル化の方法に従ってプログラムを作成した結果，構成要素分割法によるプログラムのステップ数は部分機能分割法によるプログラムに比べて約3分の2であった．

本論文では，まず，2つのモデル化の方法を使ってモデルを作る．次に，CP によるオブジェクト指向プログラミングについて説明し，作成したモデルのプログラムを記述する．最後に，2つのモデル化の方法を比較する．なお，本論文で述べる CP のプログラムは，著者の一人が CP 風に日本語で記述したものを基にしたものである[6]．

2. 在庫管理システムのモデル化

共通問題の仕様は文献[3]で述べられているので，ここでは問題の仕様については述べない．

受付係は倉庫の在庫を管理するため，倉庫の状況を知る必要がある．そこで，受付係のモデルを図-1のように，外部からメッセージを受け取る部分と倉庫をシミュレーションする部分 (以降，この部分を単に倉庫と呼ぶ) とに分けて考える．

† Description of Stock Management System by Concurrent Prolog by Masaru OHKI (Institute for New Generation Computer Technology), Yoshihiko FUTAMURA (Central Research Laboratory, Hitachi, Ltd.), Akikazu TAKEUCHI, Toshihiko MIYAZAKI and Koichi FURUKAWA (Institute for New Generation Computer Technology).

†† (財)新世代コンピュータ技術開発機構
††† (株)日立製作所中央研究所第8部

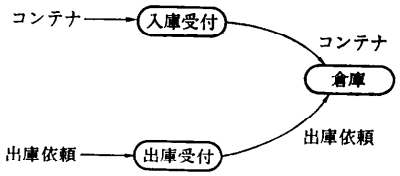


図-1　受付係のモデル

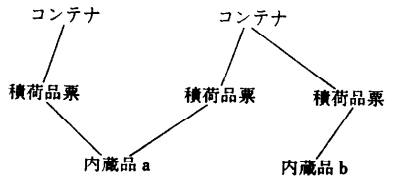


図-2　コンテナと積荷品票，内蔵品の関係

（1）　入庫受付係

①　コンテナを受け取り，コンテナを倉庫に渡す．

（2）　出庫受付係

①　出庫依頼を受け，倉庫に出庫指示を出す．

（3）　倉　　庫

①　コンテナとその積荷品を保管する．

②　出庫指示書を受け，内蔵品を出庫する．

③　在庫がない場合，在庫不足として扱う．

ここで，コンテナは受付係の入力である積荷票に対応する情報である（以降，受付係の入力である積荷票のことをコンテナと呼ぶ）．

プログラムを作成するためには，図-1 の倉庫をさらにモデル化する必要がある．倉庫をモデル化する前に，倉庫の中に実際にあるものに対応するデータについて考える．これらは，次の３つである．

（1）　内　蔵　品

倉庫に保管されている品目ごとの商品である．

（2）　コ ン テ ナ

積荷品を混載しているコンテナである．

（3）　積 荷 品 票

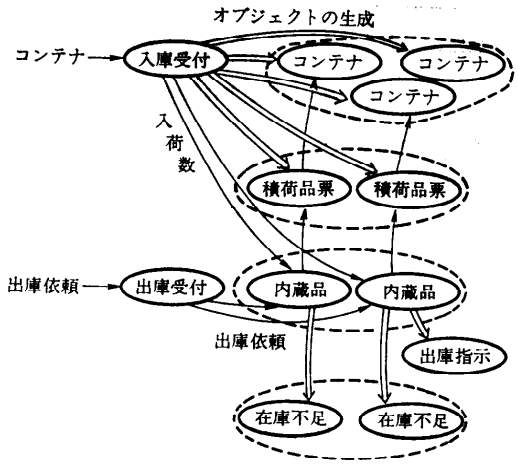
積荷票の１要素で，積荷品とコンテナとの対応を記録している．

これらのデータは図-2 に示す関係を持っている．

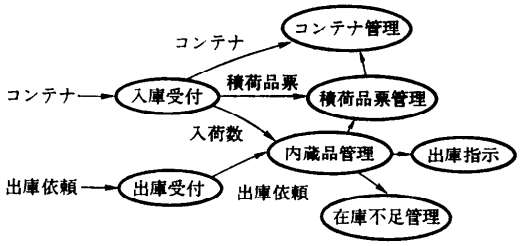
倉庫をモデル化するために，先に述べた２つの方法を適用する．これらの方法の適用を次に示す．

（1）　構成要素分割法

倉庫の中に存在するものを単位としてモデルを考える．そうすると倉庫は，コンテナや積荷品票，内蔵品から構成されていると考えることができる．次に，それらの構成要素に機能を割り当てる．割り当てる機能は，構成要素自身の処理である．



（a）　構成要素分割法によるモデル



（b）　部分機能分割法によるモデル

図-3　構成要素分割法と部分機能分割法によるモデル

（2）　部分機能分割法

倉庫の機能を機能分割し，分割された機能を処理するものについて考える．そうすると倉庫は，コンテナや積荷品票，内蔵品を管理する管理係から構成されると見なすことができる．各管理係ではそれぞれの情報を管理したり，処理したりする．

これらの方法による受付係のモデルを図-3 の(a)と(b)に示す．構成要素分割法では，入庫受付係がコンテナを受け取ると，入庫したコンテナに従い，コンテナや積荷品票そのものを生成する．そして内蔵品には入荷数を知らせる．また，出庫受付係が出庫依頼を受け取ると内蔵品に出庫依頼を伝え，在庫不足であれば在庫不足そのものを生成する．図-3 の (a) での点線は構成要素のグループを示す．一方，部分機能分割法では，入庫受付係がコンテナを受け取ると，それぞれの管理係に必要なデータを送る．出庫受付係が出庫依頼を受け取ると，内蔵品管理係に出庫依頼を伝える．もし在庫不足であれば，在庫不足管理係に在庫不足を伝える．

3. CP によるオブジェクト指向プログラミング

この章では，CP の概要を述べ，CP によるオブジェクト指向プログラミングについて説明する．

3.1 CP の概要

CP は，共有論理変数によるメッセージ・パッシングを行うストリーム AND 並列型論理プログラミング言語である．CP の特徴をまとめると次の通りである．

（1） AND 並　列

複数ゴール（プロセス）を並列に解く．

（2） OR 並　列

ゴールの候補節を並列に解く．

（3） 共有論理変数

プロセス間通信チャネルに使う．

（4） ガード付き節

ガード“|”を持つ節である．もしガードが省略されたならば右辺の1番左にあると見なす．ガードを越えることができる節は，ガードが一番早く解けた候補節だけである．

（5） 読み出し専用標記（以降，Read Only Annotation と呼ぶ）

プロセス間の同期を行う．

CP の節の例を図-4 に示す．これは，エラトステネスのふるいのアルゴリズムに基づいた素数生成プログラムの一部である．generate 述語で整数の列を論理変数 I に生成する．sift 述語ではこの整数列を使って素数を計算する．“,”で結ばれた generate 述語と sift 述語は並列に処理される．論理変数 I は generate 述語と sift 述語の共有論理変数であり，generate 述語と sift 述語間のチャネルとなっている．この論理変数 I を通して整数の列がストリームとして流れる．sift 述語の論理変数 I に Read Only Annotation が付いている理由は，generate 述語で整数の生成が遅れた場合，共有論理変数 I が instantiate していないのに sift 述語が先行して実行することを防ぐためである．

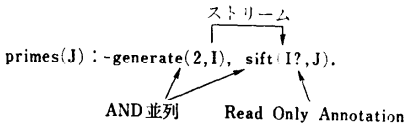


図-4 CP のプログラム例（その1）

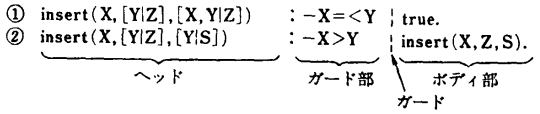


図-5 CP のプログラム例（その2）

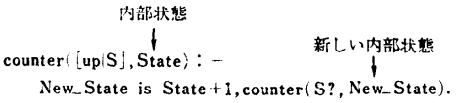


図-6 カウンタの定義

次に，ガード付き節の例を図-5 に示す．これは，順序付けられたリストに順序を守りながらある値を挿入するプログラムである．ある値 X がリストの先頭の値 Y に等しいか小さければ，X を Y の前に挿入し（①），そうでなければ，Y の次の要素と X とを比べる（②）．ヘッドを含めてガード部は，どのボディ部を実行するかを選ぶ条件に相当する．

3.2 CP によるオブジェクト指向プログラミング

まず，オブジェクトを次のように定義する[5]．

オブジェクト＝実際の問題領域に登場するものが持つ機能や知識をモデル化した概念実体．

オブジェクトは，メッセージを通してのみ他のオブジェクトへ仕事を依頼することができる．そのため，オブジェクトには，隠蔽性が高い，オブジェクトごとに並列に実行できる，と言った利点がある．このようなオブジェクトを CP では次のようにして実現する．

オブジェクト　　＝プロセス

オブジェクトの内部状態＝プロセスの引数

CP によるオブジェクトの記述例を図-6 に示す．これは，オブジェクト“カウンタ”の定義の一部である“up”というメッセージを受信すると，オブジェクトは内部状態に1を加えて，その結果を新しい内部状態とする．

CP のプロセスによって実現されたオブジェクトの実行上の特徴の一つは，メッセージの処理を並列に実行できることである．これは，CP が AND 並列にプ

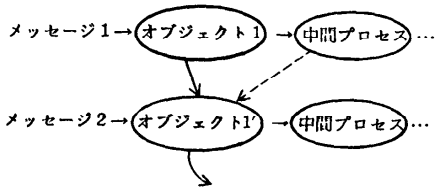


図-7 CP によるオブジェクトの実行

ロセスを実行できるためである．図-7 を使って，これを説明する．オブジェクトがメッセージを受け取ったならば，そのオブジェクトはメッセージを処理する中間プロセスと次のメッセージを受け取る自分自身のプロセスの２つのプロセスに置きかわる．２つのプロセスは並列に実行できるため，中間プロセスはメッセージの処理を続け，次のメッセージを受け取ったプロセス（オブジェクト自身）は次のメッセージを入力し，そのメッセージの処理を行う．このように，CP によるオブジェクトは複数のメッセージの処理を並列に実行することができる．

4. 在庫管理システムのプログラム

ここでは，構成要素分割法によるモデルのプログラムを中心に述べる．まず最初に，部分機能分割法のオブジェクトの構成について述べ，次に構成要素分割法によるオブジェクトとプログラムの構成について述べる．

4.1 部分機能分割法によるオブジェクトの構成

部分機能分割法によるオブジェクトの構成は図-8となる．ここで示した構成は銘柄 a と b を取り扱うシステムである．コンテナや積荷品票，在庫不足に関する情報はそれぞれの管理用オブジェクトで管理される．内蔵品は銘柄ごとに作られ，内蔵品の数を保持しているオブジェクトである．図中，双方向矢印で示されている線は，不完全メッセージによる双方向なデータの流れを示している．

このモデルの特徴は，情報を管理するための管理係がいて，情報が集中していることである．また，情報へのアクセスには，管理係を経由する必要があることである．ここで管理係は機能を分割した後のモジュールに対応する．部分機能分割法による構成がプログラマなどに受け入れられやすい理由は，実際のオフィスでの業務分担のモデルに近いからであると思われる．（これらのオブジェクトの処理概要は文献７）にある．）

4.2 構成要素分割法によるモデルのプログラム

構成要素分割法によるオブジェクトの構成は図-9となる．図に示したように，コンテナや積荷品票，在庫不足に関する情報そのものがオブジェクトになっている．これらは実行時に生成されたり，消滅されたりする．

このモデルの特徴の１つは，内蔵品から積荷品票へのメッセージ送信方式がブロードキャスト方式であることである．ブロードキャスト方式のメッセージ送信とは，１対１の送信ではなく，複数のオブジェクトに一度にメッセージを送信する方式である．このブロードキャスト方式のメッセージ送信は構成要素分割法のモデル化と大きな関係がある．それは，内蔵品が積荷品票を管理していないので，どの積荷品票を使って積荷品を出すべきかを知らないことによる．すなわち，誰がメッセージを処理するかは送信側で決めるのではなく，受信側で決めるのである．図-9において，内蔵品が出庫依頼を受け取った時，ブロードキャスト方式で積荷品票全員に出庫指示のメッセージを送信する．積荷品票の中で，一番最初にメッセージを受け取った積荷品票が積荷品を出庫する処理を行う．これ以外の

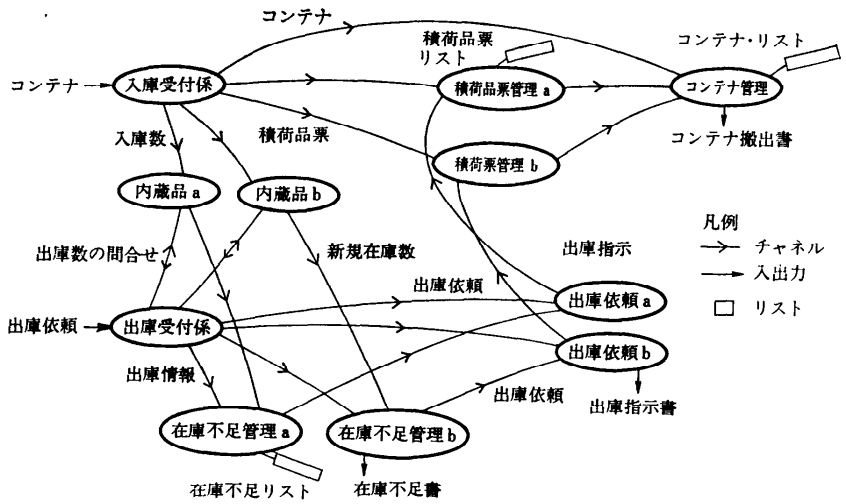


図-8　部分機能分割法によるオブジェクトの構成

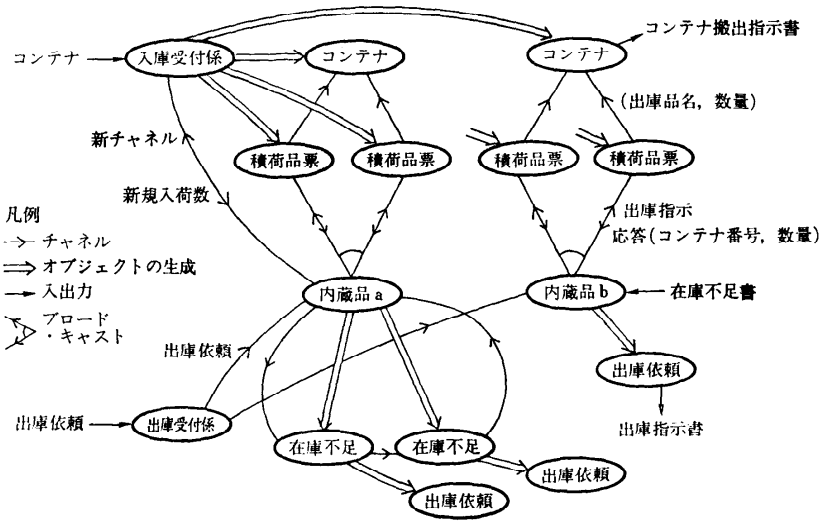


図-9　構成要素分割法によるオブジェクトの構成

積荷品票は，メッセージを読み飛ばす．

一方，内蔵品から在庫不足へのメッセージ送信の方式はブロードキャスト方式ではない．在庫不足情報をオブジェクトとしてチャネルのチェインでつなぎ，チェインに沿ってメッセージを流す方式である．ブロードキャスト方式にできない理由は，内蔵品が個々の在庫不足の不足数の状況を知らないので，新しく入荷した数量によって在庫不足のどれが在庫不足を解消できたかが内蔵品には解らないためである．このように構成要素分割法のモデルでも，情報を管理する必要がある場合はある．しかし，同じ情報の管理でも，でき上がったモデルは部分機能分割法のものとは異なるだろう．在庫不足のモデルでは，図-8と図-9に示すような違いがある．図-9では，在庫不足情報自体がオブジェクトの形になっている．

構成要素分割法によるオブジェクト構成には，部分機能分割法による構成と異なり，情報を管理するものがない．情報自体がシステムの構成要素である．情報へのアクセスは情報そのものに対して行う．ただし，ここでは情報そのものもオブジェクトであるので，アクセスはメッセージを通して行う．この構成のわかりやすさは，倉庫には物が入っているという現実によく対応している点にある．

次に，各々のオブジェクトの処理概要をプログラムと対応つけて以下に示す．

(1)　入庫受付係（図-10（a））

①　コンテナが到着すると，

(i)　コンテナを作る．

(ii)　コンテナに格納している銘柄ごとに，

(a)　内蔵品に入庫数量を知らせる．

(b)　積荷品票を作る．

(2)　出庫受付係（図-10(b)）

①　出庫依頼が来ると，内蔵品に出庫依頼を知らせる．

(3)　内蔵品（図-10(c)）

①　入庫受付係より入庫数を受け取ると，

(i)　在庫数を在庫数＋入庫数とし，在庫不足に在庫数を知らせる．

(ii)　在庫不足が解消されたために，出庫する必要が生じたならば，

(a)　在庫数を在庫数－出庫数とする．

(b)　積荷品票に出庫指示を送る．

(c)　未出庫分があるならば，再度，積荷品票に出庫指示を送る．

②　出庫受付係より出庫依頼を受け取ると，

(i)　出庫依頼数＞在庫数ならば，

(a)　出庫依頼情報を在庫不足として在庫不足のチェインの最後につける．

(b)　在庫不足書を出力する．

(ii)　在庫数≧出庫依頼数ならば，

(a)　在庫数を在庫数－出庫依頼数とする．

(b)　出庫依頼を生成する．

```
nyuko-uketuke ([[num (Cont-num) | Tumini] | C], St-ch) :-
        container (Cont-num, Ch-cont?, Total?),
        create-tumini (C, Cont-num, Tumini, St-ch, Ch-cont, 0, Total).

nyuko-uketuke ([ ], _).

create-tumini (C, Cont-num, [(Item, Tu-cnt) | Tumini], St-ch, Ch-cont,
      Cont-amt, T) :-
      send-stock (Item, stock (Item, Tu-cnt, Tu-ch), St-ch, St-ch-1),
      Cont-amt-1 := Cont-amt+Tu-cnt,
      merge (Ch-cont-0?, Ch-cont-1?, Ch-cont),
      tumini (Item, Tu-ch?, Tu-cnt, Cont-num, Ch-cont-0),
      create-tumini (C, Cont-num, Tumini, St-ch-1?, Ch-cont-1,
            Cont-amt-1?, T).

create-tumini (C, Cont-num, [ ], St-ch, [ ], Total, Total) :-
      nyuko-uketuke (C?, St-ch).
```

コンテナを受け取ると，
コンテナを生成する，
積荷品票の処理を行う．

コンテナがすべて到着すると消滅する．

積荷品票ごとに以下の処理を行う．
内蔵品に入庫数を伝える．
コンテナの格納数を更新する．
積荷品票からコンテナへのチャネルを作る，
積荷品票を生成する，
次の積荷品票の処理を行う．

処理すべき積荷品票がなくなると，
コンテナの到着を待つ．

(a) 入庫受付

```
syukko-uketuke ([(Item, Request,Dest) | U], St-ch) :-
      send-stock (Item, req (Item, Request, Dest), St-ch, St-ch-1),
      syukko-uketuke (U?, St-ch-1?).

syukko-uketuke ([ ], _).
```

出庫依頼を受け取ると，
内蔵品に出庫依頼を伝える，
次の出庫依頼を待つ．

出庫依頼がすべて到着すると消滅する．

(b) 出庫受付

```
stock (Item, [stock (Item, Tu-cnt, Tu-ch) | S1], S2, Tu-ch,
        [entry (Item, Stock-1) | Za-ch], Za-end-ch, Stock) :-
      Stock-1 := Stock+Tu-cnt |
      stock-zaiko (Item, S1, S2, Tu-ch, Za-ch, Za-end-ch?, Stock-1).

stock-zaiko (Item, S1, S2, [req (Item, Used, Ack) | Tu-ch],
          Za-ch, [entry (Item, Rest) | Za-end-ch], Stock-1) :-
      Used := Stock-1-Rest, Used>0 |
      ack-tumini (Ack?, Item, Tu-ch, Tu-ch-1) &
      stock (Item, S1?, S2, Tu-ch-1, Za-ch, Za-end-ch, Rest).

stock-zaiko (Item, S1, S2, Tu-ch, Za-ch, [entry (Item, Rest) |
          Za-end-ch], Stock-1) :-
      Used := Stock-1-Rest, Used=0 |
      stock (Item, S1?, S2, Tu-ch, Za-ch, Za-end-ch, Rest).

stock (Item, S1, [req (Item, Request, Dest) | S2],
        [req (Item, Request, Ack) | Tu-ch], Za-ch, Za-end-ch, Stock) :-
      Request =< Stock, Stock-1 := Stock-Request |
      syukko (Item, Request, Dest),
      ack-tumini (Ack?, Item, Tu-ch, Tu-ch-1) &
      stock (Item, S1, S2?, Tu-ch-1, Za-ch, Za-end-ch, Stock-1).

ack-tumini (ack (Cont-num, Rest), Item, Tu-ch, Tu-ch) :-
      Rest = := 0 | true.

ack-tumini (ack (Cont-num, Rest), Item, [req (Item, Rest, Ack) |
          Tu-ch], Tu-ch-1) :-
      Rest =\= 0 |
      ack-tumini (Ack?, Item, Tu-ch, Tu-ch-1).

stock (Item, S1, [req (Item, Request, Dest) | S2],
          Tu-ch, Za-ch, Za-end-ch, Stock) :-
      Request > Stock |
      zaiko-fusoku (Item, Request, Dest, Za-end-ch?, Za-end-ch-1),
```

入荷数と積荷品票へのチャネルを受け取ると，在庫不足に新在庫数を伝え，
新在庫数を在庫数＋入荷数とし，
在庫不足からの応答を待つ．

在庫不足を解消するために出庫する必要があるならば，積荷品票に出庫数を伝え，
出庫数を在庫数－残りとし，
積荷品票からの応答を待つ，
次のメッセージを待つ．

在庫不足を解消するための出庫する必要がないならば（在庫数＝残り），
次のメッセージを待つ．

出庫依頼を受け取り，在庫数>=出庫依頼数ならば，積荷品票に出庫数を伝え，
新在庫数を在庫数－出庫依頼数とし，
出庫依頼を生成し，
積荷品票からの応答を待ち，
次のメッセージを待つ．

出庫数の残りが0ならば，何もしない．

出庫数の残りが0でないならば，
再度積荷品票に出庫数の残りを伝え，
積荷品票からの応答を待つ．

出庫数を受け取り，
出庫依頼数>在庫数ならば，
在庫不足を生成し，

図-10 プログラム（つづく）

```
    zaiko_fusoku_monitor (Item, Request, Dest),
    stock (Item, S1, S2?, Tu_ch, Za_ch, Za_end_ch_1, Stock).
```
在庫不定書を出力し，
次のメッセージを待つ．

(c) 内蔵品

```
tumini (Item,[req (Item, Count, Ack) | Tu_ch],
        Tu_cnt, Cont_num, Ch_cont) :-
    cp_test_unify (Ack, ack (Cont_num, Rest), Res),
    tumini_result (Res?, Rest, Item, [req (Item, Count, Ack) | Tu_ch],
        Tu_cnt, Cont_num, Ch_cont).
```
内蔵品より出庫指示を受け取ると，

その出庫指示がすでに処理済の印がついているか調べ，その出庫指示の処理を行う．

```
tumini_result (fail, _, Item, [req (Item, Count, Ack) | Tu_ch],
        Tu_cnt, Cont_num, Ch_cont) :-
    tumini (Item, Tu_ch?, Tu_cnt, Cont_num, Ch_cont).
```
受け取った出庫指示が
処理済であれば，
次の出庫指示が到着するのを待つ．

```
tumini_result (succ, Rest, Item, [req (Item, Count, Ack) |
        Tu_ch], Tu_cnt, Cont_num,
        [hansyutu (Item, Tu_cnt) | []]) :-
    Count >= Tu_cnt, Rest := Count-Tu_cnt
    | true.
```
受け取った出庫指示が処理済でなく，出庫数>=積荷品票数　ならば，コンテナに搬出指示を送り，
出庫数の残りを出庫数－積荷品票数 とし，
消滅する．

```
tumini_result (succ, 0, Item, [req (Item, Count, Ack) |
    Tu_ch], Tu_cnt, Cont_num,
        [hansyutu (Item, Count) | Ch_cont]) :-
    Count < Tu_cnt, Tu_cnt_1 := Tu_cnt-Count
    | tumini (Item, Tu_ch?, Tu_cnt_1, Cont_num, Ch_cont).
```
受け取った出庫指示が処理済でなく，出庫数<積荷品票数　ならば，コンテナに搬出指示を送り，
新積荷品票数を積荷品票数－出庫数とし，
再度，出庫指示を待つ．

(d) 積荷品票

```
zaiko_fusoku (Item, Request, Dest, [entry (Item, Stock) | Za_ch],
        [entry (Item, Stock) | Za_end_ch]) :-
    Request > Stock |
    zaiko_fusoku (Item, Request, Dest, Za_ch?, Za_end_ch).
```
在庫数を受け取り，在庫不足数>在庫数ならば，次の在庫不足へ在庫数を伝え，
次の在庫数のメッセージを待つ．

```
zaiko_fusoku (Item, Request, Dest, [entry (Item, Stock) | Za_ch],
        [entry (Item, Rest) | Za_ch]) :-
    Request =< Stock, Rest := stock-Request |
    syukko (Item, Request, Dest).
```
在庫数を受け取り，在庫数>=在庫不足数ならば，次の在庫不足へ在庫数の残りを伝え，
在庫数の残りを，在庫数－在庫不足数とし
出庫依頼を生成し，消滅する．

(e) 在庫不足

```
syukko (Item, Request, Dest) :-
        syukko_monitor (Item, Request, Dest).
```
出庫依頼は，
出庫指示書を出力する．

(f) 出庫依頼

```
container (Cont_num, [hansyutu (Item, Count) | C], Total) :-
    Total_1 := Total-Count, Total_1 =:= 0 |
    container_monitor (Cont_num).
```
搬出指示を受け取り，
搬出した後，コンテナの格納数が0ならば，
コンテナ搬出書を出力する．

```
container (Cont_num, [hansyutu (Item, Count) | C], Total) :-
    Total_1 := Total-Count & Total_1 > 0 |
    container (Cont_num, C?, Total_1).
```
搬出指示を受け取り，
搬出した後もコンテナの格納数が0でないならば，次の搬出指示メッセージを待つ．

(g) コンテナ

図-10 プログラム

(c) 積荷品票に出庫指示を送る．

(d) 未出庫分があるならば，再度，積荷品票に出庫指示を送る．

(4) 積荷品票 (**図-10**(d))

① 内蔵品より出庫指示を受け取ると，コンテナに搬出指示を送り，

(i) 出庫指示数>=積荷品票の数量ならば，

(a) 未出庫数を出庫指示数－積荷品票の数量とし，未出庫数を内蔵品に返す．

(b) 自分自身を消滅させる．

(ii) 積荷品票の数量>出庫指示数ならば，

(a) 未出庫数を0として内蔵品に返す．

(b) 積荷品票の数量を積荷品票の数量－出庫指示数とする．

```
test ← Load_message=[[num (1), (wine, 10), (beer, 20)],
                     [num (2), (whisky, 30)],
                     [num (3), (wine, 30), (whisky, 10)],
                     [num (4), (wine, 15), (beer, 15), (whisky, 10)]],
       Out_message=[(beer, 15, junko), (wine, 10, taro), (whisky, 30, jiro),
            (beer, 20, hanako), (whisky, 10, nanako), (wine, 35, kanae),
            (beer, 20, satomi), (whisky, 5, nobuko), (wine, 35, seiko),
            (whisky, 5, kumiko), (whisky, 5, takako)],
       nyuko_uketuke (Load_message?, [S1_wine_1, S1_beer_1, S1_whisky_1]),
       syukko_uketuke (Out_message?, [S2_wine_1, S2_beer_1, S2_whisky_1]),
       stock (wine, S1_wine_1?, S2_wine_?,_, Za_ch_wine, Za_ch_wine, 0),
       stock (beer, S1_beer_1?, S2_beer_1?,_, Za_ch_beer, Za_ch_beer, 0),
       stock (whisky, S1_whisky_1?, S2_whisky_1?,_, Za_ch_whisky, Za_ch_whisky, 0).
```

コンテナ

出庫依頼

(a) テスト・データ

```
| ?-cp test.
-------Zaiko Fusoku List -------
        Item=beer, Request=15, Destination=junko
-------Zaiko Fusoku List -------
        Item=whisky, Request=30, Destination=jiro
-------Shipment List of Goods -------
        Item= wine, Request=10, Destination=taro
-------Shipment List of Goods -------
        Item= beer, Request=15, Destination=junko
-------Zaiko Fusoku List -------
        Item=whisky, Request=10, Destination=nanako
-------Shipment List of Goods -------
        Item= whisky, Request=30, Destination=jiro
-------Zaiko Fusoku List -------
        Item= beer, Request=20, Destination=hanako
-------Zaiko Fusoku List -------
        Item= beer, Request=20, Destination=satomi
-------Shipment List of Container -------
        Container number=2
-------Shipment List of Goods -------
        Item=whisky, Request=5, Destination=nobuko
-------Shipment List of Goods -------
        Item=whisky, Request=5, Destination=kumiko
-------Zaiko Fusoku List -------
        Item=whisky, Request=5, Destination=takako
```

(b) 実行例

図-11 実行例

(5) 在庫不足 (**図-10**(e))

① 在庫数を受け取ると,

(i) 在庫不足数>在庫数ならば,

(a) 次の在庫不足へ在庫数を伝える.

(ii) 在庫数≧在庫不足数ならば,

(a) 出庫依頼を生成する.

(b) 新在庫数を在庫数−在庫不足数とし, 次の在庫不足へ新在庫数を伝える.

(c) 自分自身を消滅させる.

(6) 出庫依頼 (**図-10**(f))

① 出庫指示書を出力する.

(7) コンテナ (**図-10**(g))

① 積荷品票より搬出指示を受け取ると, コンテナの新数量をコンテナの数量−搬出数量とし,

(i) コンテナの新数量=0ならば, 自分自身を消滅させる.

(ii) コンテナの新数量>0ならば, 新数量をコンテナの数量とする.

このプログラムの実行例を**図-11** に示す. すべてのメッセージが並列に実行されるので, 初めの出庫依頼はコンテナの入庫処理が間に合わずに在庫不足となる.

5. モデル化の比較

モデルの作成とプログラムの作成について, 部分機能分割法と構成要素分割法の比較を行う.

モデルの作成の難しさは, 大きな差がなかった. これは次の理由によるものと考えている.

(1) 例題として与えられた仕様の全体の機能が大きくなかった.

(2) システムの構成要素が容易に想像できた.

しかし, 部分機能分割法は全体の機能を分割する作業が必要である. その結果, 多くの場合, 情報を管理する1段レベルの高いものが必要となる. 一方, 構成要素分割法は現実にあるものを単位として機能を考えるため, 機能分割に関する作業は部分機能分割法に比べて少ないと思われる. 問題にもよると思われるが, システムの機能が大きくなれば部分機能分割法は構成要素分割法よりモデル化が難しくなると考えられる.

プログラムの作成の工数は, プログラムの規模によって定量的に比較できると思われる. ここでは, プログラムの規模の尺度としてステップ数を使う. 2つの方法によるプログラム・ステップ数は次の通りであ

った．

（1） 構成要素分割法によるプログラムのステップ数：92ステップ

（2） 部分機能分割法によるプログラムのステップ数：143 ステップ

構成要素分割法のステップ数は部分機能分割法のステップ数の約3分の2である．部分機能分割法のステップ数が多い理由は，部分機能分割法によるプログラムが情報の管理を行っているためだと思われる．例えば，コンテナの処理について見ると，構成要素分割法によるステップ数と部分機能分割法によるステップ数は9対 18 であった．この理由は，コンテナ管理係がコンテナをコンテナ・リストで管理しているためにそのステップ数が増加したことによる．一方，オブジェクト“コンテナ”自体はそのような管理を必要としていないため，その部分に対応するステップ数がない．

次に，プログラムにおける実行の並列性について考える．また，コンテナの処理を例にする．コンテナ管理係はコンテナの情報を集中管理している．一方，オブジェクト“コンテナ”はそれぞれの情報のみを管理している．情報を集中管理すると，一つの情報にたくさんのアクセスが集中するため，情報へのアクセスがネックとなる可能性がある．一方，情報を分散処理するとそのようなネックがないため，処理の並列性が高くなる．そのため，“コンテナ”の方が情報を分散しているだけコンテナ管理係より処理の並列性が高いと思われる．コンテナの処理に関して，“コンテナ”には最大，“コンテナ”の数の並列性がある．一方，コンテナ管理係のコンテナの処理は逐次的である．構成要素分割法によるモデル化は情報を管理しない傾向があるため，CP の高い並列性を生かす上で有効であると思われる．

6. おわりに

在庫管理システムのプログラムを CP のオブジェクト指向プログラミングを使って作成した．プログラムの作成に際して2つのモデル化の方法を使った．一つは構成要素分割法で，もう一つは部分機能分割法である．この2つのモデル化の方法をプログラムのステップ数で比べると，構成要素分割法では部分機能分割法に比べて約3分の2のステップ数でプログラムを作成できた．これは，構成要素分割法によるプログラムが部分機能分割法によるプログラムに比べて情報を管理する処理が少なかったことによる．また，構成要素分割法を使うと情報が集中しないため処理の並列性が上がり，CP の高い並列性を生かすことが期待できた．

本共通問題のプログラムを作成した経験より，CP でプログラムを作成する際には，次のような指針が有効であることが解った．

（1） CP によるオブジェクト指向プログラミングを用いると，オブジェクトごとの隠蔽性が良く，プログラムの作成が容易になる．また，個々のオブジェクトが並列に実行できる．

（2） 構成要素分割法を使ってモデル化すると，情報の管理が少なくなるので処理自体が簡単になったり，高い並列性を出すことができる．

本報告に関して有益な討論をしていただいた ICOT Working Group 2 の委員の方々および ICOT 第2研究室の方々に感謝致します．また，本論文に関して有益なコメントをしていただいた富士通国際情報社会科学研究所の田中二郎氏および ICOT 第2研究室の近藤浩康氏に感謝致します．

参考文献

1) 竹内：論理型並列プログラミング言語—Concurrent Prolog, コンピュータソフトウェア, Vol. 1, No. 2, pp. 25-37 (1984).
2) Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol. 1, No. 1 (1983).
3) 山崎：共通問題によるプログラム設計技法解説，情報処理，Vol. 25, No. 9, p. 934 (1984).
4) Jackson, M. A.: Information Systems: Modelling and Transformations, Proc. 3rd International Conference on Software Engineering, IEEE (1978).
5) 米澤：オブジェクト指向プログラミングについて，コンピュータソフトウェア，Vol. 1, No. 1, pp. 29-41 (1984).
6) 二村，藤田：プログラム設計法 PAD/PAM とその効果，「プログラム設計技法の実用化と発展」シンポジウム，情報処理学会 (Apr. 1983).
7) 大木他：Concurrent Prolog によるオンライン在庫管理システムの記述，第26回プログラミング・シンポジウム報告集，pp. 57-68 (Jan. 1985).

（昭和60年2月4日受付）