エージェント指向言語 Flage

田原康之[†1] 粂野文洋[†2] 大須賀昭彦[†1] 本位田真一[†1]

新ソフトウェア構造化モデル研究本部
情報処理振興事業協会（ＩＰＡ）

協調型ソフトウェア・アーキテクチャを記述するためのエージェント指向仕様記述言語 Flage について述べる．協調型ソフトウェア・アーキテクチャは複数のエージェントから構成され,Flage はエージェントおよびエージェント間通信の仕様記述に用いられる．更に Flage では、各エージェントはメタ階層に分かれており、これにより柔軟な処理の記述が可能となる。また複数のエージェントの協調の範囲を表わす場の概念を有している。そしてカテゴリ論に基づく数学的な意味論が与えられているので、厳密な検証によりプログラムの正しさを保証することができる。本稿では，提案する Flage の言語仕様，およびその意味論を述べる．

An Agent Oriented Language Flage

Yasuyuki Tahara, Fumihiro Kumeno, Akihiko Ohsuga and Shinichi Honiden

Laboratory for New Software Architectures
Information-technology Promotion Agency, Japan (IPA)

An agent oriented language Flage with which we can specify cooperative software architecture is proposed. Cooperative software architecture consists of several agents and Flage is used to specify the agents and the communication of them. In Flage, each agent has its meta-hierarchy which enables specification of flexible behaviors. There are also fields which represent range of cooperation. Moreover, since mathematical semantics based on category theory is given, it is possible to assure validity of programs by rigorous verification. In this paper, the specification and semantics of Flage is explained.

[†1] 東芝より出向
[†2] 三菱総合研究所より出向

1　はじめに

オブジェクト指向パラダイムは，1990年代におけるソフトウェア技術の主流を歩むといわれている．しかしながら，大規模化，複雑化する分散システムにおいて，動的なシステム構成の変更への追従という観点では，多くの課題を残している．そうしたオブジェクト指向パラダイムの弱点を克服することを目指しているのがエージェント指向パラダイムである．エージェント指向パラダイムにおいては，オブジェクト指向に自律性を付加したエージェントが，環境の変化に対して，自律的に追従していくことが可能となる．我々は，エージェント指向パラダイムのひとつとして，協調型ソフトウェア・アーキテクチャの実現を目指している．本稿で述べるFlageは協調型ソフトウェア・アーキテクチャを記述するためのエージェント指向仕様記述言語である．

本稿の構成は、次の通りである。第2章では、Flageの特徴や構文などについて説明する。第3章では、Flageの形式的な意味論について論じる。第4章では、従来の研究との比較を行う。そして第5章では、まとめと今後の課題の検討を行う。

2　エージェント指向仕様記述言語 Flage

本章では、エージェント指向仕様記述言語 Flageの特徴や構文などについて説明する。

2.1　言語の概要

Flageでは、書換論理[2, 4]などと同様に、系をエージェントの多重集合と捉え、エージェントの挙動を多重集合上の並列書換えとして定式化する。各エージェントは、属性とメソッドの定義を持ち、メソッドの呼びだしは非同期メッセージ交換によって行われる。各エージェントはメタ $0, 1, 2, \ldots$ レベルといった階層構造を持っており、メタ i $(0<i)$ はメタ $i-1$ レベルのすべての定義や状態を属性として保持する。系には場の概念が存在し、エージェントは自分がどういった場に属しているかを認識している。1つのエージェントは(0個以上の)複数の場に属することができ、場への出入りも可能である。メッセージの交換は、同じレベルのエージェント間で、1対1または場へのブロードキャストとして行われる。すべてのエージェントを要素とする特別な場としてノードが存在するので、ノードを通じてすべてのエージェントにメッセージをブロードキャストすることも可能である。本稿では、エージェントのメタ i $(0 \leq i)$ レベルからみた $i+1$ レベルを(相対的に)メタと呼び、逆に $i+1$ レベルからみた i レベルをベースと呼ぶ。

2.2　エージェントの表現

各レベルの個々のエージェントは以下のような項によって表現される。

$$\begin{array}{llll} \{ \;\; aid & | & \langle \textbf{attribute} \rangle & attr, \ldots, attr;; \\ & & \langle \textbf{method} \rangle & mtd, \ldots, mtd;; \\ & & \langle \textbf{spec} \rangle & spec, \ldots, spec;; \\ & & \langle \textbf{field} \rangle & fid, \ldots, fid;; \quad \} \end{array}$$

ここで、*aid* はエージェントの識別子、*attr*は属性名と値の対、*mtd* はメッセージパターンと手続きの対、*spec* は関数の代数的な定義、*fid* はエージェントが属する場の名前である。言語の構文規則を図1に示す。例えば、手続き proc の定義において、<=はメッセージの送信を、!は呼びだし元への値の返却を、=は属性値の変更を表す。ここで、*term-with-atid* は定数として属性名を許した拡張項である。

2.3　メタ構造

各エージェントはメタ $0, 1, 2 \ldots$ といったレベルを持つ。エージェント *aid* のメタは識別子として m(*aid*) を持ち、その属性にベースの記述や状態が保持される。これにより、ベースの記述や状態がメタではデータとして扱えるため、プログラムの柔軟性に関する記述が可能となる。さらに m(*aid*) のメタ m(m(*aid*)) においては、*aid* に関する柔軟性の修正や変更などが記述できる。ベースを変更するための組込み関数として、ベースレベル・アクセスが用意されている。表1にベースレベル・アクセスの一部を示す。

ここで、メタレベルとベースレベルの関係を、例により説明する。まず、図2にメッセージ処理の簡単な例を示す。メタエージェント m(num) は、num に送られてきたメッセージを mque (キュー) に加え、mcnt

```
agent-exp ::= "{" aid "|"
  [ "⟨attribute⟩" attr "," ... "," attr ";;" ]
  [ "⟨method⟩" mtd "," ... "," mtd ";;" ]
  [ "⟨spec⟩" spec "," ... "," spec ";;" ]
  [ "⟨field⟩" fid "," ... "," fid ";;" ]   "}"
aid ::= symbol | "m(" aid ")"
attr ::= atid ":" val
atid ::= symbol
val ::= term
mtd ::= msg-pttn ":" proc
msg-pttn ::= "(" mid [ term...term ] ")"
mid ::= symbol
proc ::= dest "<=" msg
   | "!(" dest "<=" msg ")"
   | atid "=" "(" dest "<=" msg ")"
   | atid "=" term-with-atid
   | "!" term-with-atid
   | proc ";" proc
   | "if" term-with-atid "=" term-with-atid
     "then" "(" proc ")" [ "else" "(" proc ")" ]
dest ::= aid | fid | bool-formla | fid ":" bool-formla
fid ::= symbol
msg ::= "(" mid [ term-with-atid...term-with-atid ] ")"
spec ::= sort-decl | op-decl | var-decl | eqn-decl
sort-decl ::= "sort" sid...sid
op-decl ::= "op" opid...opid ":" sid...sid "->" sid
var-decl ::= "var" variable...variable ":" sid
eqn-decl ::= "eq" term "=" term
```

図 1: Flage の構文規則

(メッセージ数) に 1 を足す。もしメッセージ数が 0 でなければ、関数 sel-msg が現状で最適なメッセージを選択し num 上で実行する。

```
{m(num)|
  <attribute>
    mque:(), mcnt:0, selm:();;
  <method>
    (send-msg I M F num):
       mque=((I M F) mque); mcnt=mcnt+1;
       F<=(receive-msg I F),
    (): if (mcnt=\=0)=true then
      ( selm=sel-msg(mque); mque=del(selm,mque);
        mcnt=mcnt-1; (exc-mtd selm) );;
  <spec>
    sort Que Msg,
    op   sel-msg:Que->Msg,
    var  Q:Que,
    eq   sel-msg(Q)= ... (get-attr ... ) ... ;; }
```

図 2: メッセージ処理の記述例

更にエージェント num のメタ 0,1 は以下のような属性とメソッドを持つとする。

```
{num |
  <attribute> number;;
  <method>
    (add n): number=number+n;;
  ...}

{m(num) |
  <attribute>
    base_cont: [[attribute, [number]],
      [method, [(add n), ...], ...],
    mque, ...;;
  <method>
    (clear_msg): mque=[],
  ...}
```

図 3: Flage の記述例

このとき、メタ 0 においては例えば次のような挙動が考えられる。

```
{num |
  <attribute> number: 3
  ...} <= (add 1)

⟶

{num |
  <attribute> number: 4
  ...}
```

図 4: メタ 0 レベルの挙動の例

すなわち、num はメッセージ (add 1) を受信してメソッドを起動する訳である[†1]。同様にメタ 1 においては、次のような挙動が考えられる。

```
{m(num) |
  <attribute>
    base_cont: [[attribute, [number, 3]],
      [method, [(add n), ...], ...],
    mque: [(add 1)],
    ...;; ...} <= (clear_msg)

⟶

{m(num) |
  <attribute>
    base_cont: [[attribute, [number, 3]],
      [method, [(add n), ...], ...],
    mque: [], ...;; ...}
```

図 5: メタ 1 レベルの挙動の例

ここで、メタ 1 ではメッセージ (clear_msg) を、またメタ 0 ではメッセージ (add 1) を、この順番で受信したとする。すると、次のような 2 通りの挙動が考えられる。

[†1] 正確には、メッセージキューの先頭から取出されて初めて起動される。

- メタ 1 において、先に (clear_msg) に対するメソッドを起動し終わってから[†2]、(add 1) を mque に追加したとする。すると次にはメッセージキューに残っている (add 1) が処理され、それに対するメソッドを起動する。

- 逆にメタ 1 において、(add 1) を mque に追加した後で (clear_msg) に対するメソッドを起動したとすると、その際にメッセージキューは空にされるため、メッセージ (add 1) は消されたことになる。したがって、対応するメソッドは起動されない。

2.4 通信メカニズム

メッセージの送信方法には、相手先の識別子を指定した送信と、場へのブロードキャスト、ノードへのブロードキャストがある。これらのメッセージは、すべてメタ間で受渡しされる。これにより、メッセージに対するメソッドの不備や予期せぬメッセージの存在をメタで検出し、メッセージの処理を拒否したり、ベースのメソッドの生成・適合化を行ったり、メッセージキューを書き換えて他のメッセージ処理を優先したりなどの対処を行うことができる。メタでメッセージを交換したり、エージェントの生成を依頼するためのプロトコルとして、メタ・プロトコルが用意されている。表 2にメタ・プロトコルを示す。

表 1: ベースレベル・アクセスの例

呼びだし形式	意味
(get-attr *atid*)	ベースの属性 *atid* の値を参照
(req-add-attr *atid val*)	属性 *atid* : *val*をベースへ追加
(req-del-attr *atid*)	ベースの属性 *atid* を削除
(req-mod-attr *atid val*)	ベースの属性値を *val*に変更

表 2: メタ・プロトコル

メッセージ	意味
(send-msg *id msg frm to*)	*id*, *msg*を *frm* から *to* へ送信
(reply *id val*)	*id* に対して *val*を返却
(receive-msg *id frm*)	*id* の受理確認を *frm* へ送信
(broadcast *msg fid*)	*msg*を *fid* へブロードキャスト
(gen-agent *aid*)	エージェントのコピーを生成

[†2] Flage ではエージェント内はシングルスレッドを仮定している。

これまでのまとめとして、図 6 に、メタ構造と通信メカニズムの概要を示しておく。

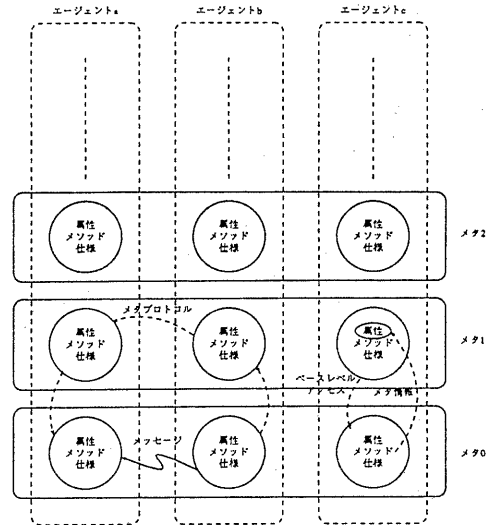

図 6: メタ構造と通信メカニズム

2.5 場

場はエージェントが協調する際の範囲を与える概念で、通常はその名前によって識別される。また、場の概念はメタ構造とは独立のものであり、各メタレベルにおいてそれぞれ固有の場が存在する。しかしその一方で、各エージェントは自分がどのような名前の場に属しているかを知っており、メタは自らこのベースレベルの場の情報を書き換えることで、新しい場に参加したり、場から抜けだすことができる。すべてのエージェントを含む特別な場としてノードが存在し、以下の方法によって、ノード内のエージェントから動的に場を構成できる。*bool-formula*(論理式) で場を指定した場合、その時点で論理式を満足するエージェントに対するブロードキャストができる。また、*fid*: *bool-formula* なる形式で場を指定した場合、その時点で論理式を満足するエージェントから構成される場に *fid* という名前をつけて固定することができる。例えば、ノード内の全エージェントへのブロードキャストは、true<=*msg* と指定すればよい。

なお、ブロードキャストメッセージの処理は、以下のような方式で行われる。

- 各エージェントは、1つのメッセージには1度しかアクセスしない。
- メッセージは全エージェント[†3]がアクセスするまではシステム内に残っているが、全エージェントのアクセスが完了すれば直ちに削除される。

なお、図7に場の概念の概要を示す。

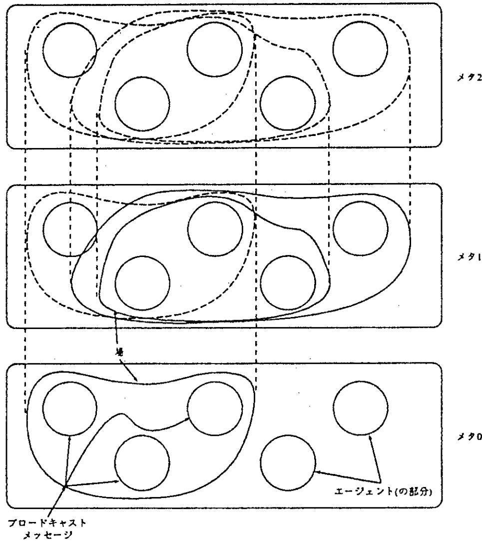

図 7: 場の概念

2.6 仕様の記述と検証

specには関数定義だけでなくプログラムの抽象的な性質も記述できる。Flageでは、任意の抽象レベルで書かれた記述に対して数学的な意味が与えられるので(第3章)、協調メカニズムの正しさを厳密に検証できる。

2.7 Flageの特徴

ここでFlageの特徴を整理しておく。

[†3]場に属しているか否かには関係しない。これは、場への自由な出入りを考慮するためである。

1. マルチ・エージェントである。
2. 無限のメタ構造を持つ。

   メタから見ればベースの記述や動作はデータである。したがって、記述変更や動作変更は容易である。スーパーバイザは仮定せず、各エージェント毎にメタがベースを管理している。メタiの自己モデルをメタi+1に記述できる。
3. メタ・プロトコルを持つ。

   メッセージの受渡しをメタで制御する。この制御のためのプロトコルとしてメタ・プロトコルが用意されている。
4. 場の概念を持つ。

   場の指定方法としては、(a)場の名前で指定、(b)ある条件を満たすものの集まりとして(その都度)指定、(c)ある時点で条件を満たすものに名前をつけて以降は名前で指定、の3方法がある。
5. カテゴリカルな意味論を持つ。

   1つのレベル内の構造が明確になる。さらに、複数レベル間での関係も意味づけされる。つまり、メタを扱うことのできる意味論を持っている。
6. 仕様記述言語である。

   「抽象的な性質が書ける」という意味での仕様記述言語である。
7. ベースレベル・アクセスを持つ。

   ベースの記述や状態を参照・変更するための組込み関数を持つ。

3 Flageの意味論

本章では、Flageの形式的な意味論について論じる。

3.1 Flage構造 – メタ構造の意味論のための枠組

本節では、Flageにおいて扱われるようなメタレベルアーキテクチャの意味論を展開するためのカテゴリ論的枠組である Flage構造 について論ずる。

Flage 構造においては、あるメタレベル階層におけるシステムの挙動の意味を、状態を対象とし、状態遷移を射とするカテゴリにより表現する。すると、各階層間の関係を表現する必要があるが、Flage 構造においてはこれを関手により実現する。その際、注意すべき点として、

- 1 段のみ異る階層間の関係だけ考慮して、そのような関係を表現する関手さえ用意しておけば、それ以上離れた階層間の関係は、1 段毎の関手の合成により表現できる。

- 各階層を表現するカテゴリの射としては、あくまでその階層において許容される状態遷移に対応するもののみを考慮すべきである。したがって、ある階層のカテゴリの射が、それより下の階層には対応する射がない場合も考えられる。そのため、このような対応を表現する関手の定義域は、もとのカテゴリの部分カテゴリとする必要がある。但し、ここでは対象、即ち状態に対しては、下の階層にも必ず対応する状態が存在するものと考えるので、この部分カテゴリは全域的である (即ち対象は全て含む) とすべきである。

- 逆にある階層に属するものに対し、それより上の階層には必ず対応するものが存在すると考えられるので、このような関手は全域的かつ充満であるとすべきである。

以上の考察により、Flage 構造を次のように定義する。

定義 1
以下を満たすような組 $(\{\mathcal{C}_i\}_{i=0,1,\ldots}, \{F_j\}_{j=1,2,\ldots})$ を Flage 構造という。

- $\mathcal{C}_i$ はカテゴリ。

- $F_j : \mathcal{C}'_j \longrightarrow \mathcal{C}_{j-1}$ は関手。但し、

  - $\mathcal{C}'_j$ は $\mathcal{C}_j$ の全域的部分カテゴリ。
  - F_j は全域的。即ち、対象に関しては全射。
  - F_j は充満。即ち、射に関しても全射。

3.2 場とブロードキャストの意味

2 で述べたように、場の概念はメタ構造とはほとんど独立のものである。特に意味論においては、メタ構造をモデル化するためのものである、上述の Flage 構造とは独立して論ずることができる。そこで本節では、場とブロードキャストの意味、特にブロードキャストされたメッセージの処理について説明する。

まず概略としては次の通りである。

- ブロードキャストメッセージは、各エージェントがそのメッセージを (受理するかしないかにかかわらず) 既に処理をすましたかどうかを確認するためのチェックリストが付随しているものと解釈する。そして、初めはチェックリストとして空リストが与えられる。このチェックリストにより、1 つのエージェントが同じメッセージを 2 度処理することがないことを保証する。

- 最終的に、全エージェントがメッセージの処理をすませたならば、そのメッセージはシステムから削除される。

これらのことを定式化するために、ブロードキャストメッセージの処理を次の 2 つのフェーズに分けて意味を与える。

1. 個別チェックフェーズ

   各エージェントが、個別にブロードキャストメッセージを処理するフェーズである。ここではまず、各エージェントは、メッセージに付随したチェックリストを参照する。そして、

   - 自分の id がチェックリストにあれば、何もしない。
   - 自分の id がチェックリストになければ、
     (a) 自分の id をチェックリストに追加 (「署名」) し、
     (b) そのメッセージを処理する。

2. 残務処理フェーズ

   まだメッセージの処理を行っていない全てのエージェントに処理をさせた後、メッセージを削除するフェーズ。

3.3 Flage 構造への翻訳

本節では、Flage の記述に対し、その意味を表わす Flage 構造への翻訳を与える (図 8参照)。

まずメタ i レベル ($i = 0, 1, \ldots$) の階層に対し、カテゴリ C_i を次のように与える。

1. 対象としてのシステム状態記述の集合 S を、次のように帰納的に定義する。
   - エージェント記述の (仕様中の等式に関する) 同値類は S に属する。
   - 任意のメッセージ項は S に属する。但し、メッセージ項とは、次のような形をした項。
     - (send-msg *msg from to*)
     - (reply *term from to*)
   - $t, t' \in S$ ならば、$t\,t' \in S$。但し、この操作は可換かつ結合的で、単位元 ϕ を持つものとする。
2. 射は、次のように定義される。
   - $\forall s \in S.\mathrm{id}_s \in C_i$
   - 各原子的状態遷移に対応する射を含ませる。例えば、
     (send-msg $(m, t_1, \ldots, t_n)ab$)
     $\{b| \ldots \langle\text{method}\rangle \ldots (m, x_1, \ldots x_n)\ :\ c\ \Leftarrow msg \ldots\}$
     $\longrightarrow\ \{b| \ldots \langle\text{method}\rangle \ldots (m, x_1, \ldots x_n)\ :\ c \Leftarrow msg \ldots\}$
     (send-msg $msgbc$)
   - 次に、[3] と同様の射の構成 (congruence, composition など) を、前述のシステム状態に対する結合操作に関して行う。但し、場へのメッセージ送信に対しては、3.2 で述べたような意味になるように congruence を制限する。

次に、各階層間の関係を表す関手 $F_j : C'_j \longrightarrow C_{j-1}$ を、次のように定義する。

- C'_j は次のように定義する。
  - 対象は C_j と同じ。
  - 射は、C_{j-1} の射に対応するものだけをとる。即ち、メタ $j-1$ レベルでのメソッド実行に対応する射、及びそれらから構成される射 (恒等射を含む) である。
- $\mathcal{F}_j$ は、C'_j の要素 (対象及び射) を、対応する C_j の要素に写すものをとる。

定理 2
以上のように定義された $(\{C_i\}_{i=0,1,\ldots}, \{F_j\}_{j=1,2,\ldots})$ は *Flage* 構造となる。

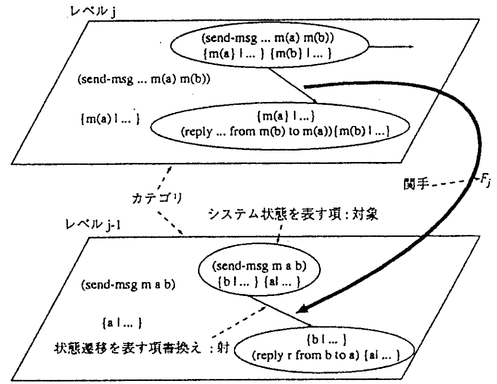

図 8: Flage の意味論

4 従来の研究との比較

本章では、従来の研究との比較を行う。

まず [2, 3, 8] においては、並行システムの記述言語である書換え論理に基づく仕様記述言語 Maude を提案し、更にカテゴリ論的意味論を展開している。そして Maude によるエージェントの協調動作の記述を行っている。しかし、メタ構造や場といった、協調メカニズムの記述に必要な概念を提供していないので、特に記述の容易さや可読性に問題がある。

また Flage のメタ構造と場に類似した概念である group-wide reflection を提供する言語として ABCL/R2[9] がある。しかし ABCL/R2 はどちらかというと OS レベルのリフレクションを目指した言語であり、したがって抽象的な制約の記述を目指した Flage とは方向が異なっている。

5 おわりに

Flage は、協調計算に基づく仕様記述言語である。

エージェントは、環境の変化に柔軟に対処しながら計算を行う。そのために、メタレベル階層を持つことにより、エージェントの挙動の制御を柔軟に記述することを可能にしている。また複数のエージェントからなる場の概念及びブロードキャストの機能を備えることにより、エージェント間の協調的動作が可能である。

エージェント・モデルの記述の観点でまとめるならば, Flage においてエージェントの自律性は, 場への出入りによるメッセージの主体的な選択および, メッセージの受渡しをメタで制御することによるメッセージの主体的な取捨選択によって実現される. 他のエージェントとの協調は場およびメタ・プロトコルによって実現できる. また, 自己モデルの記述とその取り扱いはメタ構造によって可能となる.

さらに, 仕様変更が生じた時（環境の変化）には, 各エージェントにおいて自己モデルとして自己形成プロセス（仕様から自身が作成された過程）を記述することにより, 互いの自己形成プロセスを融合, 再利用することによって環境の変化に追従（仕様変更を満たすプログラムを合成）する動作を記述することができる.

最後に, Flage がどのような適用場面において有効なのかについて述べる. 未知のメッセージがあるエージェントに到着した時に, 他のエージェントからの情報を受けながら, 未知のメッセージに対応すべくメソッドを合成する. さらに, 合成したメソッドはほかのエージェントと矛盾がないことを保証する. Flage はこうした一連の動作に対する仕様に対して明確な意味を与えると同時に, 動作が正しいかどうかの保証も与えることができる.

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] 本位田真一ほか: エージェント指向言語 Flage(1) ~ (5), 第 47 回 情報処理学会 全国大会 (1993).

[2] Meseguer, J.: A Logical Theory of Concurrent Objects, in *Proc. ECOOP/OOPSLA '90* (1990), pp. 101–115.

[3] Meseguer, J.: Conditional rewriting logic as a unified model of Concurrency, Technical Report SRI-CSL-91-05, SRI International (1991)

[4] 大須賀昭彦ほか: 並行オブジェクト系のための代数的記述法, 第 45 回 情報処理学会 全国大会 (1992), pp.4-243 ~ 244.

[5] 本位田真一: 協調型アーキテクチャによるソフトウェアの自動生成, 人工知能学会誌, Vol . 6, No . 2(1991)

[6] 本位田真一: 協調型アーキテクチャの実現へ向けて, bit , Vol . 23, No . 28(1991)

[7] 本位田真一: 協調型ソフトウェアアーキテクチャ, 日本ソフトウェア科学会, チュートリアル資料, 分散人工知能ー協調計算とマルチエージェント・システムー (1992)

[8] José Meseguer, Kokichi Futatsugi, and Timothy Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proc. of the International Symposium on New Models for Software Architecture*, pp. 61–106, November 1992.

[9] A. Yonezawa, S. Matsuoka, and T. Watanabe. Hibrid group reflective architecture for object-oriented concurrent reflective programming. In *Fifth ECOOP*, 1991.

[10] 大須賀 昭彦, 坂井 公, 本位田 真一. *Metis-AS* における代数的仕様の検証手続き. 情報処理学会論文誌, Vol. 34, No. 11, pp. 2242 – 2250, Nov. 1993.

[11] 大須賀 昭彦, 坂井 公, 本位田 真一. 等式論理の帰納的定理を証明する手続き. 信学論, Vol. J-76-D-I, No. 3, pp. 130 – 138, 1993.