ソフトウェア基礎論　37－19
プログラミング言語　27－19
（1990. 12. 14）

データフローメカニズムによる マルチエージェントシステムの実現

日下部茂　友清孝志　谷口倫一郎　雨宮真人

九州大学総合理工学研究科

あらまし　知識処理など高度の情報処理システムとして並列協調システムが有望視されている．そのようなシステムの実現には大規模な並列処理が不可欠であるが，我々は並列処理にはデータフロー方式が有効であると考え，並列協調型のシステムをデータフロー方式にもとづくメッセージフローシステムとして実現することを目指している．並列に動作する複数の処理体 (agent) をデータフローメカニズムによって実現することにより，処理体間だけでなく処理体内部でも並列処理が可能となる．本稿ではその計算モデルを示し，次にそのようなモデルを記述するため関数型言語 Valid を状態を持った処理体の記述と処理体間のメッセージ交換の記述ができるよう拡張した際，重視した点について述べる．最後に再帰とストリームの概念にもとづき，データフローメカニズムのもとで状態の更新とメッセージ処理を実現する機構について述べる．

Multi-agent System Based on Data-flow Mechanism

Shigeru KUSAKABE, Takashi TOMOKIYO, Rin-ichiro TANIGUCHI and Makoto AMAMIYA

Graduate School of Engineering Sciences, Kyushu University

abstract　The parallel coordinated system is promising to offer a basis for high level information processing system. We are trying to implement such a system as a messege flow sysytem based on data-flow mechanism. Massively parallel computing is essential in the parallel coodinated system and data-flow mechanism is well suited for parallel computation. In this paper, we first introduce a intuitive model of our system. Then we discuss the extension of functional programming language "Valid" to describe history sensitive agents and message passing between agents. Finally we present a mechanism, which implements history sensitive agents and messege passing based on the concept of recursion and stream processing under data-flow control.

1 まえがき

知識処理など高度の情報処理システムとして，並列協調型処理システムが有望視されている．並列協調処理システムは，多数の処理体 (agent) が互いに通信を行いながら情報を交換して自律的に動作し，協調して問題を解いていく[1]．このような並列協調型の計算では各処理体が並行して計算を進めるため高い処理能力が期待できる．協調型の計算モデルとして様々なものが提案されているが，我々は actor モデル[2]のようなメッセージ交換に基づく並列計算システムの実現を目指している．

我々は並列処理計算機の基礎方式としてデータフロー方式が有用であると考えている．データフロー・アーキテクチャは並列処理の同期制御をデータ依存関係にしたがって自然に実現できるという利点を持っている．われわれはまた前述のようなメッセージフローによる協調計算システムの実現には大規模な並列処理が不可欠であり，データフロー方式を基礎とするアーキテクチャが有効であると考えている．データフロー・メカニズムにより処理体間の並列性だけでなく，処理体内の並列性も利用することができる[7]．本稿ではまずデータフロー方式にもとづく並列協調計算システムの直感的なモデルについて述べる。対象とするモデルでは処理体はメッセージ交信によって情報を交換する．

データフロー方式では状態を直接持つことはできないが協調システムの各処理体は固有の状態値を持つ必要がある．そのため関数型言語 Valid[6]をメッセージ交換や状態の保持が容易に記述できるように拡張した際に考慮した問題について述べる．そして固有の状態を持った処理体からなるメッセージフローシステムを，データフローメカニズムにもとづき再帰とストリームの概念による処理系で実現する方法について述べる．

2 計算モデル

我々が実現を目指しているシステムの直感的計算のモデルについて述べる．

2.1 マルチエージェント

並列協調型システムは多数の処理体により構成され，処理体は動的に生成されまた動的に消滅する．各処理体は，消滅するまで活性化状態にあり一個の独立した計算主体として自律的に動作する．各処理体はメッセージによって情報の交換を行い，並列に処理を行う．このようなメッセージ・フローによる並列協調型システムでは，処理体は次のような性質を持つものとする．

- それぞれ固有の内部状態を持ち，自分自身の内部状態には直接アクセスできるが，自分以外の処理体の内部状態には直接アクセスできない．
- それぞれの処理体は独立していて，並列に処理を行う能力を持つ．
- 各処理体は任意の処理体と交信することができる．メッセージにより情報の交換を行い，メッセージの内容に従って処理を行う．

メッセージフローシステムではこのような処理体の間にメッセージが流れそのリンクは動的なものである．処理体は固有の内部状態といくつかの関数を持っており，メッセージによる選択的な関数の起動によって，メッセージに応じた状態の更新や新たな処理体のインスタンスの生成やメッセージ送信などを行う．

2.2 メッセージフロー

データフローメカニズムは並列処理の制御をデータ依存関係に沿って自然に実現できるという特長を持つ．しかし，データフロー方式では次のような点から知識処理などの高度の処理には不十分である．処理体は関数，演算として固定的にデータ依存グラフ中に配置され，それらの間を結ぶリンクをデータが流れるのでデータの流れが固定的である．また関数，演算の入出力関係は不変で，入出力が履歴に依存するような処理は直接実現できない．

データフローメカニズムを用いて前述のようなメッセージフロー協調計算システムを実現するには状態を持つ処理体を実現する必要がある．また処理体はストリームであるメッセージを処理する必要がある．このような計算システムを再帰とストリームの概念を用いて以下のようにモデル化する．

2.3 処理体

処理体のインスタンス Iはメッセージを介してのみアクセス可能で内部の処理の詳細は外からは見えない．処理体のインスタンスは概念的にはメッセージ・バッファ部 Q と処理ルーチン部 Pの 2 つに大別される．処理体に送られるメッセージ m_iはバッファの中に取り込まれ，キューに並べられる．処理体のインスタンス Iはメッセージ m_iを受け取るごとに処理ルーチンのインスタンス p_iを用意し，その実行を管理するという二重構造になっている．

処理ルーチン本体のインスタンスは動的に生成され再帰式の非同期ループ・インスタンスとして実現される[6]．ここで再帰式の非同期ループ処理とは再帰処理部に対し無名の再帰関数を定義し，異なる繰り返し毎に再帰本体のインスタンスをつくりだし，異なる再帰変数に対しては異なる再帰本体のインスタンスで処理を行うことである．処理ルーチンが履歴依存ではない場合は，ループの unfolding によって到着する各メッセージ m_iにループインスタンス p_iが与えられ処理ルーチンは並列に活性化される．この場合インスタンス間には依存関係がないので各処理は完全に非同期に行われる (図 1)．履歴依存のある処理ルーチンの場合もメッセージ m_i毎に新たなループインスタンス p_iが与えられ処理ルーチンは並列に活性化される．しかしこの場合はループインスタンス間には状態値に関して依存関係があり，あるループインスタンス p_iの実行により更新される状態値 s_{i+1}がまだ決定されていない場合は，ループインスタンス p_{i+1}で状態値 s_{i+1}に依存する部分は処理が中断される．

ループインスタンス p_iの実行により新しい状態値 s_{i+1}が決定され次第その値は次のメッセージ m_{i+1}に対するインスタンス p_{i+1}に送られ中断していた処理が実行可能となる (図2)．このように1つ前のメッセージの処理による状態に依存している部分はその状態値の決定まで処理は中断されるが，メッセージ毎にインスタンスを確保しておけば，その状態値に依存していない部分は先に処理を進めておくことができる．

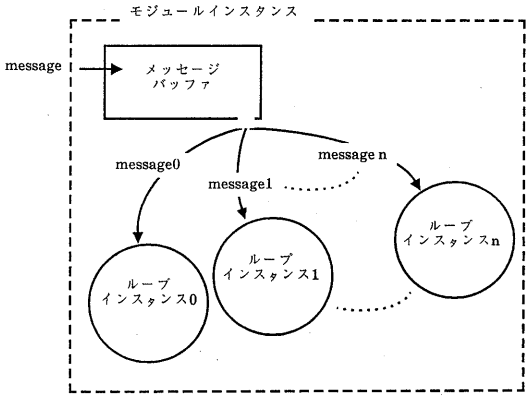

図 1:履歴依存のない処理体

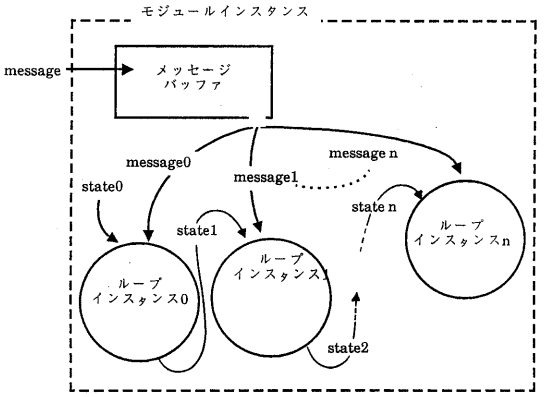

図 2:履歴依存処理体

メッセージは [message | return-point] の形をしている．ここで message はメッセージの内容で，return-point はメッセージの処理の結果の返し先である．複数の処理体間でメッセージの処理をたらい回しにする場合はこの return-point も一緒に転送することによって，最終結果の返し先を伝えていく (図3)．同期メッセージ送信を行うためには，実際に処理体にメッセージを送り出すときにメッセージにタグを付加する．メッセージの受信側ではこのタグによりメッセージ受理確認シグナルの要，不要を判断し必要ならばシグナルをメッセージの発信元に発信する (図4)．同期メッセージ送信を行いたい場合はこのシグナルによって同期をとることができる．

我々が実現しようとしているシステムは，内部処理はデータフローメカニズムによって動作する多数の処理体から構成され，それらの処理体がメッセージ交換によって相互に作用し新たなインスタンスの生成，状態の更新，メッセージ送信などを行いながら計算を進めていくものである．

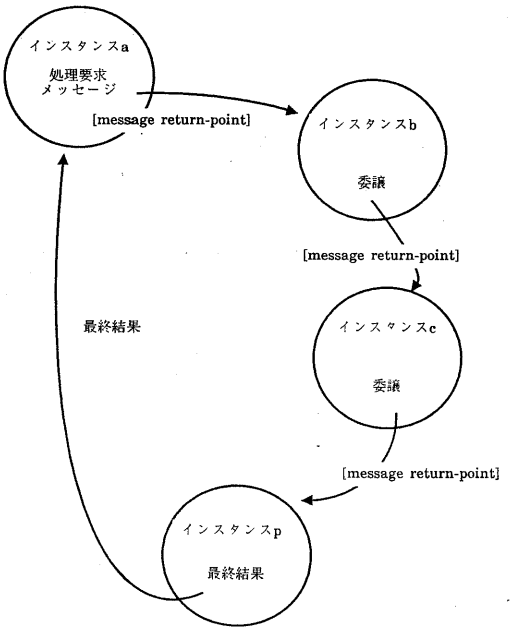

図 3:委譲

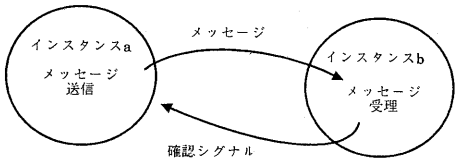

図 4:同期メッセージ送信

3 記述

以下に状態を持つ複数の処理体からなるメッセージフローシステムを記述するために関数型言語 Valid を拡張した際検討した点について述べる.

3.1 モジュールの記述

並列に動作する処理体の基本単位をモジュールとする. モジュールは固有の内部状態, 処理ルーチンを持つ. 内部状態を変化させるような履歴依存の処理ルーチンでは, その更新される状態についての情報は明示的にフィードバックするよう記述する. Valid では x = x + 1 のような循環定義は許されていず, 値が更新されるような状態値などは再帰式の記述にもとづいて次の再帰本体への入力として, 状態毎に異なるインスタンスで処理することを記述する. 状態の変更がなく履歴に依存しない処理ルーチンを持つ場合もある. またモジュールはメッセージキューを持つが, 記述レベルでは陽にそのメッセージバッファは見せないようにする. 処理系で自動的に再帰ごとにメッセージキューの先頭を取り出す処理を行う. 以下にモジュール記述の基本形の一例を示す

```
モジュール名 : module (局所状態初期値の引数)
    = { let 局所変数, 関数の定義 ···
                ···
      in for (states):(initial-states)
         recur (new_states)
         where new_states=
            case { predicate_1 → expression_1 ;
                        ···
                   predicate_n → expression_n
                 [ ; others → expression_0 ]
               end } }
```

従来の Valid では述部は単なる条件判定しか行っていなかったが, モジュールではメッセージのパターンマッチングにより実行式の選択を行えるよう拡張した. モジュールの再帰本体の recur で用いられる case では predicate_i にパターンを記述し実行式の選択を行う. つまり以下のような記述によって実行式を選択する.

$$\text{pattern}_i \rightarrow \text{exp}_i$$

モジュールの再帰式においてはメッセージキューは再帰本体への入力として記述はしないが, 処理系によって再帰本体ごとにメッセージキューからメッセージが一つ取り出されその再帰本体への入力となる. メッセージキューの残りは次の再帰処理への入力となる. 上のようモジュールの実行式の選択を記述すれば述部に書かれた pattern と処理系によって自動的に取り出されるメッセージとパターンマッチが行われ, マッチする pattern を矢印の左に持つ実行式が選択される. 選択された実行式の部分で状態の更新, メッセージの送出などを行う.

モジュールの本体定義においての実行式選択のパターンマッチでは, パターンマッチによって値がバインドされるような変数には先頭に ? が付けられる. 例として次のような場合を考える.

```
["deposit"| ?n] → exp
```

このような場合メッセージ mess の car 部が文字列 "deposit" ともう一つの要素 k からなるパターンであれば n に k の値がバインドされて式 exp が実行される.

モジュール定義の再帰式では終了条件の記述は省略される. 上述のように処理系が自動的に再帰ごとのメッセージ取り出しを行う. このメッセージの取り出しの際に処理系がメッセージキューの終了をチェックする. キューの終わりを検出すると, 処理系は再帰式の終了条件が成立したと判断しモジュールの再帰処理を終える.

3.2 モジュールの生成

高い並列性を得るために, 実行時に動的にインスタンスの生成, 消去を行う. インスタンスの生成は次のように記述する.

```
P=new(module_name,initial_states)
```

これにより module_name という名前で定義されているモジュールの新しいモジュール・インスタンス P が初期状態値 initial_states で生成される.

インスタンスの消去は次のように記述する.

```
s = erase(module_instance)
```

erase により module_instance に自分を消滅するよう指令するメッセージが伝えらえる. erase は send(後述) の一種で 'erase メッセージが指定したモジュールのメッセージキューに送られることになる. module_instance がそのメッセージを処理すると module_instance は消去され, その確認シグナルが s に送られる (正確にはシグナルが返されるのは指定したモジュールが 'erase メッセージを受け取ったとき. 詳細は 4 章参照). erase の実行では結果的に自分で自分のインスタンスを消去することになる. シグナル s の確認が必要ない時は次のように記述する.

```
!= erase(module_instance)
```

ここで!は処理の結果返されるシグナルを捨て去ることを表す.

3.3 メッセージの記述

メッセージ送信は次のように記述する.

```
(s,x)=send(module_instance,message)
```

これは module_instance に対しメッセージを送出し, その message に対する最終的な処理結果が求まればその結果を x で指定された返答先に送ることを表す. メッセージを処理した結果が必要ない場合, x の代わりに ! と書く.

ここで s には send によって送られたメッセージを相手のメッセージキューが受け取ったというシグナルが返される. s の値を確認することにより受信側がメッセージを受信したことを確認でき, 同期メッセージ送信を実現できる. 受信側でのメッセージの受理を待つ必要がない場合, x の

代わりに ! と記述すれば，メッセージの送出と他の処理を並行に行うことができ，非同期メッセージ送信を実現できる．

そのモジュールインスタンス自身を示すために self を用いる．あるモジュールインスタンスからそのモジュールインスタンス自身にメッセージを送りたい場合 module_instance に self を指定する．

メッセージの発信元に対し処理結果 result を返答する場合次のように記述する．

```
! = reply(result)
```

処理系では reply によって送り出されるデータをメッセージと区別し受信先のキューを通さずデータの到着を待っている場所に直接送りつける．

以上のようなメッセージ記述により同期メッセージ送信，非同期メッセージ送信が記述でき，必要な場合にはモジュール間の同期をとることができる．メッセージ交信によって未知変数を参照するような処理は，その変数の値が確定するまで処理は待たされ，値が確定すればデータ駆動メカニズムによって処理の中断が解かれる．またメッセージ交信において結果を待つ必要があっても，その結果に依存しない部分はデータフローメカニズムにより処理を進めることができる．そのためモジュールの内部でも，データ依存関係が許す限りの最大の並列性を活用することができる．

現在この拡張版 Valid を用いた在庫管理，哲学者の食事問題など様々なアプリケーションプログラムの記述を試みている[8]．種々の問題を記述することにより言語仕様，必要な機能の洗いだしを行っている．

4 実現機構

以上に述べたような処理体の機構をデータフロー概念にもとづきメッセージフローシステムとして実現する[7]．

4.1 モジュールの実現機構

モジュールインスタンスにはメッセージを介してのみアクセス可能でモジュール内の手続きもメッセージによって呼び出される．処理系ではモジュールインスタンスに送りつけられるメッセージごとに処理ループインスタンスを呼び出しメッセージの処理を行う．このため概念的にはモジュールインスタンス内に処理ループインスタンスがあるという 2 重構造となる．new(module_name,initial_states) によって module_name という名前で定義されているモジュールの新しいモジュールインスタンスが初期状態値 initial_states で生成される (図 5)．生成されるインスタンスはある物理 PE 上に割り付けられ，モジュールインスタンス間のメッセージ交信は PE 間のコミュニケーションネットワークを介して行われる．モジュールの定義はコンパイルされるとき，本体のコード (module_body) の他に履歴依存の状況などの情報を持った部分 (module_h) が付加される．新しくモジュールインスタンスが生成されるときにはその情報によって履歴依存性がチェックされる．生成されるモジュールインスタンスは履歴の依存性により異なる処理が行われる．以下にそれぞれの場合の処理について関数型言語 Valid でその処理を記述し (付録参照)，その処理機構の概要について述べる．

4.2 非履歴依存モジュール

モジュールの処理が履歴依存ではない場合，処理結果は実行のタイミングに関係なくメッセージの内容だけに依存する．そのためメッセージの到着順に関係なく処理インスタンスを呼び出して実行してやれば良い．履歴に依存しないモジュールの生成時に呼び出される call_module について以下に説明する．

まずプリミティブ get_message によりモジュールインスタンスが割り付けられている PE のコミュニケーションパケットキューからメッセージパケットを取り出す．このときメッセージ受理確認のアクノリッジを返す必要があれば，メッセージの発信者にアクノリッジのシグナルを返す．メッセージ送信式 send 式の関数としての実行値はこのシグナルの値である．

メッセージ授受で以上のような処理が行えるよう，モジュールインスタンス間で転送されるメッセージパケットは次のような情報を持つ．

```
[ sender | message ]
```

ここで message は [message_content | return_point] であり，sender，return_point は [module_instance | [loop_instance | entry]] の形をしている．これらの情報を用いて例えば sender が nil であればシグナルの返答の必要がないということを判断する．

同期メッセージ送信の実行により作り出されるメッセージパケットでは，sender にメッセージ受理確認シグナル返し先に対応する情報が入れられる．処理結果を必要とするメッセージ送信の実行で作り出されるメッセージパケットでは，return_point に値を必要としているエントリの情報をいれる．

取り出したメッセージが 'erase メッセージであれば r_ins によりモジュールインスタンスを消去する．'erase 以外のメッセージであればそのメッセージ処理を実行するインスタンス ins を get_instance により確保する．get_instance はインスタンス資源の空き状況を調べ，処理ループに対するインスタンスを確保するプリミティブである．処理ループインスタンスが確保されれば，execute によりメッセージに対する処理ループインスタンスの実行を行う．execute は処理ループインスタンスに対してメッセージを引数としてリンクし，その実行を起動する．

このように，非履歴依存モジュールの呼び出しに対しては，メッセージキューを作らずに即座にメッセージに対する処理インスタンスを呼び出すようにしている．非履歴依存モジュール処理の実行結果は実行タイミングには無関係でありメッセージの内容だけに依存する．これは実行結果が引数の値だけに依存する関数呼び出しの場合と同じと考えられ，非履歴依存モジュールの呼び出し結果は関数性を保っている．この点から，非履歴依存モジュールのメッセージによる呼び出しを，一般の関数の呼び出しと同程度の負荷で処理するよう処理系の最適化を試みている．

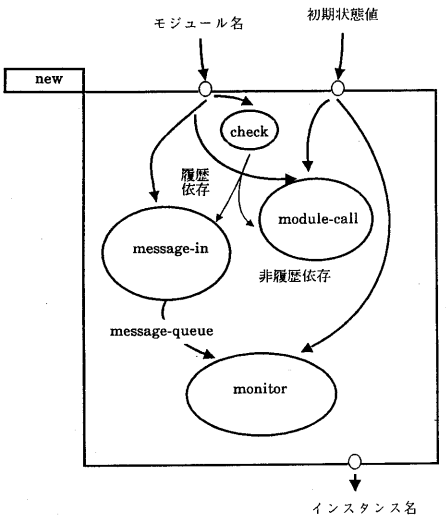

図 5:new

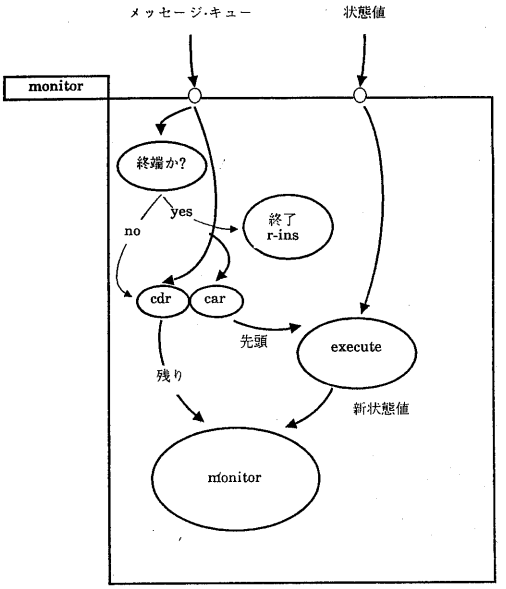

図 7:monitor

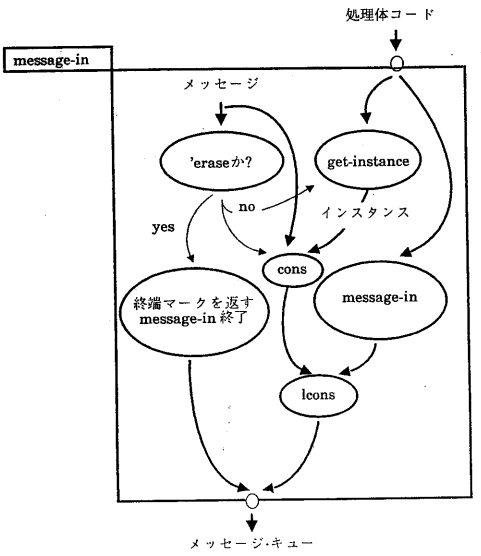

図 6:message_in

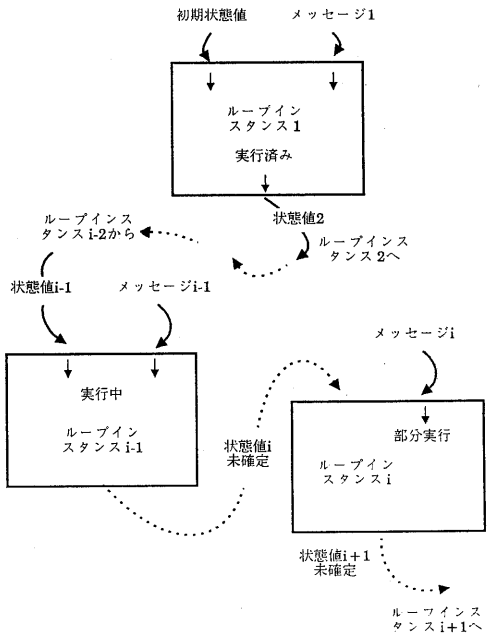

図 8:状態値の受け渡し

4.3 履歴依存モジュール

履歴依存であるモジュールの場合モジュールインスタンスに送られるメッセージはバッファの中に取り込まれ到着順にキューに並べられる．メッセージの処理は，更新される状態値とメッセージの到着順の整合性を保持するよう実行管理部の制御下で行われる必要がある．そのため新しく呼び出される履歴依存モジュールのインスタンス内ではメッセージを受け取りメッセージキューをつくる message_in と，処理ループ本体の実行を管理する monitor が呼び出される．以下に message_in と monitor の処理について述べる．

履歴依存モジュールで呼び出されるメッセージ処理部 message_in では以下のような処理を行う．まずプリミティブ get_message によりモジュールインスタンスが割り付けられている PE のコミュニケーションパケットキューからメッセージパケットを取り出す．このときもメッセージ受理確認の処理を行う．取り出したメッセージが 'erase メッセージであれば，メッセージキュー message_queue に終端のマークをいれ message_in を終了する．'erase 以外のメッセージであればそのメッセージ処理を実行するインスタンス ins を get_instance により確保する．処理ループインスタンスが確保されれば，メッセージと対にして message_queue に入れる．そして次のメッセージに対してこれらのメッセージ処理を再帰的に繰り返す (図 6)．

message_queue をつくる際には lcons(lenient_cons) を用い，メッセージストリームの先行評価が行えるようにする．lcons によってリストが生成されるときは，まずセルだけを先に用意し要素が求められた時点で書き込みが行われる[6]．この lcons を用い message_queue を生成することにより，message_queue を引数とする monitor では message_queue が完成していなくともその要素を参照できる．

処理ループインスタンスの実行を管理する monitor では次のような処理を行う．まず引数として渡されるメッセージキュー m_queue から先頭の要素を取り出す．その要素がキューの終端の記号であればモジュールインスタンスを消去し monitor を終了する．そうでなければその要素はメッセージとそれを実行する処理ループインスタンスの対であるので状態値と一緒に execute に送る (図 7)．execute では処理ループインスタンスに対してメッセージと状態値を引数としてリンクし，その実行を起動する．このとき execute では対応する状態値が求められていなくてもメッセージだけを先にそのインスタンスにリンクし，データフローメカニズムにより可能な限り処理を進めておくことができる．execute が完了すれば更新された新状態値が返される．この execute による処理インスタンスの実行と状態値の受け渡しの概要を (図 8) に示す．monitor では新状態値とメッセージキューの残りの部分を引数として再帰的に monitor を繰り返す．

以上のような処理メカニズムにもとづき，我々の研究室で研究を進めている Datarol アーキテクチャ[5]上で，状態を保持する処理体からなるメッセージフローシステムの実現を試みている．

5 まとめ

本稿では我々が実現を目指しているデータフロー方式にもとづく並列協調処理システムの直感的計算モデルを示し，状態を持つ処理体や処理体間でのメッセージ交換を記述できるよう拡張した関数型言語 Valid について述べた．そしてデータフロー・メカニズムのもとで再帰とストリームの概念によりそのようなシステムを実現することにより処理体間だけでなく処理体内での並列処理も可能とするであることを述べた．

今後はシステムのより厳密なモデル化を行いその動作を検証する．また，問題のより自然な記述のためさらに言語仕様の洗いだしを行う．そして我々の研究室で研究を進めている Datarol アーキテクチャ上での実現を目指す．

参考文献

[1] R. E. Filman and D. P. friedman : "Coordinated Computing", MIT Press(1984).

[2] Gul A. Agha: "Actors: A Model of Concurrent Computation in Distributed Systems", MIT Press(1986).

[3] 鈴木編："オブジェクト指向"，共立出版 (1986).

[4] A. Yonezawa and M. Tokoro : "Object-Oriented Concurrent Programming", MIT Press(1987).

[5] 雨宮："超並列多重処理のためのプロセッサ・アーキテクチャ",「コンピュータアーキテクチャ」シンポジウム論文集，p.p.99–108(1988).

[6] 長谷川，雨宮："データフローマシーン用関数型高級言語 Valid"，電子情報通信学会論文誌 D，Vol.j71-D, No.8, p.p.1532–1539(1988).

[7] 雨宮，長谷川："並列協調システムにおけるメッセージ処理機構"，ソフトウェア科学会第 4 回大会論文集，p.p.315–318(1987).

[8] 友清，日下部，谷口，雨宮："データフローに基づく並列オブジェクト指向言語による分散協調処理"，ソフトウェア科学会第 7 回大会論文集，p.p.61–64(1990).

```
function new(module_name,initial_states)
= { module_ins=my_instance() where
    [module_h|module_body]=module_name,
    !=case check(module_h) ->
                message_queue=message_in(module_b),
                !=monitor(message_queue,initial_states)
          otherwise ->
                module_call(message_b,initial_states)   }.


function message_in(b)
= { let message=get_message(),
        mes=car(message)
    in case mes='erase -> '#end
          otherwise -> { let ins=get_instance(b)
                         in lcons(cons(ins mes),message_in(b) }.


function monitor(m_queue,state)
= { let head=car(m_queue),
        ins=car(head)
        mes=cdr(head),
        res=cdr(m_queue)
    in if car(mes)='#end then r_ins
        else {let new_state=execute(ins,head,state)
              in monitor(res,new_state) } }.


function module_call(m_b,initial_states)
= { let message=get_message(),
        mes=car(message)
    in case mes='erase -> r_ins
          otherwise -> { let ins=get_ins(b)
                         in module_call(m_b,initial_states)
                        where !=execute(ins,cdr(mes),state) } }.


primitive my_instance() :
自分自身のインスタンス名を返す

primitive get_message() :
PE に到着するパケットからメッセージパケットを取り出す.
またメッセージ受理確認シグナルが必要ならそれを返す.

primitive get_instance(b) :
インスタンス資源をチェックし b を実行するインスタンスを確保

primitive execute(body,mq,st) :
mq,st をパラメータとしてインスタンス body にリンクし実行を起動

primitive check(h) :
モジュールが履歴依存かどうか調べる
```

付録:処理系の Valid による記述