マルチエージェントと協調計算のワークショップMACC'97　1997年12月18～20日

エージェント指向言語 Flage

条野文洋[*1]　佐藤 仁孝[*2]　加藤 哲男[*3]　永井 保夫[*4]　本位田真一[*5]
情報処理振興事業協会(ＩＰＡ)　新ソフトウェア構造化モデル研究本部

*Flage*は、ネットワーク上の分散環境において、ユーザや他のソフトウェア・モジュールからの要求の変化に自律的に適応するソフトウェアを構築するためのプログラミング言語である。本稿では*Flage*の言語機能の概要を紹介する。また、いくつかのプログラム例を参考資料として付録に添付する。

1. はじめに

ネットワーク環境においては，ネットワークを通してデータやプログラムを複数の計算機の間で容易に転送したり，複数の計算機上で稼働しているプログラムを相互に連携させることができる．さらに，近年，モバイルエージェントの言葉に代表されるように，プログラム自身がその実行中にネットワーク内の計算機上を移動するシステムも開発され始めている．加えて，ソフトウェアがネットワークを通して広まり，それが多様な場面で用いられれば用いられるほど，さまざまなニーズを持つユーザから新しい機能拡張や機能変更を要求される．こうしたネットワーク環境下では，ソフトウェアをとりまく環境はたとえば以下の場合に変化する．

- プログラムを他の計算機上からダウンロードし，そこで実行をさせる場合や他の計算機上にプログラムを送信し，送信先で実行させる場合など，プログラムが移動した結果として環境が変化する場合．

- ユーザが要求する機能が変化した場合など，ソフトウェアをとりまく環境自身が変化する場合．

こうした変化内容を設計時に予測することは不可能であり，しかも時には動的に変化が起こり得る．こうした変化に稼働時に動的かつ自動的に適応したり，そうでなくとも極力人手を労さずに適応できる柔軟なソフトウェアシステムは，ネットワーク環境の利点を効果的に活用し，より安定的な情報サービスを提供するシステムになることが期待できる．

Flageでは、こうしたソフトウェアを構成する要素として，従来のソフトウェアモジュールに代えて以下の性質を持つモジュール(エージェント)を導入している。

自律性： 自らのプログラムの実行を制御したり、プログラム自身をデータとして扱い，動的に変更できる．

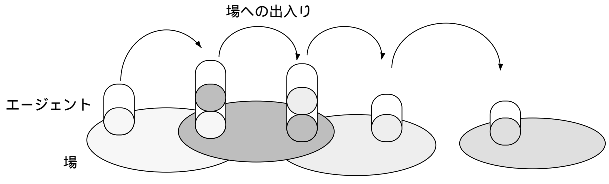
移動性： ネットワーク上を移動し，移動先に存在するプログラムを獲得したり，移動先でプログラムを実行できる．

協調性： 不特定の複数エージェントと通信でき，情報を共有することができる．

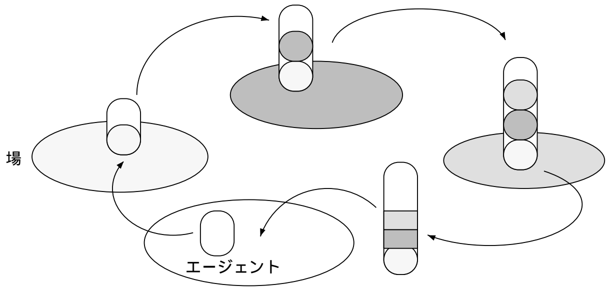
Flageのエージェントは分散システム上で稼動する非同期並行オブジェクトとなっている。自律性を実現するために，エージェントはメタとベースの二階層で構成される．さらに，移動性,協調性を実現するために，Flageでは場と呼ばれるオブジェクトを導入している。場はエージェントにプログラムを提供したり，エージェント同士で協調動作を行なうための仮想的な場所である。ネットワークを構成するマシンノード上で稼動する処理系もノードという特殊な場としてモデル化され，ノード内に様々な場が構成することができる．エージェントが場に入ると，そこに与えられているメソッド（場のメソッド）を自動的に獲得する．場のメソッドには，その場でのみ有効なローカルなメソッドと場から持ち出すことのできるグローバルなメソッドが存在する．また，場はエージェント間の共有情報のための属性を持つ．さらに，エージェントが場の中に居るエージェントに向かってメッセージをブロードキャストすることが可能である．

エージェントの適応性を実現するメタプログラミング機能として、場を以下のように利用できる。

1. エージェントは複数の場に入ったり，移動し，状況に応じて場のローカルなメソッドを動的に変更する．



2. 場を自律的に渡り歩き，必要なグローバルなメソッドを獲得する．



[*1] 三菱総合研究所より出向
[*2] アイネスより出向
[*3] 富士通FIPより出向
[*4] 東芝より出向
[*5] 現 東芝

2. Flage

Flageの言語機能について概説する。詳細についてはマニュアル[1]を参照されたい。

2.1 エージェント

エージェントは，データを保持する属性とそれに対して操作を行なうメソッドで構成される並行オブジェクトである．エージェントの状態（属性の値）は自分自身によってのみ変更できる．エージェントはお互いにメッセージ通信を行いながら，それぞれ独立並行に動作する．各エージェント内の動作は逐次実行である．エージェントの実行はそのエージェントが受信したメッセージをメソッドによって処理することにより起動される．エージェントからのメッセージは，まず，エージェントのメッセージキューに入れられ、キューへの到着順に処理される．

2.1.1 メタ構造

自らの定義を動的に変更する機能を実現するため，各エージェントはメタレベルとベースレベルの２相構造（メタ構造）を持つ．各レベルで属性およびメソッドを定義できる．メッセージもメタレベルのプログラムで処理されるメッセージとベースレベルのプログラムで処理されるメッセージの２種類が存在する．メタレベルのプログラムでは，メタレベルの状態だけでなく，ベースレベルのプログラムや状態を変更する手続きを定義できる．一方，ベースレベルプログラムはベースレベルの状態のみを変更できる．

メタレベルでは，ベースレベルのメッセージをベースレベルの定義で解釈実行するインタプリタとそれに関連する手続きを定義できる．具体的には，ベースレベルのインタプリタの，所定のプロトコル(メタプロトコル)を利用したカスタマイズ，および，カスタマイズしたそのインタプリタから呼び出される手続き(たとえば，ベースレベルの定義変更手続きなど)を定義できる．メタプロトコルはベースレベルメッセージのインタプリタが特定の時点で行う処理を再定義するための言語機能である．たとえば，受け取ったベースレベルメッセージの処理方法をメタプロトコルを利用して再定義することができる．

ベースレベルのメッセージは，メタレベルの定義によってカスタマイズされたインタプリタで処理され，メタレベルのメッセージは処理系が提供するデフォルト・インタプリタによって処理される．ただし，メタレベルのメッセージはベースレベルメッセージより優先して処理される．

2.1.2 エージェントの定義形式

エージェントの定義は以下の形式の項で表現する。

aidはエージェントの識別子，$attr_i$は属性名、$Init_i$はその初期値である。メソッド定義部において`$Attr`と記述することにより、`Attr`に保持されている値を参照できる。`Attr`と記述された場合は、そのまま属性名の記号として扱われる。

$mthd_i$は、以下の形式のメソッドの定義である。

agent(aid,
　meta{
　　attribute ($attr_1 = Init_1, ..., attr_{nm} = Init_{nm}$)
　　method ($mthd_1, ..., mthd_{mm}$)
　　spec ($PrologClause_1, ..., PrologClause_{tm}$) },
　base{
　　attribute ($attr_{nm+1} = Init_{nm+1}, ..., attr_{nb} = Init_{nb}$)
　　method ($mthd_{mm+1}, ..., mthd_{mb}$)
　　spec ($PrologClause_{tm+1}, ..., PrologClause_{tb}$) }
　)

```
Method ::  (   Method_1;
               Method_2;
               ...
               Method_n )
```

左辺の`Method`はメッセージ通信インタフェースとなるメソッド(パブリックメソッド)のパターン(`#Term`の形の項)，またはエージェント内手続き(プライベートメソッド)の呼びだしパターン(`#`が頭についていない項)である．

右辺は`Method`の処理手続き本体で，`Method_1`，...，`Method_n`の順番で逐次的に実行される．各`Method_i`$(1 \leq i \leq n)$は，属性やパラメタ（prolog変数）への代入、メッセージ送信、if文、プライベートメソッド呼びだし、のいづれかである。繰り返しはメソッドの再帰呼び出しで表現する．

`spec`では、値を計算するための、副作用のない関数の定義を行なう。関数を表現したゴールは、`spec`内でトップレベル述語となる二引数以上のゴールであり、もっとも右の引数を除いて入力とし、もっとも右の引数を出力とする。したがって、関数は一つ以上の引数をもたなくてはならない。ゴールが失敗した場合、エラー値(`error`)を返す。

また、prologの組み込み述語自身もブール値関数として利用できる。

関数呼びだしはメソッド本体部の以下の箇所で行なわれる。

- 属性またはパラメタへの代入文の右辺

- if 文の条件部

関数の計算では、まず、`$attr`の形式の式が属性値に変換され、その後、`spec`でのゴールが呼び出される。計算は関数呼び出しの項に対し、最内最左戦略で行なわれる。

2.2 場

場はエージェントに様々な情報を提供する一種の受動的なデータオブジェクトである．場は以下の項目で定義される．

場固有のメソッド： エージェントが場の中で利用するメソッド，関数および属性を定義する．定義形式はエージェントのベースレベルと同じである．場のメソッドには，その場でのみ有効なローカルなメソッドと場から持ち出すことのできるグローバルなメソッドが存在する．たとえば、場固有のメソッドとしてmethodAが定義されている場合，エージェントが異なる定義のmethodAを事前に持っていたとしても，場に入った時点で，元の定義はオーバーライドされる．場を出る時にmethodAがローカルなメソッドとして定義されてい

る場合，場を出た時点でmethodAの元の定義が復活する．グローバルなメソッドとしてmethodAが定義されている場合には，場固有のメソッド定義がエージェントの新たな定義となる．属性や関数も同様に扱われる．

場の出入りに関する制約： 場に入るときや場に入っているとき、または場を出るときにエージェントが満たさなければならない条件を記述する．条件はエージェントの属性に関するブール式である．場の条件は，たとえば、場の固有メソッドがアクセスする属性に関する条件を定義することに利用される．さらに不特定のエージェントが入ることを防いだり，場に入るエージェント数を限定したりする一種のセキュリティ機能を実現することにも利用される．また，場を渡り歩く経路にも制約を与えることができる．たとえば場Aに入るためには，場Bに入っていなければならないといった制約を記述できる．

場固有の属性： 場におけるエージェントの共有知識を保持するための属性定義である．各属性に対するアクセスは場で定義されたread/write手続きにしたがって行なわれる．場固有のメソッドのインタフェース情報を保持する属性，他の場へのリンク情報を保持する属性などをデフォルトの属性として持つ．

場はエージェントやユーザによって動的に生成できる．場の削除は，その場の中にエージェントが存在しない場合に，その場を生成したエージェントまたはユーザによってのみ可能である．

場の定義は以下の形式の項で表現する。

```
field($fid$,
   enter $EnCond$ & always $AlCond$ & exit $ExCond$
     where $PrologClauses$,
     { global(
           attribute ( $attr_1 = Init_1, ..., attr_a = Init_a$ )
           method ( $mthd_1, ..., mthd_m$ )
           spec ( $PrologClause_1, ..., PrologClause_p$ ) ),
         local(
           attribute ( $attr_b = Init_b, ..., attr_{ba} = Init_{ba}$ )
           method ( $mthd_b, ..., mthd_{bm}$ )
           spec ( $PrologClause_b, ..., PrologClause_{bp}$ ) ) },
   field_attribute (
       ($fattr_1 = init_1$ | read $rp_1$, write $wp_1$, $rw\_proc_1$),
       ($fattr_2 = init_2$ | read $rp_2$, write $wp_2$, $rw\_proc_2$),
       ...)
   )
```

fidは場の識別子である。3つの制約に関する条件式、$EnCond$、$AlCond$、$ExCond$は属性値に関する等式または不等式で、$PrologClauses$はこれらの条件式の中で利用する関数の定義(エージェント定義の`spec`と同じ形式)である。`where`の部分は省略可能である。$fattr_i = init_i$は場の属性と初期値を指定している。rp_i と wp_i は、その属性に対する読み込みおよび書き込みの許可を指定し、rw_proc_iはその属性の読み込みおよび書き込みの手続きを定義する。rw_proc_iは省略可能である。

2.3 場に対するエージェントの動作

エージェントが場に入る際には，場の制約を参照し，自分の状態（属性値）が場の制約を満たすかどうかのチェックをエージェント自身で実行する．その結果，もし制約を満たしていない場合，エージェントはその場に入ることに失敗する．ただし，エージェントは制約を満たすよう自らを変更する適応手続きを起動させることも可能である．この手続きはメタプロトコルを通して定義する．

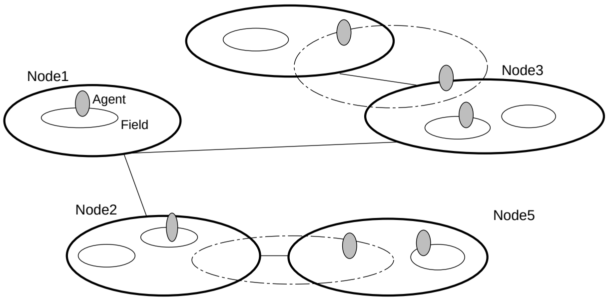
エージェントが場に入るために場の制約に適応する場合，エージェントは場のデフォルト属性`enter_adapt`から適応の方法（場に入るための制約を満足する属性値を求めるベースレベルメソッド）を参照・獲得できる．`enter_adapt`には`enter_adapt`(AuxParam,Field, Result)という共通のインタフェースを持ったメソッドの定義が格納されている[1]．適応後の属性値のリストはResultにバインドされる．

条件チェックで制約を満たす場合，エージェントは場に入ることができる．場に入ることで，場のメソッドを自動的に獲得する．

エージェントが場を出る場合も，入る場合と同様の手続きで動作する．以上の手続きにより，エージェントが場の出入りを通して常に場の制約を満たしていることが保障される．

3. Flage による分散システム

Flageで構築する分散システム（Flageシステムと呼ぶ）は，Flage処理系が稼働する複数のサーバマシンを含むLAN上に構築される．Flageのプログラムでは，Flage処理系はノードと呼ばれる組み込みの場として表現される．エージェントは場を出入りするのと同様にノード間を移動できる．SMTPを利用した移動プリミティブメソッドによって、ゲートウェイを経由して，他のLANに存在しているFlageシステムにエージェントが移動することも可能である．こうしたノード間の通信はネットワークハンドラによって実現されている。処理系起動時にノード名、ネットワークハンドラの稼働しているホスト名、ネットワークハンドラが利用しているポート番号を指定することにより、処理系とネットワークハンドラが接続され、ノード間通信が可能となる。



エージェントは，いずれかのノードに必ず属していなくてはならない。場には一つのノード内に生成されるもの(実線の楕円)と場に跨って生成されるもの(破線の楕円)の2種類が存在する。図のノード間の直線は，エージェントがネットワーク上で直接移動できるノードの関係を示している。この関係は各ノードのFlage処理系が接続しているネットワークハンドラの設定に依存する。

---

[1] 場を出るための制約に適応する場合には，デフォルト属性exit_adaptが利用される

ユーザはWWWブラウザを通してエージェントと通信を行ない，アプリケーションとして利用する[2]．

謝辞　本研究は、産業科学技術研究開発制度「新ソフトウェア構造化モデルの研究開発」の一環として情報処理振興事業協会（IPA）が新エネルギー・産業技術総合開発機構から委託をうけて実施したものである。

参考文献

[1] 粂野 文洋: “Flage-Prologマニュアル”, http://www.ipa.go.jp/NEWSOFT/public/Flage

A　移動するエージェントのプログラム例

```
/* 指定されたノードに移動し、そこでユーザからメッセージをも
   らって、元のノードに戻るエージェントのプログラム */

create_agent(
 agent(agent0,
  meta{
   method (
   #go(Node):: (
     %現在のノード名を獲得する
     node_name(PN);
     %ユーザにメッセージを送信
     user <= m(1,'I go to the next node via the Libretto');
     %ノードを移動
     move_node(PN,Node);
     user <= m(1,'Hi! I am from the next node'(PN));
     %ユーザに返事待ちメッセージを送信
     R = (user <= m(1,'Message, please'));
     move_node(Node,PN);
     user <= m(1,'I came back, and I got a message'(R))
              )
         )},
  base{null})).
```

B　メタプロトコルのプログラム例

```
/* 未知のベースレベルメッセージを受信したら、メタ属性others
   で指定されたエージェントにそのメッセージを転送するエージ
   ェントのプログラム。exec_message/2は受信したベースレベル
   メッセージの処理を再定義するためのメタプロトコル。*/

create_agent(
 agent(agent1,
  meta{
   attribute (others = [agent3])
   method (
     exec_message(_Sender,Msg) :: (
       % Msgとベースレベルのパブリックメソッドを照合
       know(Msg,Result);
       % Msgとマッチするメソッドがあれば、そのメソッドで処理
       % なければ、他のエージェントに転送
       if Result = false then (OtherAgent=hd($others);
                               OtherAgent <= Msg )
                         else interpret(Msg,_)
             )
      )
    spec ((hd([X|_],Y) :- Y = X))
      },
  base{
   attribute (state = 2)
   method (
     #comm(Number) :: (
       % selfIDは自分の識別子を返す関数
       Self = selfID;
       user <= m(0, receive(Self,Number));
       % 属性stateへの代入文
       state := plus(Number,$state);
       agent2 <= m(0,comm($state))
        )
      )
    spec ((plus(X,Y,Z) :- Z is X+Y))
    })).
```

C　場のプログラム例

```
/* エージェントがこの場に入るためには、エージェントが
   ベースレベル属性attrAを持ち、その値が正の数でなければ
   ならない。この条件はグローバルメソッドg_proc/1が正常に
   稼働するための条件でもある。 */

field(field_A,
  enter $attrA >0 & always true & exit true,
 {global(
    method (
     # g_proc(A) :: (
         Result = divfunc(A,$attrA);
         ME = selfID;
         user <= m(base, re(ME,Result))))
     spec (divfunc(X,Y,Z) :- Z is X/Y)),
  local(
    method (
     # l _proc(A) :: (
         ME = selfID;
         % 場の属性test1からの読み込み
         read(field_A,test1,L);
         user <= m(base, re(ME,L));
         % 場の属性test1への書き込み
         write(field_A,test1,[ME|L])))
               )},
  % 場の属性test1は初期値が[]で、場の外からも読み書きが
  % 可能である。
  field_attribute ((test1 = []|read public,write public))
         ), [[],[],[]]).
```

---

[2] 現バージョンでは、Tcl/Tkがデフォルトのユーザインタフェースとなっている。