ソフトウェア工学　60－2
（１９８８．６．２２）

# Ｃｏｓｍｏｓのプログラミング環境について

猪股　俊光、片野田　守人、西村　義行

豊橋技術科学大学　工学部

筆者らが開発中のＣｏｓｍｏｓはプログラム図式ＮｅＯを用いて並行システムの解析・設計を行うためのソフトウェア開発支援システムであり、エディタ、シミュレータ、デバッガ、プログラム変換系というツール群からなる。ＮｅＯはオブジェクト指向型計算モデルに基づいた並行システムのモデル記述のためのラベル付き有向グラフであり、並行システムはオブジェクトの集まりとそれらの間の相互作用によってモデル化され、モデルはＮｅＯの要素である特定の図記号の組合せからなるネットワークとして記述される。ＮｅＯプログラムはグラフィックエディタを用いて図記号の組合せとして作成され、プログラムの実行過程はシミュレータを用いて視覚的に表示される。さらに、デバッガを用いてシミュレーションの実行を制御しながらプログラムのデバッグが可能である。また、プログラム変換系を用いると自動的に２つのオブジェクトを１つに融合することが可能である。

# Programming Enviroments of The Cosmos

Toshimitsu INOMATA, Morito KATANODA, Yoshiyuki NISHIMURA
Faculty of Engineering,  Toyohashi University of Technology

1-1, Hibarigaoka, Tempaku-cho, Toyohashi-shi, 440 Japan

The Cosmos is a program development system, which has been developed by the authors, for the analysis and synthesis of concurrent systems represented by the program schema NeO. The Cosmos is composed of a graphics editor, a simulator ( NeO interpreter ), a debugger, and a program transformation system. The NeO is a labeled directed graph, which provides the models of the concurrent systems based on an object-oriented computation model. The concurrent systems are characterized by a collection of objects and their mutual interactions. The models of the systems are represented as a NeO program which consists of the specified symbolic nodes and labeled arcs connecting them. The program is constructed through the graphics editor. The simulator provides the graphical output about the progress of simulation. The user can debug his program by controlling the stepwise execution of simulation. The program transformation system automatically fuses two objects in the program into one object.

## 1．はじめに

複数の構成要素が並行して処理を行っている並行システムのモデリング・シミュレーションに関する研究が盛んに行われており、すでにいくつかの記述言語および処理系[1] が提案されている。筆者らは、並行システムの解析・設計を体系的に行うためのソフトウェア開発手法の確立を目指しており、その一環としてこれまでに並行システムのためのモデリング概念、記述言語、プログラム変換技法の開発を進めてきた。モデリング概念としてオブジェクト指向[2] の考え方を取り入れ対象システムをオブジェクトの集合とオブジェクト間の相互関係によってモデル化することを考え、モデル記述言語 ＮｅＯ（Nets for Object-oriected programming）[3] の開発を行った。ＮｅＯはオブジェクト指向型計算モデルに基づいたラベル付き有向グラフであり、プログラムは特定の図記号の組合せによって作成され、オブジェクトの構造および結合関係はネットワークとして表現される。ＮｅＯは図記号を用いたプログラミング（ビジュアルプログラミング [4, 5]）を行うための視覚言語としての役割を果たしており、今回報告するＣｏｓｍｏｓ[6,7] は、ビジュアルプログラミングによる並行システム解析・設計のための統合的ソフトウェア開発支援システムである。

2．でＣｏｓｍｏｓにおけるプログラミング環境、モデリング概念および記述言語ＮｅＯの概要について述べる。3．でプログラミングのためのエディタ、4．でシミュレータ、デバッガついて、5．でプログラム変換系について適用例とともに説明する。なお、オブジェクト指向に関する用語については断わりのない限り文献[2]に従う。

## 2．　Ｃｏｓｍｏｓの概要

### 2.1　プログラミング環境

Ｃｏｓｍｏｓは並列処理システムの解析・設計を統合的に行なうためのソフトウェア開発支援システムであり、エディタ、シミュレータ、デバッガ、プログラム変換系というツール群から構成されている。

① エディタ：対象システムのモデル記述を行うためのＮｅＯグラフィックエディタ。
② シミュレータ：モデルの動作のシミュレーションを行うためのＮｅＯインタプリタ。
③ デバッガ：実行トレースを通じてモデルの動作の検証を行うためのＮｅＯデバッガ。
④ プログラム変換系：ＮｅＯに特定の変換規則を施しながら、モデルの構造の解析・設計を行うための変換系。

これらは、図１に示すようにＮｅＯを共通のデータベースとして有機的に結合されており、いずれもグラフィック画面と対話的に作業が行えるようなマン・マシン・インタフェースを備えている。

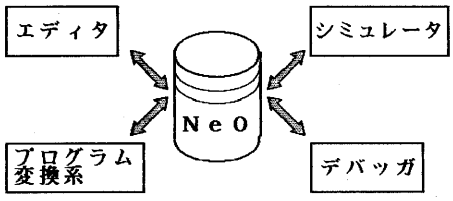


図１　Ｃｏｓｍｏｓプログラミング環境

### 2.2　モデリング概念

対象システムは、自分だけが参照／更新できる内部メモリと独自のメソッド（処理手続き）をもついくつかのオブジェクトが互いにメッセージパッシングを行いながら並列に動作するモデルとして表現される。メッセージパッシングは、メッセージキュー（以下キューと記す）とよばれるバッファを通じて行われ、オブジェクトどうしはキューで結合される（図２）。

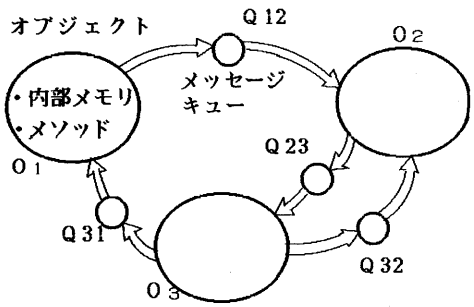


図２　メッセージパッシング型計算モデル

オブジェクトには少なくとも１つのキューが対応しており、オブジェクトがメッセージをキューから取り出すことをメッセージ受理、メッセージをキューへ入れることをメッセージ送信とよぶ。メッセージ送信は、送信先のオブジェクト（レシーバ）にとってメッセージの到着に相当する。あるオブジェクトにメッセージが到着すると、メッセージが受理可能かどうかのチェックが行われ、受理条件を満足したときそのメッセージは受理される。メッセージが受理されると、メッセージの内容に応じたメソッドが起動され、

・内部メモリの参照／更新や逐次計算、

・メッセージ送信、

という動作列の実行が開始される。なお、起動すべきメソッドは実行時にレシーバを起点としてクラス階層をさかのぼることにより特定される。メッセージの送信後、受理可能なメッセージが到着していればその処理を続けることになる。対象システムのモデルは、このようなオブジェクトの集合とそれらの間の相互作用を定めることにより定義される。

### 2.3 ＮｅＯ [3]

ＮｅＯは対象システムのモデルを図形式によって記述するためのネットであり、テキスト形式による表現法に比べ以下の特徴をもっている。

1) 並行システムの並列動作、同期、非決定性などの表現が容易である。
2) モデルを解析するためにグラフ理論技法が利用可能である。
3) 抽象性・可読性にすぐれている。

ネットは、キュー：○、内部メモリ：◎、メッセージ受信：▮、メッセージ送信：▯のノードとさらにラベル付きアークから構成され、クラス名、インスタンス名および計算の状態は特定のノードに対するマーキングとして記述される（図３）。

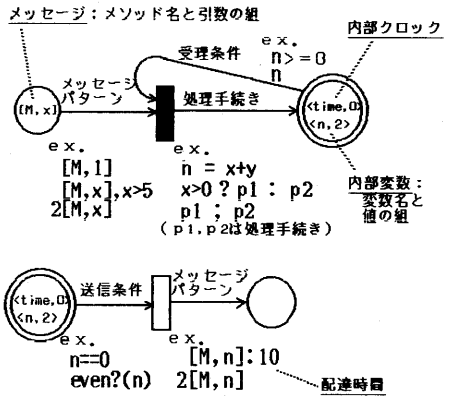


図３　ＮｅＯの要素

アークのラベルについて説明する（ラベルの構文規則は付録参照のこと）。○→▮ のアークには、メッセージの受理条件として、メッセージパターン、パラメータの条件、メッセージの個数を記述することができる。◎→▯ のアークには、メッセージの送信条件として、内部メモリ変数を引数とする述語を記述することができる。内部状態の条件を表す◎→▮ のアークにも同様の述語を記述できる。▮→◎ のアークには、処理手続きが記述され、代入文、条件文あるいはこれらの逐次処理が記述される。▯→○のアークには、送信メッセージパターンを個数とともに記述できる。Ｃｏｓｍｏｓでは、待ち行列系におけるサービス時間とメッセージの配達時間を考慮するため、オブジェクトに内部クロック、メッセージに配達時間をもたせることとした（時間の取り扱いは 2.4 参照のこと）。

計算モデルはこれらの要素を組み合わせた形として表現される（図４）。

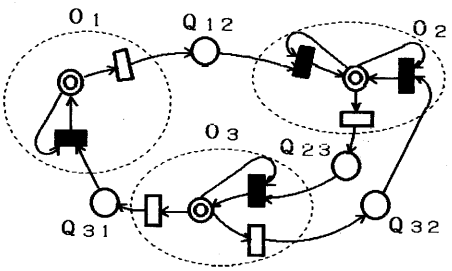


図４　ＮｅＯによるモデルの記述

ＮｅＯの解釈実行とは、次に示す遷移則に従いマーキングを変更することである。そして、ＮｅＯの実行の効果は、初期マーキングがＮｅＯを実行し停止するまでにどのように変化するかにより表すことができる[8]。

【定義】　ＮｅＯの遷移則

ＮｅＯの実行は、次の遷移則に従って行われる。

メッセージ受理：▮のすべての入力アークのラベルを満足するトークンが○（入力キュー）および◎に存在するとき、○のトークンは受理可能であるといい、受理が実行されるときトークンが取り除かれ、▮の出力アークのラベルに従うメモリの更新が行われる。

メッセージ送信：メッセージ受理後、▯の入力アークのラベルを満足する◎のマーキングが存在するならば、送信可能であるといい、送信が実行されるとき▯の出力アークのラベルに従うトークンがアークの指す○（出力キュー）に置かれる。 ■

図５に遷移の例を示す。(a)：$q_1$に到着したメッセージ[M,2]はラベル[M,a]を満足し、さらに、ｃの変数<x,1> がラベル x>=0 を満足するため、メッセージは受理可能である。受理が実行されると、各ラベルに割り当てられたトークンが取り除かれ、ラベルx=a が実行されメモリは更新される。(b)：変数ｘの値がｓの

入力アークのラベルの論理式x>0 を満足するため送信可能であり、送信が行われるとラベル[N,x] に従ったメッセージ[N,2]が$q_2$に入れられる。

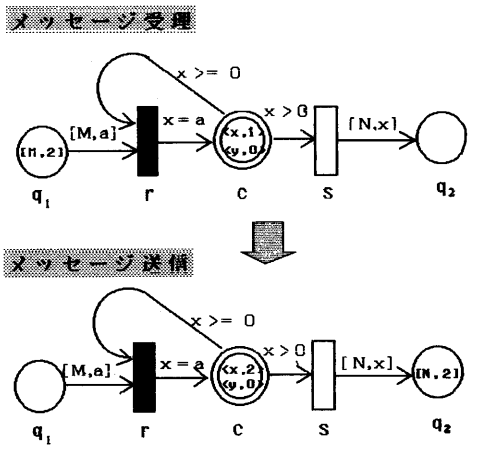


図５　ＮｅＯの遷移の例

## 2.4　時間の取り扱い

各オブジェクトは内部クロックをもっており、その値はメッセージ受理のときメッセージの到着時刻に応じて更新される。ここで、メッセージの到着時刻は、そのメッセージを送信したオブジェクトの内部クロックの値に配達時間を加えたものである。また、メッセージは到着時刻が時間的に古いものから優先的に受理される。シミュレータはオブジェクトの内部クロックの値をもとにオブジェクトの並行動作を実現している（並行動作は 4.1 参照のこと）。

## 2.5　ソフトウェア開発過程

Ｃｏｓｍｏｓを用いたソフトウェア開発は、次に示す作業を繰り返すことにより行われる。

① モデリング：2.2 のモデリング概念に基づいて対象システムのモデル化を行い、②に移る。すなわち、オブジェクトの集合とそれらの間の相互作用として対象システムをとらえる。

② プログラミング（３．参照）：エディタを用いて次の手順でモデルの記述、すなわちＮｅＯのプログラムの作成を行う。まず、(a) クラスとすべきオブジェクトの定義を行い、クラスとして登録する。(b) 定義されたクラスのインスタンスを生成する。さらに、(c) インスタンスの間の相互関係に応じてインスタンスどうしの結合を行い、③に移る。ここで、インスタンスの結合は、メッセージパッシングを行うインスタンスどうしの入出力キューを重ね合わせる（共有させる）ことにより行われる。

③ シミュレーション（4.1 参照）：キューおよび内部メモリの初期マーキングを与えた後、シミュレータ（ＮｅＯインタプリタ）を用いてプログラムの実行（ＮｅＯの解釈実行）を行う。実行結果がユーザの意図したものであれば作業を終了する。そうでなければ④に移る。

④ デバッギング（4.2 参照）：デバッガを用いてシミュレーションの実行を制御しながらプログラムの動作の確認を行い、①あるいは②に戻ってモデルやプログラムの修正を行う。

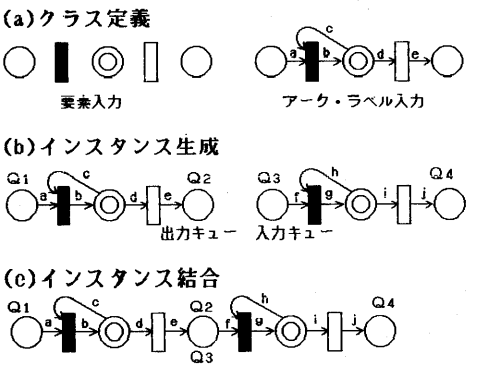


図６　プログラミング過程

## ３．　プログラミング

プログラミングはグラフィックエディタを用いてＮｅＯを入力することによって行われる。以下では、エディタのどの機能を用いてプログラミングが行われるかについて述べる。

### 3.1　クラス定義

クラス定義は以下に示すエディタの機能を用いながら行われる。

① ノード入力：ＮｅＯのノードを入力する機能。ただし、◎の入力の際にはクラス名と内部メモリ変数名をともに入力する。

② アーク入力：アークとラベルを入力する機能。ラベルの入力時に構文解析が行われる。

③ クラス参照：既に定義されているクラスを呼び出す機能。

④ 編集：ノードおよびアーク、ラベルの削除・移動・コピー・更新などを行うための機能。

⑤ クラス登録：上記①～④の機能を用いて作成されたオブジェクトをクラスとして登録する機能。

### 3.2 インスタンスの生成・結合

インスタンスの生成および結合はエディタの次の機能を用いて行われる。

① インスタンス生成：特定のクラスのインスタンスを生成する機能。クラス名、インスタンス名の入力が必要である。

② インスタンス結合：特定のインスタンスどうしを結合する機能。メッセージパッシングが行われるインスタンスどうしの指定が行われると、それらの間のメッセージのパターンが同じであるかどうかのチェックが行われる（図6-(c)のｅとｆ）。

③ アイコン化：インスタンスを特定のアイコンとして表示する機能。

以下、特に混乱の可能性のないかぎり、クラス、インスタンスをともにオブジェクトと呼ぶ。

### 3.3 グラフィックエディタの操作方法と作成例

基本的な操作方法は各機能に対応するエディタのメニューをマウスで選択しながら次々と指示を与えていく方式であり、例えばノードの入力はメニュー（○、◎など）からノードの種類を選択し、画面上に入力位置を指定するという手順で行われる。また、アークの入力はマウスを用いてメニューから直線あるいは曲線を選択し、２つのノードを結び、キーボードからラベルを入力するという手順で行われる。同様にして削除、移動などの機能を使うことができる。

以上の手順で入力したネットをStore（クラス登録）あるいはCreat （インスタンス生成）することによりプログラミングは行われる。インスタンスを生成する場合は、インスタンス名の入力および他のインスタンスとの結合を行った後、実行プログラムとして保存することができる（ファイル管理用のツールを呼び出して行う）。

クラス定義の例を図７に示す。オブジェクトCustomerは、メッセージ[next,no]を受理すると内部メモリNの値をno+1に変更した後、自分自身へ配達時間（10）を含むメッセージ[next,N]を送り、同時に出力キューへ配達時間（０）を含むメッセージ[start,N] を送るという動作を行うものである。

## 4. シミュレーション・デバッギング

### 4.1 シミュレーション

シミュレーションは、シミュレータ（ＮｅＯインタプリタ）がプログラム（ＮｅＯ）を解釈実行しながらマーキングを変化させることにより行われる。シミュレータはプログラムの解釈をアーク単位で行いながらオブジェクトの並行処理を実現している．アークの処理順序のスケジューリングは内部クロックの値を基に行われており、内部クロックの値が時間的に遅いオブジェクトのアークが優先的に処理されている。シミュレーション開始時には初期マーキングの入力が必要であり、実行が開始されると処理中のアークが点滅表示され、シミュレーションのビジュアル化が行われる。より詳しい情報（マーキングの履歴）の表示、あるいはシミュレーションの実行制御（ストップやバックトラック）を行いながらのシミュレーションは、デバッガの機能を用いて行うことが可能である。

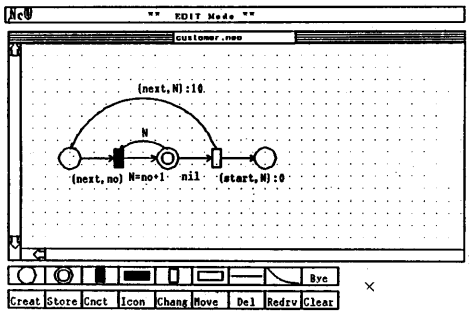


図７ プログラミング例

### 4.2 デバッギング

テキスト形式の手続き型言語のデバッガは、テキストをながめながらデバッグ作業を進めるために、現在実行中の箇所とソースプログラムとを対応づけるための反転表示機能などを備えているのが一般的である。このような従来の手法では並行に進行するプロセスの状況を理解することが難しく、並行プログラムのデバッグが困難である。そのため、Ｃｏｓｍｏｓには並行に動作するプロセス（オブジェクト）の実行過程をビジュアル表示するとともに実行の制御も対話的に行いながら並行プログラムのデバッグ作業を進めるためのツールとしてＮｅＯデバッガが用意されている。デバッガの備えている機能を以下に示す。

① マーキング表示：ユーザの指定する◎あるいは○のマーキングを表示する機能。

② シミュレーション制御：シミュレーションの中断、ワンステップ（アーク一つごと）の実行、後戻りを行う機能。後戻りではワンステップごとに直前の状態に戻る。

③ 環境設定：マーキングの変更、あるいは非決定的な処理（同時到着メッセージの処理など）の実

行順序を指定することによるシミュレーション環境設定機能。

④　結果表示：シミュレーションの終了時に指定された◎、○のマーキング変化の履歴および、▮、▯の履歴を表示する機能とアークの実行系列をグラフィック表示する機能。

### 4.3　実行例

図７のモデル（すでにインスタンス生成ずみのプログラム）に対してデバッガを適用した例を図８に示す。図８は初期マーキングの状態を示したものであり、CUSTOMER:QUE は CUSTOMERの入力キュー、CUSTOMER:MEM は内部メモリ、DEST:QUE は出力キューのマーキングを表している。キューでのマーキングは ” 到着時刻：［メッセージパターン、引数］”、内部メモリでのマーキングは”<変数名、値>”の形で表示される。ただし、キューが空の場合には”Empty”と表示される。

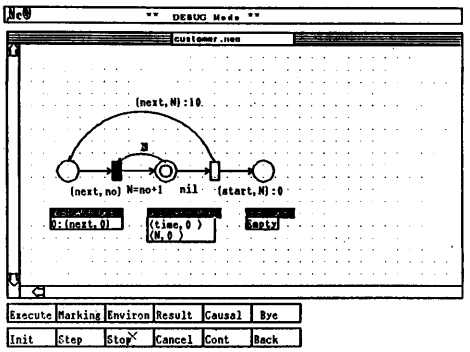


図８　初期マーキング

図９はシミュレーション実行中の画面であり、内部クロックの値が20のとき、○→▮のアーク[next,no]の処理で停止している状態である。これまでに、送信ずみのメッセージが３つあり、現在メッセージが１つ到着している。プログラムの停止箇所（ブレークポイント）は、アークのラベルが太字で表示（ N=no+1 ）される。また、現在処理が中断しているアークのラベルは、反転表示（ next,no ）される。

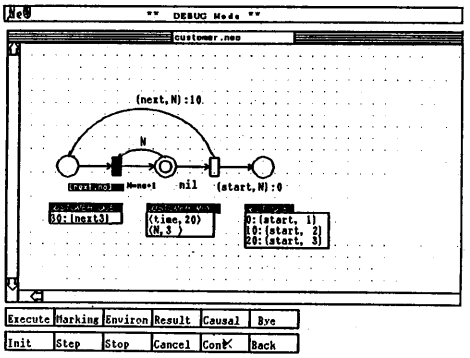


図９　デバッギング途中の状態

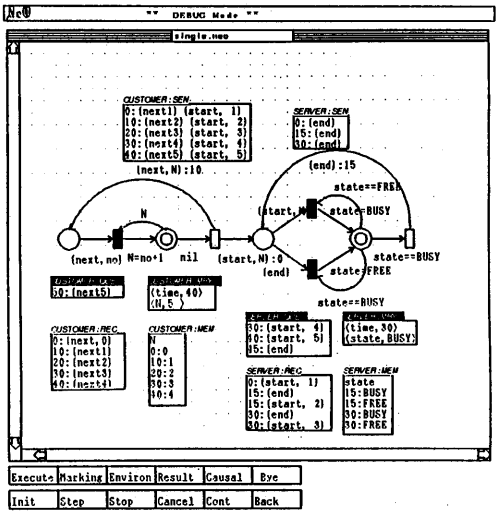


図１０　マーキングおよび履歴の表示

さらにもう１つのクラスServerを定義し、そのインスタンスをCustomerと結合して単一窓口待行列系をモデル化した例を図10に示す。ここで、Serverの動作は次の通りである。内部メモリstate が”FREE”の場合、メッセージ[start,N] を受理し、内部メモリstate を ”BUSY”に変更する。このとき、メッセージの送信条件は真なので自分自身へ到着時間（15）を含むメッセージ[end] を送る。内部メモリstate が”BUSY”の場合、メッセージ[start,N] を受理し、内部メモリstate を ”FREE”に変更する。このとき、メッセージの送信条件は偽となりメッセージの送信は行われない。

図10はシミュレーション終了後、受理メッセージ：▮（ CUSTOMER:REC、SERVER:REC ）および、送信メッセージ：▯（ CUSTOMER:SEN、SERVER:SEN ）、内部メモリ：◎（ CUSTOMER:MEM 、 SERVER:MEM ）の変更内容の履歴を表示したものである。送受信メッセージの履歴は ”時刻：［メッセージパターン、引数］” の形で、内部メモリの履歴は最初に変数名（ 、 ）が記述された ”変更時刻：変更前の値” の形で表示される。

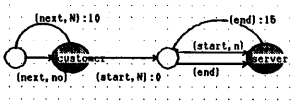


図１１　アイコン表示の例

Customer、Serverのインスタンスをともにアイコン表示して実行した結果を図11に示す。

図10のモデルのシミュレーション終了後、アークの実行系列を示したものが次の図である。文字列はＮｅＯのノードに対応しており、［＊］は○のマーキングの更新内容を、＜＊＞は◎のマーキングの更新内容を表す。この図からアークの実行系列とマーキングの変化系列の間の因果関係を読み取ることが可能である。

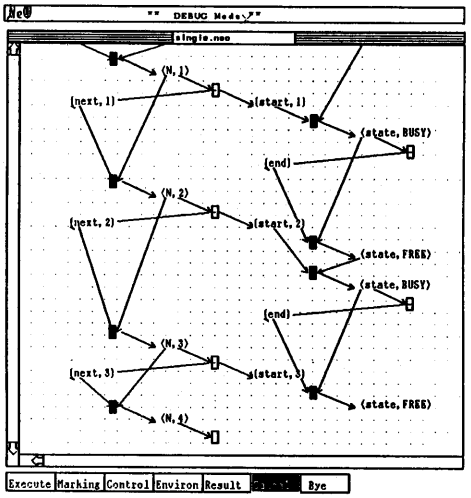


図１２　アークの実行系列

### 4.4 インプリメンテーション[7]

これまで述べたデバッギングの各機能を実現するために、シミュレーションの進行とともに内部メモリの変更内容、送受信されたメッセージの内容は、それぞれ内部クロックの値とともに内部変数スタック、送信スタック、受信スタックに記録される。アークのラベルの解釈実行が行われる際に、データ（内部メモリ、メッセージの内容）がスタックに push され、後戻りを行う際にpop される。データが内部変数スタックからpop されたとき、そのデータが内部メモリの値として設定される。データが送信スタックからpop されたときには、それと同じデータが送信先のキューから取り除かれる。データが受信スタックからpop されたときは、自分のキューへ戻される。

## 5.　プログラム変換系[8]

### 5.1 プログラム変換の概要

プログラム変換は仕様記述言語としてのＮｅＯどうしの範囲で行われ、その概要は基本的と思われるいくつかの規則に従って次々にプログラムを変換していくものである。これにより、プログラムの効率の改善（メッセージパッシングの回数の減少や内部メモリ変数の減少など）や構造の改良などが可能となる。

プログラム変換系は、変換規則と適用オブジェクトが指定されると、変換結果を自動的に出力するものである。現在オブジェクトの融合に関する規則が組み込まれている。

### 5.2 実行例

自然数の総和を求めるプログラムを例にとり変換系の実行結果を示す。generatorは自然数を生成し、sumはそれらの値の総和を計算する。２つのオブジェクトの融合を行う規則（fusion）を選択し、オブジェクトを指定すると次のような結果が表示される（図13）。

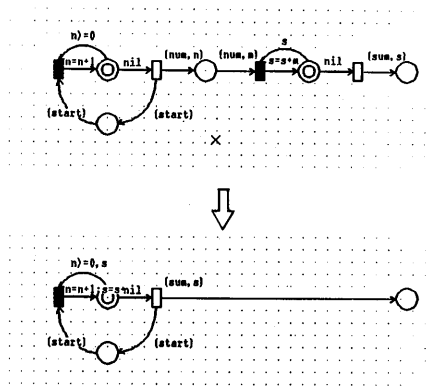


図１３　プログラム変換の例

## 6.　今後の展開と課題

現在、オブジェクトの並列処理は単一プロセッサ上でインタプリタが各オブジェクトの動作を疑似並列的に制御することによって実現している。今後は、複数のプロセッサがネットワークによって結合されている分散システムのモデリングおよび分散環境におけるプログラム開発支援システムとしてにＣｏｓｍｏｓを適用しようと考えている。簡単な例として図14 に示すよ

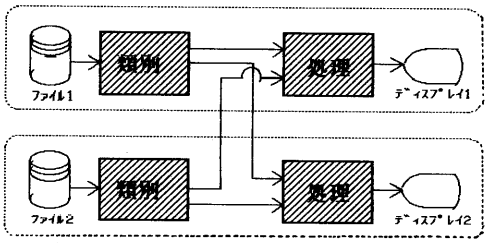


図１４　簡単な分散システム

うに、ファイルからデータを読み込み２つに類別しそれぞれに対し処理を行い結果を表示するシステムを考える。このうち類別する機能と処理を行う機能をオブジェクトとし、これらをＮｅＯを用いて単一プロセッサ上に実現した例を図15に示す。

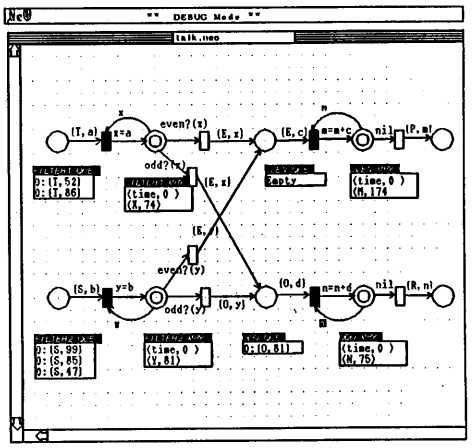


図１５　並行システム

このようなシステムをシミュレーションする際、オブジェクト間の同期をいかにして取るかということが問題となる。単一プロセッサの場合にはすべてのオブジェクトを見渡し、内部クロックをもとに処理の遅れているオブジェクトから処理する現在の方法が可能であるが、ネットワーク上に分散された複数のプロセッサによる実行（例えば点線で囲まれた部分を一台のプロセッサに割り当てた場合）では不可能となる。そこで、オブジェクトがキュー内のメッセージを処理する前に、メッセージの持つ時刻と、特定のオブジェクトの内部クロックをメッセージ通信によって比較しメッセージの時刻が最小である場合に処理を行う方法[9] などが提案されている。現在、効率の良い実現方法を検討中である。

## 7.　お　わ　り　に

本稿ではＮｅＯを用いて並行システムのモデリング・シミュレーションを行うための支援システムＣｏｓｍｏｓの概要について述べた。Ｃｏｓｍｏｓの特徴は、プログラムの作成、デバッグ、実行というプログラム開発作業を視覚化して行うために必要なツール群を備えていることにある。

今後の課題としては、６．で述べた分散型のプログラム開発環境の実現があげられる。また、モデリング概念のみならずプログラム編集作業におけるコマンド操作もオブジェクト指向方式、すなわち移動・削除などのコマンドを画面上の要素にメッセージとして送るという方式にすることが考えられる。

## 参　考　文　献


[１] Reijns,G.L.,Dagless,E.L.(eds.)：concurrent languages in distributed systems, NORTH-HOLLAND, (1985).
[２] 米澤明憲：オブジェクト指向計算の現状と展望 情報処理、Vol.29, No.4, pp.290-294 (1988).
[３] 猪股、西村：μ算法によるＮｅＯの操作的意味について、ソフトウェア基礎論研究会、88-SF-25 (1988).
[４] 市川、平川：ビジュアルプログラミング、bit、Vol.20, No.4, pp.30-38 (1988).
[５] Grafton,R.B,Ichikawa,T.(eds.)： Special Issue on Visual Programming, IEEE Computer, Vol.18,No.8, Aug. (1985).
[６] 片野田、猪股、西村：オブジェクト指向に基づく並行システムのためのモデリング・シミュレーション支援システム：Ｃｏｓｍｏｓ、第35回情報処理学会全国大会、1R-4 (1988).
[７] 片野田、田中、猪股、西村：並行システム・シミュレーション支援システムＣｏｓｍｏｓ－シミュレーション過程表示－、第36回情報処理学会全国大会、2K-4 (1988).
[８] 猪股、西村：プログラム図式ＮｅＯによるオブジェクト指向型言語プログラム変換、第36回情報処理学会全国大会、2H-4 (1988).
[９] 吉田隆一、所真理雄：離散形シミュレーションの分散時刻管理、コンピュータソフトウェア、Vol.4,No.1,pp.23-33(1987).


## 《付録》　アークのラベルの記法

ここでは、完全に形式的なラベルの構文規則を与えることは行わず、理解のために必要な部分を記すことにする。{A} は A が省略可能であることを表し、A* は AのＯ個以上の列を表し、[ ]はまとまりを表す。

```
<ラベル> ::=  <メッセージパターン> | <受理条件>
            | <送信条件> | <処理手続き>
<メッセージパターン> ::=
        {<定数>}<パターン>{:<定数>}[,<条件式>]*
<パターン> ::= [<メソッド名>[,<引数>]*]
<受理条件> ::= <変数>  |  <述語演算>
<送信条件> ::= <述語演算>
<処理手続き> ::=  <代入文> | <条件文>
                | <処理手続き>[;<処理手続き>]*
<述語演算> ::=    <述語>(<引数>{,<引数>}*)
            | <引数><比較演算子><引数>
<代入文> ::= <変数> = <式>
<式>::= <変数> | <原始演算子>(<引数>{,<引数>}*)
       | <引数><原始演算><引数>
<条件文> ::=
          <述語演算>?<処理手続き>:<処理手続き>
<引数>  ::=<定数> | <変数>
<比較演算子> ::= > | < | >= | <= | == | !=
<述語> ::= evenp | oddp
<原始演算子> ::= + | - | * | / | succ | pred | print
               | read | & | ¦
<メソッド名>::= <英文字列>
<変数> ::= <英文字列>
<定数> ::= <整数> |  <空>
```