ソフトウェア工学 85－8
（1992. 5. 21）

協調エージェントのプロセス代数による定式化「発生型ソフトウェア」の提案

大林 正晴* 本位田 真一**
obayashi@ipa.go.jp honiden@ipa.go.jp

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

従来のオブジェクト指向プログラミングの構造的な特徴には、クラス定義階層構造（継承関係）とインスタンスの相互参照構造の２つの側面がある。前者は、より一般的な構造になることが要求され、後者は、処理すべき問題に固有の構造が反映される。これらは、相反する要求であり、機能拡張や部品の再利用を難しくしている。これらを解決するために、新しい「発生型ソフトウェア」と言う概念を提案する。それぞれの部品の定義を独立させ、自由に組合わせが可能で、しかも、部品を組み立てるソフトウェアプロセスをも内蔵しているようなソフトウェアである。これにより、仕様変更などの環境変化に対応できる柔軟な部品が得られる。プロセス代数を用いて、発生型ソフトウェアの記述を試みた。本稿では、その枠組みについて述べる。

The Cooprative Agent Model Based On Process Algebra Approach Proposed new idea "Software developed in itself"

Masaharu OBAYASHI* Shinichi HONIDEN**
obayashi@ipa.go.jp honiden@ipa.go.jp

Laboratory for New Software Architectures

Information-technology Promotion Agency (IPA)
Shuwa-Shibakoen 3-Chome Bldg. 3-1-38 Shiba-kouen Minato-ku, Tokyo 105, Japan

In object oriented programming, in general, there are two aspects about the structure of modeling; one is inhritance relations in class hierarchy and the other is reference relations among instances. The former structure might be required to be generalized, but the latter might be required to be specialized to specific problems. Because of this conflict, we will have difficulty in making enhancement and reuseing components. As a cooporative agent model, we propose a new idea, "software developed in itself" in this paper. In this method, we could cut loose the dependency of each parts, combine them freely and embed software processes into themselves. Proposed method enables software to be more flexible, more applicable to potential changes of enviroment such as specifications have to be altered. We tried to describe software developed in itself by process algebra.

---

* ㈱管理工学研究所より出向 Also with Kanri Kogaku Ltd.
** ㈱東芝より出向 Also with Toshiba Corp.

1．はじめに

ネットワーク通信プロトコル、開放型の並列分散システム、ユーザインタフェースやマンマシン系、ソフトウェアプロセスなどのように自律的な実行主体が存在し相互に通信し合いながら並行的に動作するシステムを計算機で、つまりソフトウェアで自由に扱えるようにしたいと言う要請が高まっている。また、従来のソフトウェアは、機能的には固定的なものであり修正や変更などに対して必ずしも柔軟なものではない。

筆者らは、生体情報系に着目し環境の変化に適応する仕組みをソフトウェア部品に応用することを目指している。特に、遺伝子を生体の発生や分化、恒常性の維持などの制御プログラムとみなし、その本質的な特質をソフトウェア部品としてモデル化することを試みている［５］。

最終的な目標は、要素間の協調動作の結果として意図した仕事を行わせるようなソフトウェアの新しい仕組みを確立することである。

2．協調エージェント

2．1　従来のオブジェクト指向

まず、従来のオブジェクト指向プログラミングの特徴について、ソフトウェア部品の観点から考察してみることにする。

（クラス定義の階層構造）

オブジェクト指向プログラミングの特徴の１つとして、階層構造を用いたクラス定義を上げることができる。上位クラスの属性やメソッドなどを下位クラスが継承することにより、いわゆる差分プログラミングが可能で、コード量を圧縮する強力な手段になっている。

（インスタンスの振る舞いの構造）

クラス定義の階層構造以外に、インスタンスレベルの構造、つまり、オブジェクト相互参照構造を持っている。具体的には、クラス定義から生成されたオブジェクトがどのようにメッセージをやり取りし、目的の機能を果すか、そのようす（振る舞い）のことである。このインスタンス相互参照構造が、問題解決のためのアプリケーションを反映したものになっている。

一般にオブジェクト指向プログラミングの長所として、つぎの３点を挙げることができる。

①対象モデルを系統的に構成できる

②記述を簡潔にすることができる。

③部品としての再利用の機会が増える。

しかし、このようなオブジェクト指向プログラミングは、ソフトウェア開発における諸問題を根本的に解決するものではない。短所もいくつかある。

クラス定義から生成されるオブジェクトの振る舞いを決定するメッセージの送り先は、そのクラス定義の中に記述されており、普通はインスタンス変数などで表わされたオブジェクトに固定されている。アプリケーションの機能変更や拡張などで、それらを変える為には当然クラス定義を変更しなければならない。

一方、クラス定義は、一般に系統的に作られており上位のクラスは、多くの下位クラスをもつことになる。そのようクラス階層で、上位のクラス定義の変更は多くの下位クラスに影響を及ぼすことになる。もちろん、場合によってはそのような上位クラスの変更が下位クラス全体に有意義なこともあるが、必ずしもそうでないことも多い。

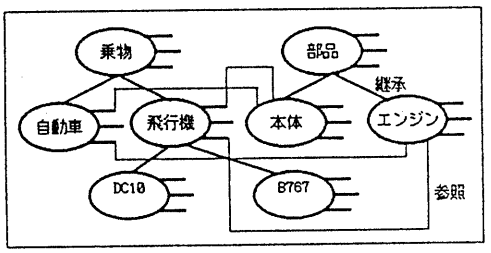

図１．オブジェクト定義の構造例

このようにクラス定義の階層構造が厳密に決められているので変更が難しくなる。これを解決する１つの方法は、クラス定義の階層を深くせずに、なるべく平坦にすることである（差分プログラミングの利点は失われるが）。

クラス定義を部品化の仕組みとしてみたとき、情報隠蔽がなされているので参照するインスタンスの内部構造や実現には影響されず、ある程度自由にシステムの中に他の部品を組込むことが可能である。しかし、実際には、メッセージのインタフェースや機能などを厳格に合せなかればならないなど制約も

多く、再利用を難しくしている。

この問題に対しては、いわゆるリフレクション機構などをもちいてクラス定義を動的に変更して解決しようとする研究も行なわれている。

２．２　協調型システム

協調型システムは、個とよばれる複数の自律的な要素があり、それらの個の間および環境との相互作用を通じて秩序を形成し、特定の機能を果すとともに、環境の変化に対して適用できる能力を備えたものと考える。

また、自律協調（分散）型システムという概念は、従来の工学システムの限界を認識した上で、生物に代用される自然界のシステムからその柔軟性や適応性を学び、その原理を工学システムの設計に反映させることを意図してつくられた目標概念である[7]。また、小林［７］らは、協調型システムとして要請される基本的な要件としてつぎの４つをあげている。

①個の自律性

各個は、自己完結的、すなわち各個はそれぞれの目標、行動プログラム、制御機能をもつこと。

②個間相互作用の非決定性

個間相互作用、すなわち個と個の結合は、事前に設定されることはなく、実際に、ある個がどの個と結合するかは非決定的であること。

③秩序の形成

環境という場が設定されると、個と個および個と環境の相互作用を通じて、システムの大局的目標を反映した秩序が形成されること

④環境変化への適用

集団を構成する個を動的に更新することにより、環境変化への適応が行なえること

ここでは、協調型システムのモデルとしてこれらの要件を満たすものをエージェントモデルとして考えることにする。

２．３　エージェントモデル

まず、ソフトウェアの単位としては、従来のモジュールのようにあらかじめ相互参照関係が定まった固定的なものではなく、可塑的で自由に組合せが可能であるような個体（エージェントと呼ぶ）あるいは、その集合体を考える。

その上で組合せ方によって、その働きが変化するような仕組みを考える。言換えれば、周りの環境（組合せ方）によって、機能が変るような機構が基本的な枠組みになる。計算メカニズムとしては、個体間のアクションの相互作用の結果として、目的の演算が行なわれ、新たな個体集合となるような計算モデルが、その１つの候補として考えられる。

このようなモデルは、従来のオブジェクト指向的な考え方と発想の上では共通する点も多々あるが、プログラミング機構としてはかなり別の形態となるであろう。筆者らは、将来確立されるであろうエージェントモデルの原型を求めて種々の実験を試みたいと考えている。その出発点として、つぎのような仮説のもとに、新しいソフトウェア構造化手法の開発に挑戦している。

（個の自律性）

個々のエージェントは、アクションを並列に起こしながら動作する。従来の実用的なオブジェクト指向言語の多くは、逐次実行を前提にしている（もちろん、並列計算のモデルの研究も多数あるが、まだ、実用にはなっていない）。

また、エージェントの定義階層を平坦なものにして機能の変更の影響が、他の定義に直接及ばない仕組みを考える。類似のものを定義する際には、概念的に複製を作りそれに変更を加える。これにより、部品の独立性が高まり、機能の拡張、変更が行ないやすくなる。この点は、オブジェクト指向の利点である差分プログラミングに逆行するように思われるが、支援系での定義の管理方式を工夫すればある程度解決できる問題だと考えている。

（個体相互作用の非決定性）

アクションの送り先は、エージェント定義の中に固定的にあるのではなく、その送信するアクションを受理できるエージェントなら何れにも特異的に送ることができる。

この点は、従来のオブジェクト指向と大きくことなる点であり、部品の組合わせがより柔軟にできるようになる。もちろん、意味のある組合わせにするには、部品の相互作用を十分把握していなければならない（この点は、新たな複雑さの問題をもたらすことになることが予想される）。

（秩序の形成）

従来のオブジェクト指向では、インスタンスの振る舞いの構造、つまりメッセージの相互参照の構造

もクラス定義の中に記述されていた。しかし、ここで考えているエージェントモデルでは、個々のエージェント定義とは別に、その外でエージェントの組合わせ方を定義でき、それらの相互作用でシステムの振る舞い、つまり、秩序の形成がなされるような仕組みを考える。

（環境変化への適用）

小さな環境（相互作用をおこす周りのエージェント）の変化に対しては、個々のエージェント定義を静的に強化して、適用性を高める。大きな環境（新たな機能追加や変更）の変化に対しては、個々のエージェント定義の変更だけでなく、エージェントを環境に応じて動的に組み替えることで対処する。

従来のオブジェクト指向言語でも、リフレクション機構などをもちいてクラス定義を動的に変更することにより適用性を高めることができる。しかし、前述のようなクラス定義の構造、および、インスタンスの振る舞いの構造を前提にして変更を行なうのは、そうやさしくはない。

3．発生型ソフトウェア

3．1　定義

（1）ソフトウェアプロセス

一般的には、ソフトウェア開発の諸過程をソフトウェアプロセスと呼び、種々の研究がなされている。

現実のソフトウェアプロセスは、採用する方法論や開発チームなどの人間的要素を含んだものであり複雑である。特に、各プロセスは、単純に進行するのではなく、バックトラックなども頻繁に起る。

ここでは、エージェントモデルのためのシステム構築の論理、つまり、エージェントの組合わせの論理過程のことをソフトウェアプロセスと定義する。

具体的には、最終成果物（エージェントの集合）を得るまでに起った中間の成果物の変化のみに着目し、バックトラックなどの過程は基本的には含まれないものとする。

（2）発生型ソフトウェア

前節のエージェントモデルの議論の中の秩序の形成と環境変化への適用を実現するために、新しい考え方「発生型ソフトウェア」を提案する。すなわち、システムの組み立ての論理（ソフトウェアプロセス）をも内蔵したソフトウェアのことを発生型ソフトウェアと呼ぶことにする。

我々のエージェントモデルでは、エージェント相互の参照構造を順次、組み立てるような論理を自らのエージェント定義の集合の中に含むような体系を考える。つまり、個々のエージェント定義の中ではなく、別にエージェントの組合わせ方を制御するプログラムが、同じエージェントとして定義されているのである。

3．2　環境と世代

つぎに、発生型ソフトウェアにおける環境と世代の概念について説明する。ソフトウェアプロセス（ＳＰと略す）と対象とする問題領域を表わすアプリケーション（ＡＰと略す）とを区別して考える。

具現化されているエージェントの集合を環境と呼ぶことにする。また、発生したソフトウェアを初期環境ごとに世代と呼ぶ。

環境には、ＡＰを処理するエージェントだけでなく、ＳＰに関与するエージェントが共存する。これらの関係を示したものが図２である。ＳＰとＡＰの共通部分にあるエージェントは、各ＡＰに固有のＳＰを表わすエージェントである。

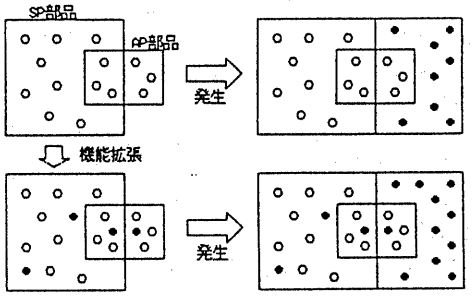

図２．発生型ソフトウェアの世代

発生型ソフトウェアは、環境内のエージェント定義に従ってＡＰを段階的に形成して行く、環境を変化させることにより、環境に適応した類似のシステムを発生することができる。異なる初期環境のもとで発生したソフトウェアは、互いに世代が異なる。同じ初期環境のもとで発生したソフトウェアは同じ世代に属する。

ＳＰの機能強化やＡＰの機能強化など世代の変更は、人が介入して新たなエージェント定義を環境に追加したり、修正して行なう。

３．３　部品

１つのエージェントの定義は、１つのソフトウェア部品とみなすことができる。初期環境には、システムを発生し、ＡＰを解くために必要な部品の集合が収められている。発生の過程で、組み立てられた部品が環境に追加され、高度な機能をもつエージェントとして振る舞う。

部品を分類すると、つぎのようになる。

①ＳＰ部品

部品を組み立てるための汎用的な部品

②ＡＰ固有のＳＰ部品

部品を組み立てるためのＡＰ固有の部品

③ＡＰ部品

ＡＰを解くための部品

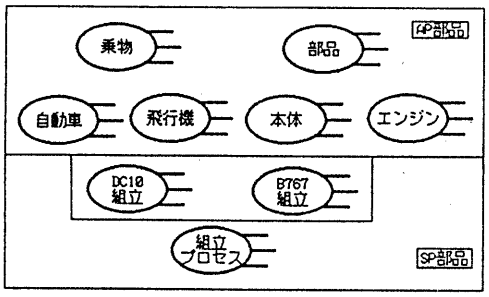

図３．発生型ソフトウェアの構造例

４．計算モデル

４．１　プロセス代数

並列プログラムの分野では、ペトリネット、並列オートマトン、データフローネットワークなどのモデルが抽象マシンとして研究されてきた。しかし、これらは、複雑なシステムを記述するには表現力に乏しいという欠点をもっていた。一方、プロセス代数のように、より表現力を備えた計算モデルが分散システムの振る舞いを記述するために提案され、注目を集めている。プロセス代数は、

①システムの状態をモデル化した動作式

②動作式間の遷移関係を定義する遷移規則

③等価性によって定まるモデルとしての代数

の３つの基本的な枠組みからなる。

これまで提案されているプロセス代数を分類すると、大きくつぎの３つの流れがある。

・ＣＣＳ（Calculus of Communicating Systems）

・ＣＳＰ（Communicating Sequential Processes）

・ＡＣＰ（Algebra of Communicating Processes）

プロセス代数は、単純な仕組みであるが、形式性と汎用性をもつ強力な計算モデルである。特に、現実世界におけるさまざまな事象を並行に動作するプロセス（ａｇｅｎｔと呼ぶ）としてモデル化し、形式的な意味モデルをもつことから、基本的な性質の検証や分析などに有力な手段となる。

４．２　化学抽象マシン

化学抽象マシン（Chemical Abstract Machine）[11]は、化学溶液の中で浮遊している分子同士が反応規則に従って相互作用を起こし、システムの状態を変化させることにより計算を行なう様子をモデル化したものでありプロセス代数ＣＣＳを拡張したものになっている。

化学抽象マシンでは、分子を代数の項として表現し、相互作用を可能にするものをイオン（ＣＣＳのアクション）と呼んでいる。溶液は、熱することにより複雑な分子をより小さい分子に、最後はイオンまで分解することができる。逆に、溶液を冷やすことにより要素部品から合成された重い分子を組み立てなおすことができる。

さらに、抽象的で階層的なプログラミングができるようにするため、膜で囲まれた部分溶液を含むような分子を導入している。また、カプセル化した溶液間やその外部環境との通信を許すための穴をもつことができるようになっている。

化学抽象マシンは、単純な構造規則に従い、各固有のマシンは、もとになる分子から新しい分子をどのように生成するかを規定する単純な規則を追加することにより定義される。

化学抽象マシンの構文規則と意味定義は、つぎの通りである。

①　記号

$N=\{a,b,\ldots\}$ を名前の集合、$L=\{a,\bar{a} \mid a \in N\}$ をN上のラベルの集合とする。$\bar{a}$は、aの補名前を表わす。

化学抽象マシンは、分子、溶液、変換規則から定義される。

m, m', ...で分子（molecules）を表わす。

S, S', ...で溶液（solutions）を表わす。

$m1, m2, \ldots, mk \rightarrow m1', m2', \ldots, ml'$ で変換規則を表わす。

分子は、代数の項であり、各化学抽象マシンに固有

の操作をもつ。
溶液は、分子の有限集合で【m1,m2,...,mk】のように書く。

② 推論規則
・反応法則
m1,m2,...,mk → m1',m2',...,m1'
具体的な溶液に対しては、
【M1,M2,...,Mk】→【M1',M2',...,M1'】
となる。Miは、miのインスタンスである。
・化学法則 $\dfrac{S \to S'}{S \cup S'' \to S' \cup S''}$
・膜法則 $\dfrac{S \to S'}{【C[S]】 \to 【C[S']】}$
ただし、C[]は、コンテキストを表わす。
・空気ロック法則
【m】∪ S ⇔【m⊿S】

③ TCCS化学抽象マシン[11]

TCCS計算モデルの化学抽象マシンによる形式化の概要はつぎの通りである。

(構文規則)
・エージェント (agent)
p::=0 | α.p | (p | q) | p＼a | p[φ] | p+q |
p[]q | $fix_i(x_i=p_i)$
・分子 (molecules)
m::=p | α.m | m＼a | m[φ] | S | m⊿S | <m,m> |
l:<m,m> | r:<m,m>

(遷移規則)

並行	p | q ⇔ P, q
反応	α.m, $\overline{\alpha}$.n → m, n
制限膜	m＼a ⇔ 【m】＼a
制限ｲｵﾝ	(α.m)＼a ⇔ $\overline{\alpha}$.(m＼a) a∉{α,$\overline{\alpha}$}
名前換え膜	m[φ] ⇔ 【m】[φ]
名前換えｲｵﾝ	(α.m)[φ] ⇔ φ(α).(m[φ])
左内部和	p+q → p
右内部和	p+q → q
外部和	p[]q ⇔ <【p】,【q】>
左外部和ｲｵﾝ	<【α.m】,S> ⇀ α.l:<m,S>
右外部和ｲｵﾝ	<S,【α.m】> ⇀ α.r:<S,m>
左射影	l:<m,m'> ⇀ m
右射影	r:<m,m'> ⇀ m'
不動点	$fix_i(x_i=p_i)$ ⇀ $p_i[fix_i(x_i=p_i)/x_i]$
清掃	0 ⇀
	【】＼a ⇀
	【】[φ] ⇀

(溶液の遷移例)

	【a.0 | ($\overline{a}$.p | q)＼b】
⇎	【a.0,【$\overline{a}$.p,q】＼b】
⇀	【a.0,【($\overline{a}$.p)⊿【q】】＼b】
⇀	【a.0,【$\overline{a}$.(p⊿【q】)】＼b】
↓	【a.0,($\overline{a}$.(p⊿【q】))＼b】
⇀	【a.0,$\overline{a}$.((p⊿【q】)＼b)】
→	【0,(p⊿【q】)＼b】
⇀	【【p⊿【q】】＼b】
↓	【【p,q】＼b】

4．3 発生型ソフトウェアの定式化
発生型ソフトウェアを定式化するための計算モデルとして、最初の試みとしてCCSの1つである化学抽象マシンを採用した。その理由は、つぎの通りである。

(1) 記述の簡潔さ
これまでの並行計算モデルは、ポートやチャンネルなどの概念的なアーキテクチャを基礎にしたものが多かった。これらは並列動作のネット構造を正確に伝えるものであるが、並列の動作を厳格にプログラミングすることは、逐次プログラミングに比べて非常に難しい。
一方、CCSは、並列システムの複雑な制御の詳細をあまり気にせずに、その振る舞いを書くことができる。特に、並行して動作するエージェントを個々に独立して定義できる。

(2) エージェントモデルとしての要件
CCSは、前述のエージェントモデルの要件、
①個の自律性
②個間相互作用の非決定性
を満たしている。また、要件
③秩序の形成
④環境変化への適用
は、発生型ソフトウェアの概念を実現することにより満たされる。
特に、SP部品とAP部品を同じCCSを用いて記述することにより、いわゆるリフレクションと同

様の機能を表現できる。これにより発生型ソフトウェアの部品化の枠組みをＣＣＳで定式化できると考えている。

(3) 記述能力

化学抽象マシンは、膜の概念を導入したことで能力が強化されている。従来のＣＣＳの能力をもつ化学抽象マシンや、並列化されたラムダ計算として振る舞う化学抽象マシンを構築することがこの膜の概念を用いてできる。

５．発生型ソフトウェアの記述例

発生型ソフトウェアを化学抽象マシンで記述することを試みた。その概要を簡単な例[10]で説明する。

（ＡＰ部品）

自然数を状態としてもつようなエージェントを考え、その数を１だけ増やしたり、１だけ減らしたりする機能をもつ部品カウンタがＡＰ部品としてつぎのように定義されているとする。

fix (C = inc.(C^C) + dec.D)
fix (D = $\overline{d}$.C + $\overline{z}$.B)
fix (B = inc.(C^B) + z.B)

ただし、

P^Q = (P[i'/i, z'/z, d'/d] |
Q[i'/inc, z'/zero, d'/dec])＼{i', z', d'}

とする。

このカウンタの構造をもとに、類似する部品スタックを発生させることを考える。期待されるスタックのエージェント定義は、つぎのようになる。

fix (CC(x) = push(y).(CC(y)^CC(x))
+ $\overline{pop}$(x).DD)
fix (DD = o(x).CC(x) + $\overline{e}$.BB)
fix (BB = push(y).(CC(y)^BB) + empty.BB)

（ＳＰ部品）

ＳＰ部品としては単純なつぎのようなエージェントを考えることにする。これは、部品を修正したり、組み立てたりする機能を持つ。なお、r(a), w(b)は、a, $\overline{b}$を意味するものとする。

fix (SP = Modify | Combine)
fix (Modify = Σr(x).M《x》) x∈E
fix (Combine = Prefix | Inaction |
Summation | Composition |
Restriction | Relabeling |
Recursion)

ここで、エージェントの構文要素に対応した変数をa, E, F, L, f, Xで表わすことにする。

fix (Prefix(a, E) = a.E)
fix (Inaction = 0)
fix (Summation(E, F) = E + F)
fix (Composition(E, F) = E | F)
fix (Restriction(E, L) = E＼L)
fix (Relabeling(E, f) = E[f])
fix (Recursion(X, E) = fix(X = E))

また、エージェント定義を具体的なエージェントに翻訳する関数Mは、つぎのように定義される。

M《α.E》 = Prefix(modify(α), M《E》)
M《0》 = Inaction
M《E + F》 = Summation(M《E》, M《F》)
M《E | F》 = Composition(M《E》, M《F》)
M《E ＼ L》 = Restriction(M《E》, M《L》)
M《E[f]》 = Relabeling(M《E》, M《f》)
M《fix(X=E)》 = Recursion(M《X》, M《E》)
M《X》 = modify(X)
M《f》 = modify(f)
M《L》 = modify(L)
modify(x) = w(x).r(y).0

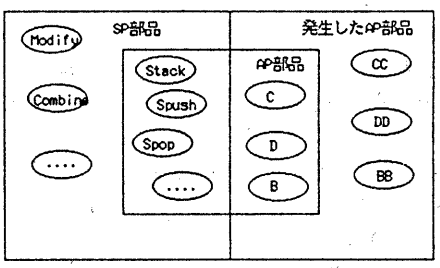

図４．スタックの発生例

（ＡＰ固有のＳＰ部品）

最後に、ＡＰ固有のＳＰ部品として、カウンタからスタックを組み立てるために必要なＡＰ固有の部品をつぎのように定義する。

fix (Stack = r(C).w(CC(x)).(Spush |
Spop | Sempty))
fix (Spush = r(inc).w(push(x)).0)
fix (Spop = r(dec).w(pop(x)).0)

```
fix ( Sempty = r(z).w(empty).0 )
fix ( Sd = r(d).w(o(x)).0 )
fix ( Sz = r(z).w(e).0 )
```

これらの関係を図示すると、図４のようになる。カウンタの定義がＡＰ部品に、スタックを組み立てるエージェントの定義がＳＰ部品に、それぞれ最初の環境として与えられている。ＳＰ部品Modifyは、カウンタの定義式を受取り、それを構文規則に従って関数Ｍで分解する。分解された構文要素ごとに、スタックを組み立てるためのＳＰ部品Spush, Spopなどと反応して、すなわち、カウンタ定義中の名前などを置換した部品スタックがCombineによって組み立てられる。

ここでの記述は、まだ、モデルも単純であり、記述も完全なものではないが、今後、発生型ソフトウェアの基本的な枠組みを追求する手がかりとなるものであると考えている。

６．おわりに

今回の記述実験で、プロセス代数は、発生型ソフトウェアのようなエージェントが並行して行動するような複雑なモデルの記述に適用できる見通しを得た。

一般に何をエージェントとするかは、様々な議論があるが、プロセス代数では、いわゆるプロセスの各状態を識別できなければならない。つまり、それぞれの状態に対応して個別のエージェントを割り振る。しかし、同じプロセスは共通した振る舞いをすることが多い。同じプロセスの各エージェントを添え字を用いて識別し同じ式でまとめて記述できる点がＣＣＳの強力な点である。

状態の数が増えれば、エージェントをいかに名前付けするかによって、動作式が簡潔に書けるか決まる。当然、問題が複雑になれば、動作式も複雑なものになる。

発生型ソフトウェアを従来の手続き的な記述や関数的な記述でも表現できるであろう。しかし、プロセス内部の状態変化を他のプロセスとの同期を気にしながら記述するのは大変難しい。

また、発生型ソフトウェアは、エージェントの組合わせによって、機能を追加したり変更することが容易であり、従来のオブジェクト指向に比べてより柔軟な仕組みと言える。これらの特徴は、ＳＰとＡＰを同じプロセス代数という枠組みで記述したことで実現しやすくなっている。動的にエージェントの定義を変えるような、いわゆるリフレクションと同様のことができるからである。

今後は、より精密なモデルを作り定式化する必要があり、また、動作の正しさをどのように確認し保証するかなど、解決しなければならない問題も多い。

［謝辞］
本研究は、次世代産業基盤技術開発「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］中村運「細胞進化」培風館
［２］五條堀、他「生命科学が情報科学に期待するもの」情報処理、vol. 31, No. 7, pp. 904-905 (1990)
［３］塩川光一郎「分子発生学」東京大学出版会
［４］本位田真一「協調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, pp184-186 (1991)
［５］大林正晴、本位田真一「生体情報系における協調システムについて」情報処理学会第４３回全国大会、3K-9、1991
［６］阿形清和「多細胞生物の発生」計測と制御、vol. 29, No. 10 (1990)
［７］小林重信、山村雅幸「遺伝ｱﾙｺﾞﾘｽﾞﾑと自律分散ｼｽﾃﾑ」文部省科学研究　第２回　重点領域研究「自律分散ｼｽﾃﾑ」全体講演会論文集、平成４年１月
［８］二木厚吉、富樫敦「形式仕様とﾌﾟﾛｾｽ代数」bit, Vol. 23, No. 11, pp. 11-26, 1991.
［９］富樫敦「ﾌﾟﾛｾｽ代数等価性（前、中、後編）」bit, Vol. 23, 24, No. 12, 13, 1, 1991-19992.
[10] R. Milner. Communication and cincurrency. Prentice Hall, 1989.
[11] Berry, G. and Boudol, G., The Chemical Abstract Machine. POPL 1990.
[12] 本田耕平、所真理雄「非同期通信意味論について」ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ－言語・基礎・実践－研究報告、情処研報 Vol. 91, No. 99