ソフトウェア基礎論　35－9
(1990. 5. 24)

並列オブジェクト指向言語Gのリフレクションとその応用

新井 潤　　E-mail jun@astem.or.jp
京都高度技術研究所　　〒600 京都市下京区中堂寺南町

この論文では、並列オブジェクト指向言語Gにリフレクティブ・アーキテクチャを導入する試みについて述べる。オブジェクト指向言語にリフレクションを導入する試みは、さまざまな形でなされている。オブジェクト指向言語にリフレクションを導入する場合、メタな操作の対象となる自己表現をどういう形で入れるかによって、大きく二つの立場が分かれる。一つは、システム全体に対して自己表現を与えることで、もう一つは、各々のオブジェクトに対して、自己表現を与えることである。これまでの多くの試みは、後者の方法で行なわれてきており、前者の立場から試みたものは少ない。オブジェクトは、それ自身一つの計算機であるかのように振舞う。従って、それのみに注目すると従来のプログラミング言語にリフレクションを導入した手法がそのまま使える。しかし、これではオブジェクト指向言語の本来の特徴であるオブジェクトの間の関係をリフレクションで扱うことはできない。そこで本論文では、前者の立場に立ってリフレクションを導入する。また、簡単にその応用例を示す。

A Reflection of The Concurrent Object-Oriented Programing Language G and its Application

Jun Arai
ASTEM (Advanced Software Technology & Mechatronics Research Institute of KYOTO)
Chudoji Minami-machi Shimogyo-ku Kyoto 600 Japan.

This paper presents that we introduce a reflctive archtecture into concurrent object-oriented programing langauge G. When a reflection is introduced into an object-oriented language. There are two kind of methods according to what is represented as self-representation. One is that a whole system, which consists of many objects, is given a self-representation. And the other is that each object is given. An object behaves as if a sigle computer system do, so when one introduce a reflective archtecture into an object, one can take the tecnique by which a reflection is introduced into other programing language, like a 3-lisp. But in the latter method, relations among some objects, which is a fundamental feature of OOL, can't treat as reflection. From the point of view, in this paper, a reflective archtecture is introduced by the formar method. At last, some application of reflection will be presented.

1 はじめに

近年、オブジェクト指向言語に、リフレクションを導入する研究がされている。Maes の 3-KRS[1, 2]、Watanabe の ABCL/R[6]、Briot と Cointe の ObjVlisp[5]、MUSE[9] などが、その一例である。

本論文で考えているリフレクションとは、次のような特徴を持った計算メカニズムである。([1] 参照)

1. システムが自己表現 (self-representation) を持っている。
2. この自己表現が実際のシステムと因果的に結合されている。(causaly connectin)

リフレクションは、1) の自己表現を使って、自分の状態を知ったりその状態を操作したりできるものである。そして、2) によって、その自己表現によって得られた知識が正しいことやそれに対して行なった操作が実際に正確に反映されていることを保証されていなくてはならない。

Ferber[10] は、オブジェクト指向言語のリフレクションに三つの異なるモデルを提案している。一つ目は、メタ・クラスの概念に基づくモデル (meta-class model) で、オブジェクトのクラスをそのオブジェクトのメタ・オブジェクトとするモデルである。ObjVlisp がこのモデルで実現されている。二つ目は、クラスではなくオブジェクトごとにメタ・オブジェクトが存在するモデル (specific meta-object model) である。このモデルでは、各々のオブジェクトのメタ・オブジェクトは、**META-OBJECT** というクラス (または、そのサブクラス) のインスタンスである。これは、3-KRS や ABCL/R などで採用されている。

三つ目は、通信そのものを具体化させるモデル (meta- communication model) である。メッセージを **MESSAGE** クラスのオブジェクトとして、送信が起きると **MESSAGE** クラスのインスタンスを作ってそれに **SEND** メッセージ (このメッセージだけはオブジェクトではない) を送る。**MESSAGE** のサブクラスを定義して、送信する時にそれを指定できるようにしていろいろな型のメッセージ送信を実現できる。

前の二つは、どちらもメタ・オブジェクトの概念を使ったものであるが、一つ目のモデルは、クラス (のインスタンス) 全体の振舞いを制御しているのに対し、二つ目は各々のオブジェクトの振舞いを制御している。この違いを Ferber は、前者を構造的リフレクション (structural reflection) と呼び、後者を計算的リフレクション (computational reflection) と呼んで区別した。

これまでに挙げたものは、(Ferber の三つ目のモデルを除いて) オブジェクトを単位としたリフレクションであった。これに対して、システム全体を対象としたリフレクションが考えれらる。この一つの例として、Watanabe[7] がある。これは、各々のアクターではなく、アクター系の自己表現を別のアクター系で与え、それに対して操作が出来る。また、MUSE[9] もこの一例といえる。これでは、システムを多層化して、上の層が下の層の自己表現になっている。

本論文で提案するリフレクションの機能も、システム全体の自己表現を持ちそれに対する操作が可能なものである。(MUSE のそれに近いものである。) ここで一つ注意すべきことは、オブジェクトを単位として扱うリフレクションとシステム全体を扱うリフレクションは、決して相入れないものではなく、両者を同時実現することも可能であるということである。

本文では、並列オブジェクト指向言語 G に、リフレクティブ・アーキテクチャを導入し、言語に柔軟性を持たる。以下、2 章では、G 言語の動作を形式的に述べる。3 章で、そのメタ・システムを定義し、4 章では、それを使ってリフレクティブな機能を導入する。そして、5 章でその応用例を示す。最後に、まとめと今後の課題などについて述べる。

2 システムのモデル

G のシステムでオブジェクトとメッセージはそれぞれ次のように表すことができる。

$$\mathbf{Object} = \mathbf{Oid} \times \mathbf{Type} \times \mathbf{Body} \times \mathbf{Mqueue}$$
$$\mathbf{Message} = \mathbf{Oid} \times \mathbf{Mbody} \times \mathbf{Context}$$

オブジェクトの識別子 (**Oid**) は、それを特定する

ための整数値であり、型(Type)は、その振舞を決めるもので、一般にはクラスに相当する。そして、内部データ(Body)はそれに固有のデータで、インスタンス変数などに相当する。また、メッセージ・キューは、オブジェクトに届いたメッセージのをためておくものである。($\mathbf{Mqueue} = \mathbf{Message}^*$)

また、メッセージは、メッセージ・ボディ(Mbody)が、送られた時の引数や処理中の一時変数などの記憶領域を表す。(また、メッセージの返事を返すオブジェクトも、ここに記録されている。)($\mathbf{Mbody} = \mathbf{Value}^* \times \mathbf{Robj}$)そして処理状態は、次にどんな処理をするかを決めるものである。

オブジェクトの実行は、メッセージの処理の繰り返しである。メッセージは、その引数の第一番目がセレクタと呼ばれる特別なもので、それを受け取ったオブジェクトでメソッドの選択に使われる。メッセージが送り先のオブジェクトで受け取られると次のように処理される。

1. 受け取ったオブジェクトの型とセレクタから実行すべきメソッドを決める。
2. 求めたメソッドが、メッセージの引数を引数として実行される。

メソッドの実行は、処理状態から得られるオブジェクトの動作(Action)を実行することである。

処理状態 Context とオブジェクトの動作 Action は、次のように与える。

$$\begin{aligned} \mathbf{Context} &= \Sigma \to \mathbf{Context} \times \mathbf{Action} \\ \mathbf{Action} &\subset \Sigma \to \\ &\quad \Sigma \times (\mathbf{Message} + \mathbf{Object} + \{\mathbf{null}\}) \\ &\quad \text{where} \quad \Sigma = \mathbf{Body} \times \mathbf{Mbody} \end{aligned}$$

処理状態は、オブジェクトとメッセージの状態を与えると実行すべきオブジェクトの動作を表す関数と次の処理状態を返す関数である。オブジェクトの動作は、その動作の実行でオブジェクト及びメッセージがどう変わるかという関数として定義される。(但し、システムに許された動作に対応するもののみが存在する。)

$(\mathbf{Message} + \mathbf{Object} + \{\mathbf{null}\})$は、その動作により新たにオブジェクトやメッセージが作られることを意味する。両者とも新たに作られない時は、null となる。

ところで、メッセージの送信方法には二種類ある。一つは、send 型、もう一方は rpc 型と呼ばれる。両者の違いは、メッセージの送信を行なった後、そのオブジェクトが処理を継続して行なうか、送ったメッセージの返値を受け取るまで処理を中断するかの違いである。send 型のメッセージ送信だった場合、メッセージを送信した後、直ちに次のアクションの実行に移る。これに対して、rpc 型だった場合には、メッセージに対する返値を受け取るまで、そのオブジェクトは休止状態になる。休止状態のオブジェクトは、一部の例外(再帰的なメッセージの送信)を除いて、返値を受け取って実行が再開されるまで、他のメッセージの処理を受理したり、別のアクションをしたりはしない。また両者は、Message の中で、Mbody の Robj の値で区別される。すなわち、$\mathbf{Robj} = \mathbf{Oid} + \{\mathbf{null}\}$で、その値が null の時は send 型であることを意味し、それ以外の時は rpc 型で、返事をするオブジェクトの識別子が値となる。

ここで、rpc 型のメッセージの返値を待っている状態を Context が表せないとならないので、Context を次のように改める。この状態を、次のように表す。

$$\mathbf{Wait} = \mathbf{Value}^* \to \mathbf{Context}$$

また、さっき定義した Context は Doing と改める。

$$\mathbf{Doing} = \Sigma \to \mathbf{Context} \times \mathbf{Action}$$

またメッセージを受けたばかりの状態は、start で表す。

$$\mathbf{start} \in \mathbf{Type} \times \mathbf{Mbody} \to \mathbf{Context}$$

(これは、型と引数によって最初の処理状態が決まることを意味する。)そして、処理が終了したことを表すのは、end 状態である。これらによって、Context は表される。結果として次のようになる。

$$\mathbf{Context} = \{\mathbf{start}, \mathbf{end}\} + \mathbf{Doing} + \mathbf{Wait}$$

最後に、システム全体の状態を定義する。システムは、さまざまなオブジェクトとその間を行き

```
script meta_object_script
    on .get_type
        return object_type
    end on

    on .get_body
        return object_body
    end on

    on .set_body
        object_body = %[1]
    end on

    on .get_next_message
        return mqueue
    end on

    on .remove_top_message
        mqueue = %next(.get_next)
        return %next
    end on

    on .receive_message
        %msg = %[1]
        if color == %msg(.get_color)
            @(.insert_top_mqueue, %msg)
        else
            @(.insert_last_mqueue, %msg)
        end if
    end on

    on .insert_top_mqueue
        %top = %[1]
        %top(.set_next, mqueue)
        mqueue = %top
    end on

    on .insert_last_mqueue
        mqueue(.insert_last, %[1])
    end on
        .
        .
        .
end script
```

図 1: オブジェクトのメタ表現

来するメッセージの集まったものと考えられるので、その状態を **Object** の集合として定義する。

$$\mathbf{State} = \{S \in \Omega | \forall \langle o,t,b,q\rangle, \langle o',t',b',q'\rangle \in S, o \neq o'\}$$

where Ωは **Object** の有限部分集合全体

システムの状態は各オブジェクトの動作に伴って、遷移する。ただし、前節の最後で述べたようにオブジェクトの動作は個々独立していて同時に実行できるが、いまは、逐次処理の計算機で実現することを考えているので、オブジェクトの動作は順に一つずつ起こるものとする。システムの状態がS_1の時、オブジェクトOの動作によってS_2に状態が遷移したことを、

$$S_1 \xrightarrow{O} S_2 \qquad (S_1, S_2 \in \mathbf{State})$$

と、記述する。

3 メタ・システム

2章で述べたシステムを扱うメタ・システムを同じモデルで表現する。メタ・システムを考えるために、まずオブジェクトとしてどういうものがこのシステムに必要であるかを考える。

オブジェクトとメッセージ

対象システムのオブジェクトやメッセージを表現するのに、二つの方法がある。一つは、メタ・システムのあるオブジェクトの内部データとして表現する方法である。もう一つは、各々の独立したオブジェクトとして表現する方法である。

ここでは、後者の方法で扱うことにする。つまり、対象システム内のオブジェクトやメッセージの各々に対応するオブジェクトがメタ・システム内に存在する。これらのオブジェクト群が対象システムの状態Sを表していると考えられる。(このオブジェクト群を$\uparrow S$と書く)

対象システムのオブジェクトに対応するメタ・システム内のオブジェクトはすべて同じ型を持っている。この型の定義するオブジェクトは、次のような形をしている。オブジェクトOに対応するメタ・システムのオブジェクト$\uparrow O$は、内部データとして、Oの型、内部データ、メッセージキューなどを持っている。メソッドとしては、Oの型の参照やOの内部データの参照や変更などがある。これをG言語で書いてみると図1のようになる。

また、メッセージに関しても同様で、内部データとして引数の列や処理の状態を持っていて、メソッドとしてそれらの参照や変更が定義されている。

実行器

メタ・システムにおいて、対象システムの実行(メッセージの処理)は、次のような手続きとして表現される。対象システムでオブジェクトoに届いたメッセージmを処理するものとして、それぞれに対応するメタ・システム内のオブジェクトを$\uparrow o$、$\uparrow m$とすれば、

1. $\uparrow o$のメッセージ・キューから一つ取り出す。(これは、$\uparrow m$である。)
2. $\uparrow m$に$\uparrow o$を与えて処理状態から、次のアクションを決定する。
3. アクションを実行する。
4. $\uparrow m$の処理状態を更新する。
5. これを決められた回数(AC:Action Count)繰り返す。

3)でアクションの実行により生じたoの内部データを変更するは、$\uparrow o$で対応するデータを変更することによって処理する。(メタ・レベルで$\uparrow o$に内部データ変更のメッセージを送る。)

これらの処理をするのが実行器オブジェクトである。(オブジェクトやメッセージと同様Gのコードとして示すこともできるが紙面の都合上ここでは省略する。)処理すべきオブジェクトを渡された実行器は、それが処理中のメッセージを取り出す。そしてメッセージの処理状態から実行すべきアクションを得て、そのアクションを処理する部分を実行する。これは、Contextに、Σを与えて、次のContextと実行すべきActoinを得て、そのActionを実行することである。メッセージの処理が終了すれば(処理状態がendになれば)メッセージのメタ・オブジェクトを消去する。

またアクションの例として、rpc型のメッセージ送信の場合を見てみる。rpc型のメッセージを処理するプログラムは次のようなものである。

```
1 case .RPC:
2    %target_obj = 送り先オブジェクト
3    %new_msg = *message_script(.create)
4    %new_msg にrpc型であることとその返値
        を送る先、それと引数を設定
5    %target_obj(.receive_message, %new_msg)
6 end case
```

まず2行目で、送り先のオブジェクトを求めている。これは、それまでの計算結果から求まる。つまりcurrent_msgの処理状態(Context)とメッセージ・ボディから分かるので、current_msgにメッセージを送って求める。続く二行で新しいメッセージ・オブジェクトを作って、返値を送る先(すなわち現在のオブジェクト)や引き数など必要な値を設定する。(引数なども送り先のオブジェクトと同様にcurrent_msgにメッセージを送って知ることができる。)そして、送り先のオブジェクトのキューへ繋ぐ。これをするのがreceive_messageというメソッド(図1参照)である。今の場合は、rpc型なので処理状態は送ったメッセージの返値を待つ状態(処理状態がWait)になっている。

メタ・システム

メタ・システムは、主なオブジェクトとしてここに挙げた三つのものである。

(1) 対象システムのオブジェクトを表すもの

(2) 同じくメッセージを表すもの

(3) それらを実行するもの

1)は対象システムにあるオブジェクトの数だけインスタンスを持ち、2)も未処理のメッセージの数だけある。3)は、一つのインスタンスを持っている。

また、オブジェクトの実行順序を決定するためのオブジェクトも定義でき、それでスケジューリングなどもリフレクションで扱えるようになる。この他にも実行環境を表すようなものが必要であるが、それらもオブジェクトとして実現できる。例えば、対象レベルのオブジェクトとそのメタ・オブジェクトの対応を与えるものなどが実行環境である。

図2)は、メタ・システムと対象システムの関係を図にしたものである。横線より下が対象システムで、上の小さな四角と菱形が、それぞれオブジェクトとメッセージのメタ・オブジェクトである。丸

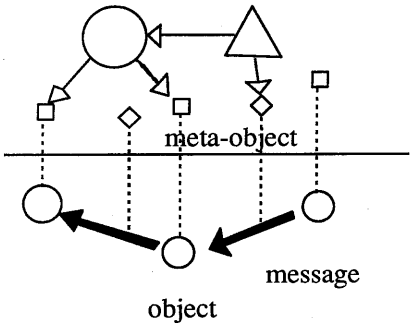

図 2: メタ・システム

や三角が、実行器などを表していて、矢印はすべてメッセージである。

4 リフレクション

ここまで、Gのシステムとそのメタ・システムについて見てきた。前節で与えたメタ・システムは、Gのインタープリタになっている。従って、これを使って対象システムを実現することができる。

いま、対象システムの解釈実行をメタ・システムが行なっているものとする。もしも、メタ・システムとメッセージのやりとりをしたり、メタ・システムのオブジェクトを作ることが出来れば、メタ・レベルにある(オブジェクトの集合として表現されている)対象システムの状態を参照・変更することが出来る。つまり、メタと対象の間のメッセージのやりとりができて、またメタ・レベルのオブジェクトを定義できるシステムを作れば、対象レベルでリフレクティブ・アーキテクチャを持つシステムを実現できる。

こうして作ったシステムは、1節で述べたリフレクションとしての二つの条件を満たしている。1)の自己表現として、メタ・システムのオブジェクト群を持ち、実際の実行は、そのオブジェクトの操作なので(メタ・システムがインタープリタだから)自己表現と対象システムの状態は一致している(2)。

以下のこの節では、メタと対象の間のメッセージのやりとりを実現する方法とメタ・レベルのオブジェクトを定義する方法について簡単に述べる。

メタと対象レベル間のメッセージ送信

まず、メタと対象レベルの間でメッセージをやりとりする方法について考える。

メタ・レベルから対象レベルへメッセージを送ることは比較的簡単で、3節で述べた message_script 型のオブジェクトを作って目的とするオブジェクトのメタ・オブジェクトのメッセージ・キューに繋いでやれば良い。これは、メタ・システムの実行器のコードを書き換えることで簡単にできる。

問題となるのは、この逆で対象レベルのオブジェクトからメタ・レベルへメッセージを送信することである。これを実現するためには、二つのことを解決しなくてはならない。一つは、メタ・レベルのオブジェクトを指定する方法で、もう一つは、指定したオブジェクトに実際にメッセージを送るメカニズムである。

前者は、対象システムのすべてのオブジェクトに与えられているオブジェクトの識別子をメタ・レベルのオブジェクトにも与えることで解決できるが、新たに「どうやってその識別子を得るか」という問題が発生する。簡単な解決策として特定のメタ・レベル・オブジェクトには、固定の名前を与える方法が考えられる。G言語では、名前とオブジェクトの対応関係を維持する環境を持っていて、プログラムの中で、*<name>という形でその名前<name>に対応するオブジェクトを参照することが出来る。(正確に言うと*<name>は、オブジェクトの識別子を表す。)この環境を利用して、メタ・システムの代表的なオブジェクトにはあらかじめ識別子を与え環境に決められた名前(例えば、実行器ならば the_interpreter という具合)で登録しておく。この方法は、実行器のように、既にあることが分かっているものには有効であるが、個々のオブジェクトのメタ・オブジェクトなどのように計算過程で生成・消滅を繰り返すものには使えない。そこで、別の手段を提供しなくてはならない。そこで、Oを与えると$\uparrow O$を返す関数を用意する。記法として@meta(o) と書くと o の表すオブジェクトのメタ・オブジェクトを表すことにする。(G言語には "@" で始まるさまざまなオペレータがあるので、@meta も単項のオペレータとして定義すれ

```
1 case .RPC:
2   %target_obj, %meta = 送り先オブジェクト
3   if %meta==$true;%target_obj がメタ・オブジェクト
4     current_msg の処理状態から引き数を得る。
5     引数を持って%target_obj に RPC する。
6     返値を current_msg に与えて新しい状態にする。
7   else
8     %new_msg = *message_script(.create)
9     %new_msg に引数などを設定
10    %target_obj(.receive_message, %new_msg)
11  end if
12  break for
13end case
```

図 3: メタ・レベルのオブジェクトにメッセージが送れるように変更された rpc の手続き

ば良い。)

実際にメッセージを送るメカニズムは、実行器のメッセージ送信のアクションを実行する部分を (send 型と rpc 型ともに) 変更することで解決できる。ここでも、前と同じ用に rpc 型の場合を例にとって説明する。実行器の rpc 型メッセージ送信の処理部分を図 3に示す。この図の 2 行目で送り先オブジェクトを求めたとき、それがメタ・レベルのオブジェクトかどうかも一緒に判断する。二番目の返値が$true (真の値を表す) であれば、送り先はメタ・レベル・オブジェクトである。メタ・レベルであれば current_msg の処理状態から引数を取り出して、直接メッセージを送信する。そうでなければ今までと同じようにメッセージ・オブジェクトを作って、必要な値を設定し、送り先のオブジェクト (正しくはそのメタ・オブジェクト) のメッセージ・キューに繋ぐ。

リフレクティブ・オブジェクトの作成

前節では、リフレクティブな機能の一つとしてメタと対象レベルのオブジェクトの間でメッセージの送信を行なう機能を提供した。この節では、もう一つのリフレクティブな機能としてメタ・レベルのオブジェクトを生成する機能を考える。

オブジェクトの型を表すスクリプトは、その型が "script" であるオブジェクトである。(型が "script" のものをスクリプト・オブジェクトと呼ぶことにする。) このスクリプト・オブジェクトを型とするオブジェクトは、メッセージ.create を送ることで生成される。これは、"script" という型に.create と言うメソッドが定義されていて、そのメソッドを実行するとそのオブジェクト自身を型とするオブジェクトが生成される。

同様に、メタ・レベルのオブジェクトを生成するにはスクリプト・オブジェクトにメッセージ .meta_create を送ることにする。つまり、スクリプト "script" には、.meta_create メソッドも定義されていて、このメソッドはメタ・レベルのオブジェクトを生成する。(このメタ・レベルのオブジェクトを、メタ・システムに始めからあるオブジェクトと区別するためにリフレクティブ・オブジェクトと呼ぶ。) この時生成されたオブジェクトの型は、.meta_create メッセージを受けたスクリプト・オブジェクトになる。

リフレクティブ・オブジェクトは、メタ・レベルで実行されるので、このオブジェクトを実行するためのさらにメタ・レベルのシステムが必要になる。メタ・システムが対象システムと同じモデルで与えられていたので、このメタ・レベルのシステムも同じメタ・システムで実現することができる。ところが、メタ・レベルでさらに、.meta_create メッセージをスクリプトに送ったとすれば、メタ・レベルにおけるメタ・レベルのオブジェクト、すなわちメタ・メタ・レベルのオブジェクトが生成される。

このようにして、メタと対象のこの関係は (概念的には) 無限に続くものである。この無限に続くメタと対象の関係を実現するには、無限のレベルのメタ・システムが存在しなくてはならない。この無限に続く階層をリフレクティブ・タワーと呼んでいる。スクリプトはそれを定義すると、この階層のすべてにスクリプト・オブジェクトが作られることにする。

ここで、オブジェクトの生成について整理し直してみる。

- スクリプトを定義するとすべてのレベルに同じスクリプト・オブジェクトができる。
- N レベルで実行されているオブジェクトがスクリプトに.meta_create メッセージを送ると $N+1$ レベルで実行されるオブジェクトがで

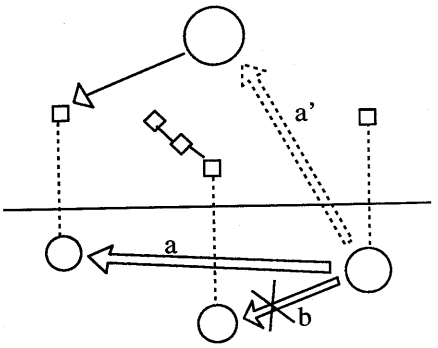

図 4: リフレクティブ・オブジェクトの簡単な例

きる。

- この時できたオブジェクトの型は、$(N+1)$ レベルにある同じスクリプトになる。

簡単な例

ここでリフレクティブ・オブジェクトを使った簡単な例を示す。これは、あるオブジェクトにメッセージを送る時、そのオブジェクトのメッセージ・キューにメッセージがなければ送り、あれば送らないというもの。これは、あるリフレクティブ・オブジェクトに送り先のオブジェクトと送りたい引数を引数として .send というメッセージを rpc 型で送れば良い。この例では、スクリプト example を、リフレクティブ・オブジェクトとして作っておく。送り先オブジェクトとその引数を引数として、メッセージ send を送る。図 4) は、この様子を表したものである。この図では、a のメッセージは送れるが b は送り先で、メッセージが溜っているので失敗する。また、a' が、a を送る時のリフレクティブ・オブジェクトへの rpc 型メッセージである。(a は、send 型)

同じことを rpc 型で実現するのは少し難しい。このリフレクティブ・オブジェクトを呼び出した時の返値が、目的とするオブジェクトへの rpc の返値になることが期待されている。ところが、この

```
1  script example_reflection
2    on .do
3      %t = *the_catch_throw(.catch, .tag, @)
4      if %t == 1
5        return 1
6      end if
7      @(.test, %[1])
8      return 0
9    end on

10   on .test
11     if %[1] > 0
12       *the_catch_throw(.throw, .tag, @, 1)
13     else
14       return
15     end if
16 end script
```

図 5: リフレクションを使った catch/throw の例

オブジェクトから普通にリターンすれば、その値が返値となる。そうしないように、ダミーのメッセージを用意してすぐ終る状態にしておいて、リターンするなど工夫が必要である。

5 リフレクションの応用例

リフレクションの応用として、catch/throw をリフレクティブ・オブジェクトとして定義する。ここで、G の catch/throw について、少し説明しておくと、throw 文で指定されたタグと同じタグを指定した catch 文に実行が移るのであるが、そのとき、catch 文は、rpc の連鎖をたどって探される。rpc 型のメッセージ送信をすると送信したオブジェクトは返値を待って実行を中断する。送信した先のオブジェクトがまた rpc 型のメッセージを送るとそのオブジェクトも中断される。こうして、中断しているオブジェクトの作る列を rpc の連鎖という。

この機能を実現するプログラムを図 6) に示す。catch 文と throw 文の変わりにこのオブジェクトに catch と throw タグと自分の識別子を引数として rpc 型のメッセージを送れば良い。

図 5) は、catch/throw を使ったプログラム例である。最初に 3 行目で、.catch を送った時は、0 が返ってくるので、4,5,6 行目を飛ばして 7 行目を実行する。しかし、12 行目で .throw を送ったら、

```
script catch_throw
    on .catch
        %tag, %o = %[1:2]
        %msg = %o(.top_message)
        %context = %msg(.get_context)
        %catchs = @assoc(tags,%tag,0)
        if %catchs == 0
            %catchs = *v_any(.create,0)
        end if
        @body(%catchs) = 2, %msg, %context
        @assoc(tags,%tag) = %catchs
        return 0
    end on

    on .search
        ...
    end on

    on .goto_tag
        ...
    end on

    on .throw
        %tag, %o = %[1:2]
        %i = 0
        %catchs = @assoc(tags,%tag,0)
        %msg, %context = @(.search, %o, %catchs)
        if %msg != 0
            @(.goto_tag, %o, %msg, %context)
            return @body(%[3])
        else
            ERROR
        end if
    end on
end script
```

図 6: catch/throw のリフレクションによる実現

3行目を呼び出した時の状態(**Context**)に戻って、1を返すので、4行目の条件を満たし5行目が実行される。

スクリプト catch_throw の動作を説明すると、.catch が送られると、その時のメッセージの処理状態とメッセージを与えられたタグとペアで登録しておく。.throw が来ると、そのタグで登録されているメッセージとその処理状態のうちメッセージがrpcの連鎖をたどって始めに現れるものを持ってくる。(.search でこれを行なう。)見つかれば、またrpc連鎖をたどりながら(中断している)メッセージを消去しながら、見つけたメッセージまで戻る。そのメッセージの処理状態を登録されていたものにして、処理中のオブジェクト(これを実行中に.throwを送ったオブジェクト)から、そのメッセージを実行するオブジェクト(.catch を送ったオブジェクト)に処理を移す。catch を rpc で送ってあるので、登録されていた処理状態(**Context**)は、rpc の返値待ち状態なので、throw でそのまま return すると、その返値が、.catch を送った時の返値のように返ってくる。(図3の5,6行目を参照。)

6 まとめ

この論文では、はじめにG言語のモデル化を行なって、オブジェクトの集合として計算状態を表した。次に、そのモデルを扱うメタ・システムを導入した。このメタ・システムで、言語のモデルを実現することができる。そして、このメタ・システムを用いてメタ・レベルのオブジェクトとメッセージをやりとりしたり、メタ・レベルのオブジェクトを作ったりすることができる機能を持つリフレクティブ・システムを実現できることを示した。

その応用例として、メッセージを送るときオブジェクトの状態によって送るのを止めたりできることをみた。また、catch/throw という制御構造をリフレクションを使って実現できることを示した。

今後の研究課題としては、オブジェクト指向言語の二つ立場からのリフレクション、ここで述べたものとオブジェクトごとに自己表現を持たせるものを融合するするにはどうすれば良いかということ。また、融合したことによって更に実現できることが増えるのか、あるいは、融合したことによってどんな弊害があるかなどの研究が挙げられる。

この他に、スケジューリングの応用例などは、実際にインプリメントした場合に、どれだけ性能が改善されるのかなどは、興味深いところである。

謝辞

本研究を行なうにあたって、御指導をいただきました京都大学数理解析研究所の中島玲二教授に心からお礼申し上げます。また、貴重な御指導、御助言を頂きました京都大学大型計算機センターの萩野達也助手に深く感謝致します。

参考文献

[1] Pattie Maes: "Computational Reflection", Proc. 11th German Workshop on Artificial Intelligence, 1987.

[2] Pattie Maes: "Concepts and Expreriments In Conputational Reflection", Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Application, Orlando, 1987.

[3] Weyhrauch, W: "Prolegomena to a Theory of Mechanized Formal Reasoning", Artificial Intelligence, Vol 13, 1980.

[4] Smith, B.C.: "Reflection and Semantics in Lisp", Proc. 11th ACM Symposium on Principles of Programming Language, 1984.

[5] Jean-Pierre BRIOT, Pierre COINTE: "The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language", Proc. of the European Conference on Artificial Intelligence, 1986.

[6] Takuo Watanabe, Akinori Yonezawa: "Reflection in an Object-Oriented Concurrent Language", Proc. ACM Conference on Object-Oriented Programming, System, Languages and Application, 1988.

[7] "分散システムのための並列自己反映計算モデルに向けて", 日本ソフトウェア科学会 第6回大会論文集 , 1989.

[8] Akinori Yonezawa, Takuo Watanabe: "An introduction to Object-Based Reflective Concurrent Computation", ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Vol24, No.4 1988.

[9] 横手靖彦、寺岡文男、山田正樹、手塚宏史、所真理雄: "MUSE オペレーティングシステムの設計・実装、及びプロトタイプシステムの評価", 日本ソフトウェア科学会 第6回大会論文集 , 1989.

[10] Jacques Ferber: "Computational Reflection in Class based Object Oriented Languages", Proc. ACM Conference on Object-Oriented Programming, System, Languages and Application, 1989.

[11] 菅野博靖、田中二郎: "メタ推論とリフレクション", 情報処理 Vol.30, No.6, 1989.

[12] 田中二郎: "メタプログラミングとリフレクション", Bit Vol.20, No.5, 1988.

[13] 大谷浩司、角野宏司、児島彰、萩谷昌己、服部隆志、劉樹令: "GMW ウィンドウ・システム上のアプリケーション構築について" , コンピュータ・ソフトウェア Vol 7, No.1, 1990.