プログラミング 11−3
(1997. 1. 23)

並行オブジェクト指向言語 COOL の開発

金村 星吉　　上田 賀一
kaneyan@cis.ibaraki.ac.jp　　ueda@cis.ibaraki.ac.jp

茨城大学 工学部 情報工学科
茨城県日立市中成沢町 4-12-1

あらまし

オブジェクト指向方法論に基づいて分析／設計された対象世界には多くの能動的オブジェクトが含まれる．しかし，逐次型オブジェクト指向プログラミング言語でこれらを表現しようとすると，分析／設計モデルとは対応の取れない表現となってしまう．また，並行型言語では，記述そのものが困難である．そこでオブジェクトをより自然に記述するために，並行オブジェクト指向インタプリタ言語 COOL を開発した．COOL は永続的操作を持つオブジェクトを明確に記述でき，メッセージの送信形態を気にすることなく記述できる．また，用途に応じたカスタマイズ可能なメッセージ通信機構を持っており，柔軟な記述性を提供した言語である．

キーワード　オブジェクト指向プログラミング言語，メッセージ通信，並行性，インタプリタ

Development of COOL : Concurrent Object-oriented Programming Language

Seikichi Kanemura　　Yoshikazu Ueda
kaneyan@cis.ibaraki.ac.jp　　ueda@cis.ibaraki.ac.jp

Department of Computer and Information Sciences,
Faculty of Engineering, Ibaraki University
4-12-1 Nakanarusawa,Hitachi,Ibaraki,316 JAPAN

Abstract

There are many active objects are in analysis models and design ones based on the object oriented methodology. When these objects are represented in sequential object oriented programming languages (OOPLs), their representations don't correspond to the analysis models and the design ones. And in parallel OOPLs, the representation itself is very difficult. So we developed the concurrent OOPL named COOL in order to represent the objects more naturally. COOL has the flexible presentation features. Namely, using COOL, we can represent the persistent operations of the object definitely and the message communication style easily. And we can also customize the message communication mechanism.

key words　Object oriented programming language, Message communication, Concurrency, Interpreter

1 まえがき

現実世界をモデル化するための手法として，様々な方法論が提案されてきた．オブジェクト指向方法論もこれらの一つである．この手法の特徴は，モデル化対象を**もの**(または，それらの集合)と捉えて，モデル化する手法である．しかし，実際に**もの**をモデル化する際に，全てを受動的な**物**でモデリングするよりは，能動的な**者**を用いたほうが，より自然といえる [2]．しかし逐次言語に写像した時点で，全てのモデルコンポーネントを受動的な**物**へと写像する必要があり，この作業はかなり技巧的作業になる．また，このような技巧的作業を経て作成されたオブジェクトは必ずしも再利用性が高いとは言い難いものである．なぜなら，このような写像によって作成されるコンポーネントは使用されるシステムと強い依存関係を持つようになるからである．

図1(a)において，現状におけるおおまかな開発工程であり，それに対して(b)は理想に一歩近づけた開発工程と成果物について示してある．

この図において言えることは，対象をモデル化した際に，如何にその特性を損なうことなしに実装にまで，導いていけるかが重要であるとわかる．対象と成果物の間のギャップが大きければ大きい程，成果物生成までの道のりは遠く，また，その再利用性は下がっていく．

この問題を解決するためのアプローチとして，以下に挙げるような改善が考えられる．

- モデル化方法論の改善
- 成果物生成のためのプロセスの改善
- 外部制約の軽減

今回は，最後のアプローチを採ることにより，より自然なモデリングと高度な再利用性を得ることを目標に，並行オブジェクト指向言語 COOL を開発した．COOL は，プロトタイププログラムと再利用性の高いオブジェクトを記述するために開発されたインタプリタ言語である．

2 並行オブジェクト指向言語COOL

COOL の構成要素は全てオブジェクトであり，そのオブジェクト間のメッセージ通信によってシステムの振る舞いを記述する．COOL の特徴は，クラス

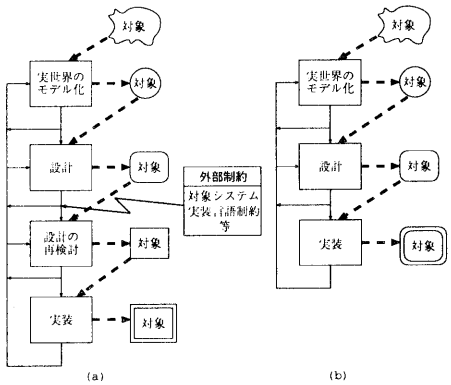

図 1: どのように，改善されるか？

ベースのオブジェクト指向プログラミング言語 (以下 OOPL) である．また，Smalltalk[1] と同程度に弱く型付けされた言語である．ここでは，COOL でのオブジェクトの特徴について述べる．

2.1 オブジェクトの並行性

逐次 OOPL では，対象モデル内のエージェントの並行性を表現するのは困難である．また，並行 OOPL を用いる場合にも，大半の並行言語は既存の逐次言語を並行化したものがほとんどであるために，技巧的なプログラムテクニックをプログラマに要求することに変わりはない．さらに，それらほとんどがコンパイラであったために，プラットホームに強く依存してしまうという問題も生じる．

本稿において開発した COOL はインタプリタであるために，この言語で作成したアプリケーションはプラットホームに依存せず，また，COOL 自体もプラットホームに依存した機能を用いて作成されていないので，移植性は極めて高い．なぜなら，COOL のプロセス (スレッド) 管理は，OS に任せるのではなく，COOL 自体が管理し，それによって言語内の並行性を確保しているからである．ただ，COOL における並行性はあくまでもソフトウェアによって疑似的に発生させているもので，ある瞬間において動作しているオブジェクトは必ず一つである．しかし，ユーザはそれについて考える必要はあまり無く，COOL が自動的にそれぞれのオブジェクトにプロセスの割り当て/解放を行ってくれる．

これにより言語自体の移植性の向上，およびオブジェクトの一元管理といった利点が見込まれる．

COOLにおける並行性は，アプリケーションのパフォーマンスを向上させるためのものではなく，あくまでもより自然にオブジェクトやエージェントをモデリングできるようなプロトタイプ環境を提供するために開発されたものである．

2.2 オブジェクトの拡張

逐次OOPLにおいて，メンバ関数内で長い処理や永久ループなどを作成すると，レスポンスの低下や，他のオブジェクトへのプロセスの明け渡しが行われなくなり，プログラムが正常に動作しなくなることがある．一方，並行OOPLにおいても，「オブジェクトの多重化」＝「メンバ関数の多重化」(ここでのメンバ関数の多重化とは，あるオブジェクト内のメンバ関数を並行化するという意味だけではなく，複数のオブジェクトのメンバ関数が並行化している状況に対しても指している) と見なしているものが多い．

それに対してCOOLでは，「オブジェクトの多重化」＝「オブジェクトボディの多重化」という観点に立っている．オブジェクトボディとは，各オブジェクトが一つだけ持てるメンバ関数の一種であり，このメンバ関数に，オブジェクトが永続的に行う処理を記述する．

一般的にOOPLにおいてオブジェクトは「データ＋メンバ関数」と表すことができ，そしてそれ以上でもそれ以下でも決してない．

本言語におけるオブジェクトは従来のものの定義に加えて，メッセージを受け取る部分として**メッセージレシーバ**(以下レシーバ)と，それを適切な処理に振り分ける部分として**メッセージハンドラ**(以下ハンドラ)，そしてオブジェクトが永続的に行なう作業を表す部分として**オブジェクトボディ**(以下ボディ)をまとめたものとする．

2.2.1 オブジェクトボディ

オブジェクトは，永続的な作業を行うためのメンバとして，必ず一つのボディを持つ．同一オブジェクトのメンバ関数内部で，ロック(COOLでは，非同期メッセージ送信を行うと，その結果が必要になった時点でメンバ関数をロックする) が行なわれない限り，メンバ関数とボディは並行に実行される．ただし，メンバ関数は多重化されない(厳密にはメンバ関数は多重化されないがメンバ関数内部は多重化される．この議論については次項で述べる)．

ボディは，ユーザが明示的に記述することが望まれるが，記述されないときには，デフォルトで実行停止がプロセスに割り当てられる(この状態のオブジェクトは受動的オブジェクトといえる)．

ボディは，他のメンバ関数と並行に実行可能であるという性質と，永続的にプロセスが割り当てられているという点から，オブジェクトの制約チェック等にも適した機構になっている．

2.2.2 メッセージレシーバ

オブジェクトがメッセージを受信した時に，起動されるメンバ関数である．ここでは，受信したメッセージをメッセージキューに格納するという作業を行う．

2.2.3 メッセージハンドラ

メッセージキューからメッセージを取り出し，対応するメンバ関数(または，それらを組み合わせたもの) を起動するという一連の制御をこのメンバ関数を用いて行う．また，他のオブジェクトの返答要求に答えるのもこのオブジェクトが行う．

2.3 システムオブジェクト

システムオブジェクトとは，あらかじめ定義されているオブジェクトのことである．COOLが言語として最低限，振舞うために必要なオブジェクトの総称である．システムオブジェクトはユーザの手によって，変更されことを許していない．また，ユーザ定義のクラスと異なる点は，このオブジェクトに対するメッセージ送信は，ハンドラ，レシーバを介さず直接メンバ関数が起動されるという点である．

2.3.1 object

全てのオブジェクトのスーパクラスである．ここに既定義メンバが登録されている．ハンドラ，レシーバ，ボディのデフォルトもこのオブジェクトが持っている．

2.3.2 string

文字列を表現するためのクラスオブジェクトである．このオブジェクトは，文字列の連結や数への変換などのメッセージを受信可能である．

2.3.3 number

数を表すためのオブジェクトである．このオブジェクトは，四則演算や文字列への変換などのメッセージを受信可能である．

2.3.4 block

ブロックは，リストを表現するための手段である．要素間は';'を用いて区切る．ただし，このブロックは評価することが可能であり，実際，COOLにおけるレシーバ，ハンドラ，コンストラクタも含めた全てのメンバ関数は，ブロックと等価である．このオブジェクトは，様々なリスト操作及び，評価，ループといったメッセージを受信可能である．また，評価対象のブロック全ては，ローカルメンバを持つことを許している．

```
[[a;b;c];
    a       := 3;
    b [a]   := (a + 3) * 4;
    c [a;b] := (a - b) / 2;
]
```

上記のような例において，[a;b;c]はローカルメンバ宣言部であり，a,b,cがローカルメンバとなる．

2.3.5 true, false

真偽値を表すためのオブジェクトである．論理計算及び条件分岐などの処理は，このオブジェクトに対するメッセージにより，処理される．

2.3.6 null

全てのメッセージを受け付けないオブジェクトである．

2.4 既定義メンバ

ここでは，オブジェクトを生成した際に，全てのオブジェクトがデフォルトで持っているメンバについて述べる．前述のボディ，レシーバ，ハンドラも既定義メンバである．

2.4.1 self

自分自身を指すための既定義メンバである．また，selfに対して代入を行うことによって，そのブロックの評価値とすることができる．

2.4.2 super

スーパクラスを表すための既定義メンバである．このデフォルトは，全ての親クラスを指しているが，代入することによって，親クラスを指定することができる．ただし，代入するクラスは必ず最初の親クラス集合に含まれていなければならない．また，デフォルトに戻すには，nullを代入することで行える．

2.4.3 messageQueue

到着したメッセージを格納するためのブロックオブジェクトである．このメンバに対する特別な制約はなく，プログラマの目的によって重みを付けたり，スタックにするといったことが可能である．このカスタマイズは，レシーバ，ハンドラの書き換えによって行える．

2.4.4 new

オブジェクトを生成するためのコンストラクタである．このメンバは，普通はクラスオブジェクトからオブジェクトを生成した時点で継承されないが，クラスの定義次第で継承可能である．このメンバを継承したオブジェクトは，新しいオブジェクトを生成できるという点で，クラスオブジェクトとしての振舞いが可能となる．ただし，クラスオブジェクトは全てのオブジェクト内でグローバルにクラス名で参照可能であるのに対して，前記のオブジェクトはそのようなグローバルには参照されないという点で異なる．またクラスオブジェクトは，オブジェクトボディがアクティブでないという点においても異なる．

3 COOLのメッセージ送信

COOLにおいて，メッセージ送信は最も重要な要素の一つである．全ての構成要素がオブジェクトであるCOOLにおいて，ある事象を記述するには，オブジェクトにメッセージを送信するという手段が，唯一つの方法である．

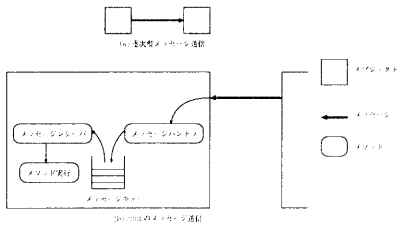

図 2: メッセージ送信の比較

COOLは，オブジェクトに対していくつかのプロセスを割り当て，またそのプロセスはいくつかのスレッドに分割されて並行に実行される．

3.1 動的メッセージ受渡し機構

従来の逐次OOPLにおいては，メッセージとメンバ関数が同意に扱われているものが多い．また，並行OOPLにおいても，ほとんどがメッセージキューやメッセージハンドラを持っているが，大抵はシステム内に隠蔽されており，変更は困難となっている．

OOPLでは，オブジェクトに対するメッセージが処理の流れになる．よって，OOPLではメッセージ送受信は重要な要素であるといえる．COOLでは，より柔軟なメッセージ送受信を実現するために，メッセージレシーバ，メッセージハンドラという機構を備えている (図2参照)．これらは全て，変更可能であるが，明示的に定義しなければ，クラス object から継承してきたものを使用する．

3.2 メッセージ送信形態

COOLがサポートしているメッセージタイプは過去型，現在型，未来型のいずれかであり，自動的に最適なメッセージ送信形態を割り当てる．

3.2.1 現在型

現在型メッセージ送信は，メッセージ連鎖 (あるメッセージ送信の結果に対してメッセージを送信する形態) の中で発生するメッセージ送信に対して用いられる．例えば，1というオブジェクトに"+"というメッセージとともに，パラメータ2というオブジェクトを渡し，その結果を4倍し，その結果を出力したいというメッセージ連鎖は次のように記述する．

```
(( 1 + 2 ) * 4) println;
```

上記の記述で発生する三つのメッセージはそれぞれ同期を取りながら通信を行う．また，メッセージ送信順序は例のように括弧を用いて明示的に示す必要がある．

あるメッセージ連鎖に注目したとき，その処理は逐次処理となるので，COOLはメッセージ連鎖一つに対して一つのスレッドを割り当てる．

3.2.2 過去型

過去型メッセージ送信は，メッセージ送信結果が必要でないときに用いられる．例えば，次に挙げる二つのメッセージ連鎖は非同期に実行される．

```
(``abc'' + ``def'') println;
(1 + 2) println;
```

3.2.3 未来型

未来型メッセージ送信は，メッセージ送信結果を代入する必要があるときに使われる．同期が必要になるのは，オブジェクトの状態 (メンバの指す値) が変更されるときである．これが起こり得るのは，代入操作のときのみである．

```
a := (1 + 2) * b;
c := (3 + 4) * c;
[a;c] := (a + c) println;
```

上記の例では，1行目と2行目は非同期に実行する．ただし，a と c にはそれぞれロックが掛けられる．そして，3行目で1，2行目が終了するまで同期待ちとなり，ロックが外れた時点で3行目が実行される ([...] を同期ブロックと呼ぶ)．また，3行目は，次のように，代入も併せて記述可能である．

```
c [a;c] := (a + c) println;
```

このように，ロックは代入作業が終った時点で，自動的に外れるが，ロック状態を継続したいときには，次のように記述する．

```
queue := queue insert 0 two;
queue[*] := queue insert 0 one;
```

上記の作業は，二つの要素 (one, two) を queue に追加するための記述であるが，この二つの作業中に queue の状態が変わらないことが，この記述によって保証される．もし，2行目の [*] が存在しないと

きには，queue の先頭が one，two という順番になるという保証はされなくなる．

このように COOL の代入は，表記上では Call/Return Style[3] をとっている．しかし，実際の振舞いにおいては COOL が暗黙の内に Customer-Passing Style[3] と同等の振舞いになるように，メッセージ送信形態を選択してくれる．

3.3 自分自身に対するメッセージ送信

自分自身に対するメッセージ送信が存在するときには，無条件に同期を取る．他のメッセージ連鎖が非同期に実行されていても，その時点でそのプロセス内の全てのスレッドを一時停止させる．以下に例を示す．

```
(a + b) println;
((1 + 2) + (self mes)) println;
(3 + 4) println;
```

ここで，最初は 1~3 行目は非同期に実行するが，self mes というメッセージが送信された時点で，2，3行目のスレッドの実行を停止させ，1行目のスレッドが終了するのを待つ．そして，1行目のスレッドが終了した時点で，2行目のスレッドが再開される．また，次のように明示的に記述することにより，実行した時点で全てのスレッドを停止させることも可能である．

```
(a + b) print;
[self] := (1 + (self mes)) print;
(3 + 4) print;
```

このように，同期ブロックに self というエントリが存在するときには，完全に現在型でメッセージが送信される．

このように複雑な機構を持つ理由として，自分自身に対するメッセージ送信によるデッドロックの回避のためである．すなわち，ロックを掛けている対象が少ないほど，デッドロックの発生を抑えることができるというポリシからである．

3.4 メッセージ送信による並行

ここでは，メッセージ送信によって，どのように COOL がプロセスを割り当て/解放するかについて述べる．

COOL におけるオブジェクト A は，メンバ変数集合 V_A と，メンバ関数集合 F_A，レシーバ r_A，ハンドラ h_A，ボディ b_A，そしてメッセージキュー mQ_A の組 $A=<V_A, F_A, r_A, h_A, b_A, mQ_A>$ として表すことにする．また，A に割り当てられているある時点のプロセス集合を $P_{A(0)} = \{p_{A.b}/b_A, p_A/f_i, p_{A.h}, p_{A.r_0}/r_A, \ldots, p_{A.r_j}/r_A\}(f_i \in F_A)$ とする．ここで p/f という表記は，プロセス p にメンバ f が割り当てられていることとする．また、v/B という表記は、メンバ v がオブジェクト B を指していることとする。

COOL におけるメンバは全てオブジェクトであるので，メンバにアクセスするにはメッセージ送信を用いる．今，A が $v_{A_k}/B(\in V_A)$ に対してメッセージ送信 m(同期通信) を行ったとき，プロセスは次のように割り当てられる．ただし，ここでは簡単化のために，m に対する処理が終了するまで，外部からメッセージが届かないと仮定する．また，m が届いた時点での B のメッセージキューを $mQ_{B(0)}$ とする．

1. $p_A \Rightarrow p_A^w$，$P_{B(1)} = P_{B(0)} \cup \{p_{B.r_l}/r_B\}$ $(p_{B.r_l} \notin P_{B(0)})$
2. $p_{B.r_l}/\phi$ のとき，$P_{B(2)} = P_{B(1)} - \{p_{B.r_l}\}$，$mQ_{B(1)} = mQ_{B(0)} \cup m$(キューに対する操作はプログラマがレシーバで記述できるので異なることもある)
3. p_B/ϕ のとき，$p_{B.h}/h_B$(このとき $mQ_{B(2)} \Rightarrow mQ_{B(3)}$，$|mQ_{B(2)}| > |mQ_{B(3)}|$ となることが条件であるが，ここも先程と同様ハンドラに記述できるので，この限りでない)
4. $p_{B.h}/\phi$ で，かつこの結果の戻り先が A であるなら，$p_A^w \Rightarrow p_A$ に遷移し，処理を継続する．

また，同様に非同期通信においては，最初の $p_A \Rightarrow p_A^w$(p^w とは，メッセージを発送したプロセス内のスレッドが同期中であるということを示す) と $p_A^w \Rightarrow p_A$ という部分が，省略される．また，未来型においては通信結果を使用する段階で上記のプロセスの同期が発生する．

次にスレッドについて考えてみる．スレッド t の状態は t^n(評価待ち)，t^x(評価中)，t^w(同期中)，t^e (評価終了) の四つである．また，単に t と表記するときは，t が上記のいずれかの状態にあるとする．

プロセス内には複数のスレッドが割り当てることが可能なので，あるプロセス $p_0 \in P$ は，$p_0 = \{t_0^n, \ldots, t_m^n\}$ とする．同様に，あるスレッド t

が実行する際にロックが掛かっていてはならない集合をU_tと，ロックが掛かっているメンバの集合をLとする．このとき，COOLがあるスレッドt_iに対して，$t_i^n \Rightarrow t_i^x$に遷移させる条件は，t_iが次の二つの条件を満たすときである．

- $i > 0$であるならば，$t_{i-1} \in \{t_{i-1}^x, t_{i-1}^w, t_{i-1}^c\}$
- $U_{t_i} \cup L = \phi$

以上のように，COOLはオブジェクトにプロセスやスレッドを割り当てて行く．

このように，オブジェクト内部は多重プロセスモデルであるが，オブジェクトからあるオブジェクトを見ると単一プロセスモデルのように振舞う．これによって，多重プロセスモデルにおける排他制御を単純化している．また，単一プロセスモデルにおける欠点である応答時間は有限時間内でなくてはならないという制約も排除している．なぜならば，あるオブジェクトが永続的に行う処理は，ボディに記述することが可能であるからである．

3.5 スロット

メッセージの多相性を表現するためにスロットを用いる．例えば，次に挙げた二つのメッセージ通信は，異なる意味を持つ．

```
obj mes number:3;
obj mes string:``abc'';
```

上記の記述において，number と string がスロットとなる．スロット名を省略することも可能であるが，このときはパラメータの数だけを見てマッチングを行うので，期待した結果が得られないこともある．スロットを用いる利点としては，プログラムのドキュメント性を上げるだけでなく，パラメータの受渡し順を考える必要がなくなるという点がある．例えば，次の二つのメッセージ通信は，同じメッセージである．

```
obj mes dist:a src:b;
obj mes src:b dist:a;
```

このようにCOOLは柔軟な記述を可能としている．

3.6 メンバタイプ

オブジェクトの持つ各メンバに対して，共有メンバタイプ，継承許可タイプを指定することができる．共有メンバタイプ宣言によって，共通のクラスを持つインスタンス間で，メンバ関数/変数を共有する．継承許可宣言によって，子クラスへの継承を許す．

メンバタイプには，C++におけるプライベートやパブリックに相当するものは存在しない．これは，外部からの不当なアクセスは許さず，全てのメンバはメッセージ送信によって，アクセスされるものとしているからである．これによってCOOLにおけるオブジェクトは，情報隠蔽とカプセル化がより強固となっている．

4 COOLの文法

COOLのBNFを以下に示す．

```
class    ::= CLASSID INHERIT path EXEC '{' decls '}'
path       ::= aDepthBlock
decls      ::= decl ';'
             | decls ';' decl ';'
decl       ::= type MEMBERVALUE
           ::= type MEMBERFUNC aDepthBlock block
type       ::= kindOfType
             | type '|' kindOfType
kindOfType::= 'inherit'
             | 'share'
aDepthBlock ::= '[' args ']'
args       ::=
             | MEMBERVALUE
             | args ';' MEMBERVALUE
             | args ';' MEMBERVALUE ';'
block      ::= '[' args2 ']'
args2      ::=
             | assign
             | args2 ';' assign
             | args2 ';' assign ';'
assign     ::= obj
             | aDepthBlock ':=' obj
             | MEMBERVALUE aDepthBlock ':=' obj
             | 'self' aDepthBlock ':=' obj
obj        ::= MEMBERVALUE
             | block
             | STRING
             | NUMBER
             | '(' objcet ')'
             | obj MESSAGE parameters
parameters::=
             | parameter
             | parameters parameter
parameter ::= obj
             | SLOTNAME ':' obj
```

5 関連研究

SCOOLはアクタモデル[3]に基づいた並行オブジェクト指向言語である[4]．特徴としては，クラスを持たないプロトタイプベースであり，通信は非同期メッセージパッシングであり，メッセージは有限

時間内に宛先に到着し処理される．また，継続オブジェクトという概念を持っている．COOLと異なる点は，アクタモデルをベースとしているために，オブジェクトの永続的振舞いを記述するのは困難であり，メッセージ送信も非同期のものしかサポートしていない．また，レシーバ，ハンドラという概念を持っていないという点においても異なる．

Javaは，Sun Microsystems社によって開発されたC++をベースとしたオブジェクト指向言語である．特徴として，コンパイラ方式であるがバイトコードを解釈する仮想機械を構築することによって機種依存を無くしつつ，処理速度も比較的早い．C++において難解であったポインタの排除，自動メモリ管理を導入することにより，シンプルな構成となっている．並行処理のためにインスタンスに対してスレッドを用いることができる．COOLと異なる点は，Javaは多重プロセスモデルであるという点と，メッセージ送信による非同期通信ではなく，スレッドインスタンスに対して，インスタンスを割り当てることにより実現しているという点である．またC++をベースにしているために，COOLにおけるレシーバ，ハンドラ，ボディという概念は存在しない．

西尾らの場に基づいた協調的プログラム言語[5]は，オブジェクトの集団を自然にモデル化するために場という概念を導入した，Smalltalkの拡張による言語である．この言語の他の特徴として，非同期/同期のメッセージ送信と場を使ったブロードキャスト通信を持っている．また，オブジェクトはメッセージキューとセレクタ(COOLにおけるハンドラ)を持っている．このセレクタは，COOLと同様にカスタマイズが可能であるが，レシーバに相当する概念を持たず，システムの中に隠蔽されておりカスタマイズは不可能である．また，ボディに相当する概念も持たないという点で異なる．

6 終りに

COOLは，全ての操作はオブジェクトに対するメッセージ送信という統一したスタイルを持っているという点で非常にシンプルな構成となった．

COOLは自動的に最も適したメッセージ送信形態(過去型，未来型，現在型)を選んでくれるという点で，ユーザの負担が小さい．

COOLは，オブジェクト指向パラダイムで，本来含まれていたレシーバ，ハンドラといった概念を，初めてプログラム言語レベルで導入した．これと，言語の持つ並行性によって，自然なモデリングが可能となった．

COOLの持つロック機構も設計段階で自然に導ける形になっている．すなわち，あるメッセージ(又はメッセージ連鎖)に関係するオブジェクトに対してロックを掛けるという記述になっている．このようにCOOLはプロトタイププログラムに適したプログラミング言語といえる．

7 今後の課題

現段階において，デッドロックが必ず起こるメッセージ送信形態が存在する．あるオブジェクトに対して同期のメッセージ送信を行い，その先のオブジェクトの内部において，そのリクエストに答えるための作業途中に，発送元にメッセージ送信を行うという処理が発生すると，COOLはこの二つのオブジェクト間でデッドロックを起こす．この種のデッドロックを無くすためには，メッセージ送信のログを取ることによって検出可能となるが，この方法はメモリとCPUに大きな負荷を掛けることが予想できる．よって，COOLでは，これに類するデッドロックは，検出可能なデバッグモードを別に用意する予定である．

参考文献

[1] L.J.Pinson,R.S.Wiener:Smalltalk: オブジェクト指向プログラミング、トッパン (1990)

[2] 所 真理雄, 松岡 聡、垂水 浩幸:オブジェクト指向コンピューティング、岩波書店 (1993)

[3] G.A.Agha:Concurrent object oriented programming, CACM, Vol.33, No.9, pp.125-141 (1990)

[4] 菅原 太郎,渡辺 卓雄:並行オブジェクト指向言語に対する部分計算, ソフトウェア工学の基礎 III, pp.42-49、近代科学社 (1996)

[5] 西尾 郁彦,渡辺 豊英,杉江 昇:オブジェクトと場に基づいた協調的プログラム言語, 情報処理学会論文誌, Vol.34, No.12, pp.2499-2508(1993)

[6] L.Lemay,C.L.Perkins:Java 言語入門, トッパン (1996)