記 号 処 理　40－1
（１９８７．１．１３）

ＣＯＯＬ

－　オブジェクト指向と制約伝播機構　－

中島　震
日本電気（株）Ｃ＆Ｃシステム研究所

オブジェクト指向プログラミングと相補的なプログラミング・パラダイムとして制約指向／関係指向が知られている．前者が対象の構造記述に向いているのに対して，後者は対象間の関係記述に向いている．問題の側面に応じた記述を行うために，両者をひとつの枠組み内に統合しようという試みが成されてきた．ＣＯＯＬ* は関係記述の評価機構としてＴＭＳ機能を有する制約伝播機構を内蔵するオブジェクト指向言語であり，Ｌｉｓｐ上で両者を融合している．本稿で，ＣＯＯＬにおけるオブジェクトの考え方，言語の概要，デーモンの実行制御について述べる．また，依存型バックトラックの制御をユーザに開放する方式についても述べる．

ＣＯＯＬ

－　Object Oriented Programming & Constraint Propagator　－

Shin Nakajima
C&C Systems Research Laboratories, NEC Corporation
1-1, Miyazaki 4-Chome, Miyamae-Ku,
Kawasaki, Kanagawa 213 JAPAN

Two complementary programming paradigms are known: object oriented and constraint/relation oriented. The former allows the structural description of objects, the latter being suitable for describing relations among them. Much effort has been made to integrate both in a single environment. COOL* is an object oriented language which incorporates TMS as the evaluator for relation descriptions. It thus integrates both programming paradigms on top of Lisp. This report presents COOL's view of objects, the language overview, and the daemon execution control mechanism. It also discusses how the dependency directed backtracking control is available to users.

* COOL ... Constraint and Object Oriented Language

1．はじめに

複雑で大規模なシステムをモデル化する際に有効なプログラミング・パラダイムとしてオブジェクト指向プログラミングがある．本パラダイムによれば，オブジェクトとはデータを中心とするモジュールであり，問題をモジュールの階層としてモデル化することになる．しかし，オブジェクト間に階層を無視した関係が存在するなどの理由により"自然な"モデル化を乱す場合がある．このような場合，関係を記述するために不必要なオブジェクトを導入せざるを得ない．ところで，オブジェクト指向に相補的なモデル化手法として関係記述によるプログラミング手法が知られている[1]．本パラダイムをオブジェクト指向パラダイムと組み合わせることにより，オブジェクト間の関係を記述することができ，自然なモデル化を行うことが可能となる．

一方，モデル化の道具であるプログラミング言語が，オブジェクト指向[2][3]，関係／制約指向[4]，各々の方向から研究されてきている．また，上に述べたような観点から両者をひとつの言語／システムに取り込もうとする試みもあり[5][6]，とくに知識工学の進展と共に研究が活発化している[7][8][9]．本稿で提案するＣＯＯＬ (Constraint and Object-Oriented Language)は，このような研究の流れに沿ったオブジェクト指向言語であり，関係記述の評価機構としてＴＭＳに基づく制約伝播機構[10]を内蔵し，Lisp上で２つのモデル化技法を融合したものである．

以下，２節でＣＯＯＬのオブジェクトを中心に言語の設計方針について，３節で言語の概要について説明し，４節でＣＯＯＬが内蔵する制約伝播機構（ＢＣＰ）とオブジェクト指向／関係指向のインタフェースであるデーモン実行管理方式およびバックトラック制御方式について述べる．

2．ＣＯＯＬオブジェクト

言語の概要を述べる前に本節では簡単な例題とＣＯＯＬにおけるオブジェクトの考え方について説明する．

交通信号の例

３つの信号のうちのひとつ(たとえば青信号)を点灯した時に他の２つ(赤信号と黄信号)を消灯させる問題である．Smalltalk-80 ではオブジェクト間の関係を記述する手段が弱いため，全系を管理するオブジェクトを導入し，かつ特別な処理である dependency を用いて記述している[3]．dependency による記述が直感的でないという欠点がある．ＣＯＯＬでは信号クラスの定義と３つの信号オブジェクトに成立すべき関係(排他関係)の定義により記述した．制約伝播とデーモンを組み合わせて消灯メッセージを送っている（巻末参照）．

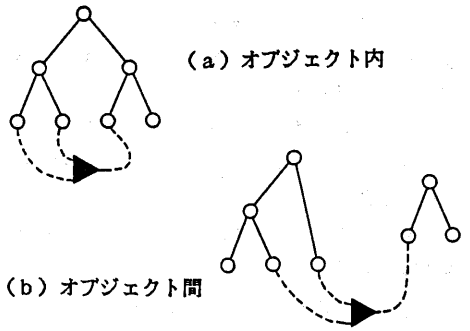
構造記述としてのオブジェクト

ＣＯＯＬでは，オブジェクトおよびメソッド呼び出しに関して "classsical" な立場をとる．すなわち，オブジェクトとは内部状態を有するモジュールであり内部状態として他のオブジェクトを取り得るため階層構造を持つ．さらに，オブジェクトで定義しているメソッドを通じてのみオブジェクトを操作することが可能であり，メソッドの起動はターゲット・オブジェクトに処理内容を記したメッセージを送ることによる．したがって，オブジェクトは構造を持つ能動的な情報担体 (active agent)と言うことができる．

命題としてのオブジェクト

関係記述はオブジェクトの階層モデルに沿わない記述を行うために導入したものである．第１図 (a) のようにオブジェクト内に閉じた範囲の関係記述も (b) のようにオブジェクト間に渡った関係記述も一様に取り扱う．すなわち，構造を超越して関係を記述する．

関係記述の基本は命題論理式である．指定したスロットが値として取り得るオブジェクトが満足すべき関係を命題論理式として規定する．したがって，関係記述の対象になるオブジェクトはひとつの命題とみなすことができる．関連する命題オブジェクト間で論理値を伝播させてデーモンを起動することにより関係を評価する．



第1図　関係の記述

また，関係記述は局所的である．すなわち，命題オブジェクト間で局所的に成立する関係のみについて記述する．大域的な伝播はシステムで自動的に行うので，関係間に矛盾が生じることもある．したがって，矛盾の検出および矛盾解消のバックトラック機構が不可欠となる．

大域環境とメタクラス

ＣＯＯＬではシステム構築(bootstrap)を容易に行うために，クラスはオブジェクトにしてはいない．システムが認識するデータ記述である．したがって，メタクラスも存在しない．強いて言えばＣＯＯＬの大域環境そのものがメタクラスである．大域環境でクラスを一括管理しているからである．また，プログラマが与えた局所的な関係を大域的に伝播するために関係も一括管理している．

３．言語の概要

クラス

クラス定義は def-class により行う．新しいクラスを定義する際には，そのクラスのスーパークラスはすべて存在していなければならない．なお，クラス階層の頂点に位置するクラスとしてクラスObjectがある．

```
(def-class (クラス名　スーパークラスのリスト)
                スロット定義リスト                )
```

```
[例] (def-class (Foo (Bar Baz))
                ((x 10) (y :R 20) z) )
```

上例では，２つのクラス Bar/Baz のサブクラスとしてクラス Foo を定義し，新たに３つのスロット(x y z)を追加した．スロット定義はスロット名とディフォールト初期値の組からなり，z にように初期値を省略した場合は nil になる．また，関係定義（後述）を可能にするためには y のように :R オプション指定が必要である．

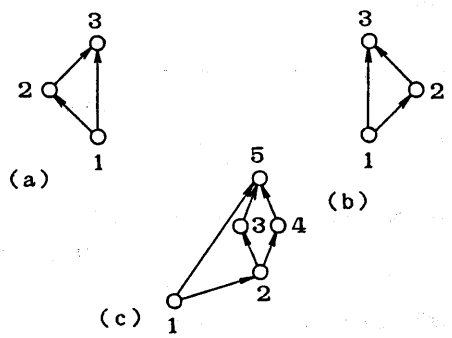
多重継承を考える場合に問題となるのは継承順序である．ＣＯＯＬではクラス定義時に，クラス順序リスト(class precedence list)を作り，以降の継承解釈は本リストを基にして行う．クラス順序リストは次に示す手順で作成する．

① 新たに定義するクラスを先頭に持ってくる．

② スーパークラスのリストを左から右へ深さ優先の探索(left-to-right depth-first)を行ってスーパークラスを集める．

③ ②により複数回現れたクラスは，それ自身のサブクラスよりも後に持ってくる．（この結果，クラスObjectは必ずリストの最後にくる）

第２図にいくつかの例を示した．図中，○はクラスを，→の行き先がスーパークラスを，数字がクラス順序リスト内の順序を示している．局所的なスーパークラスの指定順序が異なるにも関わらず (a) と (b)とで同じ順序になる．



第２図　クラス順序リスト

スロットの継承については次に示す３つの規則により下位クラスで定義するスロットの構造を決定する．

① 上位クラスのスロットをそのまま継承する．

② 同一名称のスロットが複数あっても領域はひとつしか確保しない．すなわち，異なったクラスのメソッド間で領域を共有する．

③ 下位クラスで同名スロットを再定義した場合は下位クラスでの定義を優先する．同名スロットに異なる初期値を与えたり，:R オプションを追加したりするためである．

メソッド

メソッド定義は def-method により行う．この時クラスはすでに存在していなければならない．

```
(def-method (セレクタ名　クラス名)
            メッセージ引数のリスト
            . メソッド本体            )
```

```
[例] (def-method (test Foo) (a b &aux s)
       (setq s (+ a b))
       (setf x s)
       (setf y (* s z))   )
```

例に示したようにメソッド本体からスロットを汎変数(setfの対象)として扱うことができる．

メソッド起動は send により行う．send を明示的に指定する方式と簡略形の両方を提供している．また，send はセレクタ名を評価しないが，performは評価する．

```
(send セレクタ名 レシーバ・オブジェクト
                 . メッセージ引数のリスト)
($セレクタ名 レシーバ・オブジェクト
                 . メッセージ引数のリスト)
(perform セレクタ名　レシーバ・オブジェクト
                 . メッセージ引数のリスト)
```

```
[例] (setq foo-object (new Foo))
     ($test foo-object 1 2)
     (send test foo-object 3 4)
```

メソッド探索は以下の手順で行う．

① レシーバ・オブジェクトのクラスからクラス順序リストを求める．

② 本リストを順にたどる．リストから得たクラスのメソッド・リストを探してセレクタに対応するメソッドを求める．

③ メソッドが見つかるか，リストが尽きるまで②を繰り返す．

④ 見つからなかった場合にはセレクタ名と同名のLisp関数を実行する（ディフォールト・メソッドになっている）．

⑤ Lisp関数も定義されていなければ，レシーバ・オブジェクトにセレクタ未定義を知らせるメッセージ（not-found）を送る．

擬変数 self

メソッド本体／デーモン本体(後述)から実行中のアクティブなオブジェクトを参照するために擬変数 self を提供している．メソッド本体で記述した場合は Smalltalk-80の self と同様である．

メソッド組合せ

メソッドを記述しようとする場合，スーパークラスで定義済みの同一セレクタ名のメソッドに一部機能追加するだけでよいことがある．このような場合のメソッド組合せに関して Smalltalk 流の手続き組合せ方式(super send)と Flavors 流のデーモン組合せ方式(before-after daemon)が知られている．手続き的に記述することにより後者と同等の機能を実現できる．そのためＣＯＯＬでは前者の方式を採用し，次のような関数を提供している．

① (run-super セレクタ名 . メッセージ引数)

指定セレクタを持つメソッドをレシーバ・オブジェクトが属するクラスのひとつ上位のクラス(クラス順序リスト上で直後にくる)から探し始める．レシーバ・オブジェクトは self であり， メソッド探索方式は send の場合に準じる．

② (run-at セレクタ名 クラス名
. メッセージ引数)

run-super に準じるが，指定セレクタを持つメソ

ッドを指定クラスから探し始める．また，レシーバ・オブジェクト(すなわちself)が属するクラスのクラス順序リストに指定クラスが無い場合はエラーとなる．

インスタンス生成

通常，インスタンスを生成するためには生成すべきオブジェクトのクラスにメッセージを送ると考える．この時に起動するメソッドを管理しているのはクラスのクラス，すなわちメタクラスである．ＣＯＯＬでは，大域環境とメタクラスを同一視しているので，クラス名を引数にして関数 new を起動することになる．

インスタンス生成で問題となるのはスロットの初期値をどのようにして与えるか，である．ＣＯＯＬでは new のオプション引数として外部から初期値を与えることができる．また，init というセレクタを持つメソッドを定義することにより初期化メソッドを登録することが可能である．本メソッドのメッセージ引数を初期化リストと呼び，関数 new 起動時のクラス名以外の引数をそのまま渡す．すなわち，初期化リストを解釈し初期化済みインスタンスを生成するメソッドをinitという名称で定義することになる．

関係定義

関係定義は def-relation により行う．関係を課すオブジェクトを命題とみなし，命題間に存在する関係をクローズ形式で与える．命題オブジェクトを値として持つことが可能なスロットにはクラス定義時に :R オプションを付加しておかなければならない．別の見方をすれば，スロットが値として取り得るオブジェクトが満足すべき関係を与えると言うこともできる．

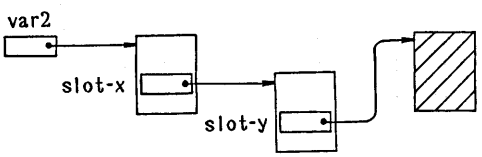
(def-relation 関係名　ポート名のリスト
　　．関係記述の本体　)

関係記述本体は，クローズ定義の他に命題オブジェクトにデーモンをリンクする記述(add-daemon)，命題オブジェクトにアトミックな名前を与える記述（set-tag)，仮説となる命題オブジェクトを指定する記述(assert/assume)等からなる．また，ポート名とは関係記述の本体から命題オブジェクトを参照するための仮変数のようなものである．

関係をオブジェクトに課す(instantiation)ためには，次の例のようにオブジェクトを引数として関係名を指定する．これにより関係記述本体の処理が行われ，論理値を伝播し(４節参照)，デーモンを起動することにより関係評価を行う．

また，パス記述(path description)を提供し関係記述の対象となるオブジェクトの指定を容易にしている．第３図は次例の第２引数の記述を図示したもので斜線のオブジェクトが関係記述の対象となる．ただし，slot-y は :R オプション付きであることを仮定している．

[例] (And var1 ~var2/slot-x/slot-y ~var3)



~ var2/slot-x/slot-y

第３図　パス記述

デーモン定義

デーモン定義には def-daemon を用いる．

(def-daemon デーモン名 ポート名のリスト
　　．デーモン本体)

ポート名は本デーモンを使用する関係定義（def-relation）で定義したポート名に一致させなければならない．デーモン本体には起動された時に成すべき処理を記述する．通常は，メソッド起動（send）等を記述することになる．また，他の制約指向言語と異なり，デーモンの手続き記述により任意のオブジェクトを伝播させることが可能である．

デーモン起動

ひとつの関係記述内で，ひとつの命題オブジェクトに３種類のデーモン(true-daemon, false-daemon ,unknown-daemon) をリンクすることができる．リンクされた命題オブジェクトの真偽値が変化する度に対応するデーモンを起動する．たとえば，真偽値がtrue になると対応するデーモンであるtrue-daemonを起動する．

デーモン本体からも擬変数 self を参照することができ，この場合に self はそのスロットに関係が課されているオブジェクトを指す．第３図の斜線を施した命題オブジェクトにリンクしたデーモンを起動したとする．この時に，そのデーモンから self を参照すると slot-y を持つオブジェクトを参照することになる．

４．制約伝播機構

本節で関係記述の評価機構として内蔵している制約伝播機構(ＢＣＰ)について説明する．

ＢＣＰ

ＣＯＯＬが内蔵する制約伝播機構はＢＣＰ(Boolean Constraint Propagation)と呼ばれている．命題論理を基にして D.McAllester がＴＭＳとして定式化したものであり，以下の特徴を持つ[10]．

① 命題の真偽値を true/false/unknown の３値で表す．ここで unknown は真偽値が未確定であることを示す．

② 命題間の関係を選言クローズ(disjunctive clause)で表現する．

③ 選言クローズを常に満足させるという制約を関連する命題に課すことにより，命題間に論理値を伝播させる．

たとえば，小英文字を命題として，

$$a \wedge \neg b \rightarrow c \qquad ①$$

を $(\neg a) \vee b \vee c$ ② と表す．

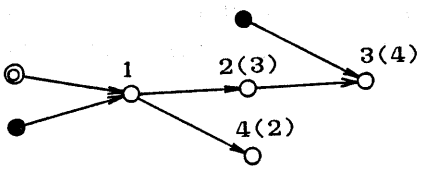
②は宣言的な関係を表すので，③④のような解釈をすることもできる．

$$a \rightarrow ((\neg b) \rightarrow c) \qquad ③$$

$$a \wedge (\neg c) \rightarrow b \qquad ④$$

a，b，c 共に未確定であるとしよう．いま，a ← true，b ← false，とすると①からｃ ← true，を得る．また，a ← true，c ← false，とすると④の解釈からｂ ← true，を得ることができる．このように，ＢＣＰとはクローズを満足するように論理値を伝播させる機構である．

第４図に複数のクローズが介在する例を示した．●の命題はすでに真偽値がさだまっていて，◎の命題に真偽値を設定した(仮説)とする．この時，数字で示した順序で真偽値が伝播し関連する命題の真偽値が(矛盾がなければ)確定する．このように関連する命題の真偽値がすべて定まった状態を安定状態と呼ぶ．また，矛盾が生じたときには矛盾を解消するための依存型バックトラック(Dependency-Directed Backtracking)機構を起動する．このため知識工学でよく見られるような依存関係を用いた探索問題を効率的に実行することが可能である．



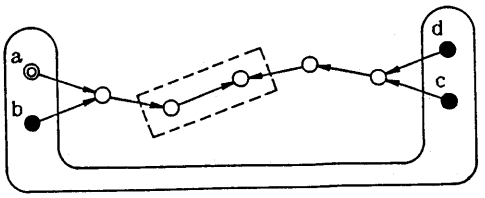
第４図　論理値の伝播

ＢＣＰは論理関係を大域的に伝播するので，関係記述を与えるプログラマは局所的な論理関係のみを考えればよい．

仮説操作

仮説間に矛盾が生じた時に矛盾を解消するためにバックトラックを起こす．ここで矛盾とはクローズが成立しないことであり，本矛盾クローズに関わる命題を導く仮説を求めて，その真偽値を削除することにより矛盾を解消する．

第５図の例では●の仮説に真偽値が与えられている状況で安定状態にあるとする．この時新たに◎で示した仮説の真偽値を設定したために点線で囲んだ論理関係に対応するクローズが矛盾になったとしよう．ここで，原因仮説(ａ～ｄ)から仮説をひとつ選択して，その論理値を削除することにより矛盾を解消しなければならない．ところがａ～ｄのうちどの仮説を選択するかは扱っている問題に依存して決定すべきもので，システムで一意的に規定できるものではない．



第５図　原因仮説

ＣＯＯＬでは上記の問題に対処するために原因仮説を選択する機構をプログラマに開放している．大域変数 *belief-set-class* に仮説選択を行うメソッド select を定義したクラスを設定する．バックトラック時にシステムは原因仮説の集合を求めて，これらを要素として持つオブジェクト(大域変数が示すクラスに属する)を生成する．次いで select メソッドを起動することによりプログラマが指定した仮説選択機構を利用して削除すべき仮説を得る．また，上記大域変数が nil の場合はシステム組み込みの仮説収集手続きを用いる．この場合の方がはるかに良い実行効率を得ることができる．

デーモン実行制御

３節で述べたように制約伝播機構からオブジェクトへのインタフェースはデーモン起動により行う．ところが，一般にはバックトラックが起こる可能性があるため，命題の真偽値が変化する毎にデーモンを起動する方式では問題が起こる．すなわち，デーモン実行により一度変化したオブジェクトの状態をバックトラックと共に元の状態に復帰させる必要があるが，これはきわめて困難である．

ＣＯＯＬではキュー・ベース(Queue-Based)のデーモン管理方式を採用している．キュー内にデーモンとその起動条件とを組にして蓄えておき，安定時に起動条件を満たしているデーモンだけを順に実行する．本方式ではバックトラック過程とデーモン実行過程を独立のフェーズで行うので，バックトラック発生時のオブジェクトの状態に関する問題を避けることができる．また，あるデーモンの実行によりすでにキューされている他のデーモンが無効になることも有り得る．

５．おわりに

オブジェクト指向による構造記述と制約伝播機構を評価機構にする関係記述とを Lisp 上で統合したオブジェクト指向言語ＣＯＯＬについて述べた．現在ＣＯＯＬは UNIX 4.2 bsd の FranzLisp 上で稼働している．

Smalltalk-80の特徴がプログラミング環境にあるように，プログラミング環境のないオブジェクト指向言語は無力であろう．現在は最小限のデバッグ機能(トレーサ，ステッパ，インスペクタ)しか備えていない．クラス階層を検索するツール（browser）なども必須である．さらに，プログラミングの過程ではクラスの修正等が頻繁に起こるため，クラスを変更した場合にシステムで対処する必要がある．しかし，そのためにシステム全体が重くなって通常機能の性能が低下してはならない．現在，詳細な機能を検討中である．

また，ＣＯＯＬの特徴は問題解決機構の一部を成すＴＭＳを内蔵していることである．今後，プログラミング支援システム／論理合成システムの記述を通して設計支援システムにおけるＴＭＳの役割を明確にしていきたい．

最後に本研究の機会を与えて下さった当研究所山本昌弘部長ならびに小池誠彦課長，および本稿に有

益なコメントをくださった当研究所小長谷明彦氏にそれぞれ感謝致します．

参考文献

[1] 淵一博，鈴木則久/プログラミング言語とVLSI，岩波書店，1985.

[2] M.Stefik & D.Bobrow / Object-Oriented Programming: themes and variations, The AI Magazine, (Winter 1985).

[3] A.Goldberg & D.Robson / Smalltalk-80: The Language and its implementation, Addison-Wesley, 1983.

[4] G.Steele Jr. / A Computer Programming Language based on Constraints, MIT/AI-TR-595, (Aug 1980).

[5] 大森ほか/知識埋蔵オブジェクト指向言語Monju 27回情処大会，名古屋，(Oct 1983).

[6] S.Nakajima, K.Ohmori & H.Horita / Monju: Constraint-Keeping Object-Oriented Language COMPSAC84, Chicago, (Nov 1984).

[7] V.Dhar & M.Jarke / Using Technological Design Knowledge for Large Systems Development and Maintenance, 6th Expert Systems & Applications, Avignon, (Apr 1986).

[8] D.Harris / A Hybrid Structured Object and Constraint Representation Language, AAAI-86, Philadelphia, (Aug 1986).

[9] 小長谷明彦 / 型付きユニフィケーションとクローズの対象指向解釈について，コンピュータソフトェア，4(1)，(Jan 1987).

[10] D.McAllester / A Widely Used Truth Maintenance System, unpublished, MIT, (1985).

```
; Light Object Definition

(def-class (Light) ((state :R 'off) color))
(def-method (start Light) ()
   (assume state)
   (send on self))
(def-method (on Light) ()
   ($state! self 'on)
   (format t "Light ~S is on ~%" color))
(def-method (off Light) ()
   ($state! self 'off)
   (format t "Light ~S is off ~%" color))

; Relation among three Light Objects
(def-relation ThreeLights (a b c)
   (add-daemons a 'on-daemon 'off-daemon nil)
   (add-daemons b 'on-daemon 'off-daemon nil)
   (add-daemons c 'on-daemon 'off-daemon nil)
   (add-clause (list (false-term a) (false-term b)))
   (add-clause (list (false-term b) (false-term c)))
   (add-clause (list (false-term c) (false-term a))))
(def-daemon on-daemon (a b c)
   (send on self))
(def-daemon off-daemon (a b c)
   (send off self))

; Start Execution
-> (setq blue (new Light 'color 'blue))
-> (setq yellow (new Light 'color 'yellow))
-> (setq red (new Light 'color 'red))
-> (ThreeLights ~blue/state ~yellow/state ~red/state)
-> (send start blue)
Light red is off
Light yellow is off
Light blue is on
()
-> (send start yellow)
Light blue is off
Light red is off
Light yellow is on
()
->
```

交通信号のプログラム例