# Open XM の設計と実装: クライアントサーバモデルと数学共通表現 (OpenXM-RFC 100, proposed standard)

— Open message eXchange protocol for Mathematics

野呂正行, 高山信毅*

1997 年, 11 月 20 日 — 2000 年, 11 月 17 日,
2001 年, 8 月 27 日, 2002 年, 1 月 20 日 (小修正),
これ以後の修正は changeLog を見よ.

ChangeLog
2004-3-8: 新しい SM コマンド SM_executeFunctionWithOptionalArgument を加えた.
2005-3-4: 浮動小数点数を送るための byte order negotiation についての記述を加えた. CMO_64BIT_MACHINE_DOUBLE

*神戸大学理学部数学教室,http://www.math.kobe-u.ac.jp/~taka

# Contents


**1 はじめに** **4**

**2 CMO Primitive object** **5**

**3 CMO の形式的表現方法** **7**

**4 OpenXM の通信モデル** **10**
4.1 OX Message の 表現方法 . . . . . 11
4.1.1 OX メッセージの表現法 (Lisp 風) . . . . . 11
4.1.2 OX メッセージの表現 (XML) . . . . . 12
4.2 OXexpression の 標準 encoding と TCP/IP ソケットによる実装法 12

**5 OX スタックマシン** **14**
5.1 サーバスタックマシン . . . . . 14
5.1.1 サーバスタックマシンのグループ SMobject/Primitive に属するオペレータ . . . . . 14
5.1.2 MathCap . . . . . 18
5.1.3 例 . . . . . 19
5.1.4 グループ SMobject/Basic に属するオペレータ . . . . . 20

**6 現在検討中の機能** **21**
6.1 OX DATA with Length の構造 . . . . . 21
6.2 サーバスタックマシンはローカルな拡張機能をもってよい . . . . 22
6.3 MathLink や OpenMath などの通信規約の実装 . . . . . 22
6.4 共通スタックマシン言語 . . . . . 22

**7 Session Management** **22**
7.1 Control server . . . . . 22
7.2 New OpenXM control servers . . . . . 23
7.3 OpenXM reset protocol . . . . . 23
7.4 Control message (SMObject/TCPIP/Control) . . . . . 24
7.5 Notification from servers . . . . . 25

**8 TCP/IP でのセッションのスタート** **25**
8.1 Standard I/O on OX servers . . . . . 25
8.2 Launcher . . . . . 26
8.3 byte order の決定 . . . . . 26
8.4 launcher の例 : `ox` . . . . . 26
8.5 OX サーバの利用例 . . . . . 27

**9 オブジェクトの文字列表現** **27**

**10 数, 多項式 の CMO 表現** **28**
10.1 Indeterminate および Tree . . . . . 29
10.2 Zero . . . . . 31
10.3 整数 ZZ . . . . . 31
10.4 分散表現多項式 Dpolynomial . . . . . 32
10.5 再帰表現多項式の定義 . . . . . 33
10.6 CPU 依存の double . . . . . 34

**11 OX Local Data** **35**

**12 CMO ERROR2** **35**

**13 メンバの再定義に関する注意** **36**

**14 実装, デバッグ, 検証** **36**
14.1 実装の手順 . . . . . 36
14.2 歴史 . . . . . 36
14.3 サンプルサーバ, クライアント . . . . . 37
14.3.1 OpenXM/src/ox_toolkit にあるサンプル実装 . . . . . 37
14.3.2 ox_null . . . . . 37
14.3.3 testclient . . . . . 37
14.3.4 `ox` . . . . . 38
14.3.5 `ox_asir` phrase book . . . . . 38
14.3.6 `ox_sm1` phrase book . . . . . 39
14.3.7 `ox_sm1` を用いたクライアントのテスト方法 . . . . . 40
14.3.8 `Asir` を用いたサーバのテスト方法 . . . . . 40
14.4 最小の TCP/IP クライアントの例 . . . . . 40
14.5 クライアント asir, sm1 . . . . . 40
14.6 開発中のサーバ, クランアント . . . . . 41
14.7 Change log . . . . . 41
14.8 . . . . . 41

**15 どのように協調機能にすぐれた Math Soft を書くのか?** **42**
15.1 open gnuplot の実装 . . . . . 42
15.2 open phc の実装 . . . . . 42
15.3 open M2 の実装 . . . . . 42

**16 新しい CMO の登録** **42**
16.1 新しい CMO を作るときの約束 . . . . . 42
16.2 OpenXM プロジェクトに参加するには? . . . . . 42

**17 OX サーバに対する C ライブラリインタフェース** **43**

Draft for protocol version 1.1.3.
1.1.3 is encoded as 001001003 in mathcap.

kxx/openxxx.tex. **Draft** 1997, 11/20 — . この文書は open XM の設計用のメモとしての役目もあるので, 一部のファイル名は開発者のみが参照できる.

# 1 はじめに

OpenXM は, おなじタイプまたは異なるタイプの数学プロセス間のメッセージのやりとりの規約である. 開発の動機は, 手作り (または研究的な) 数学ソフトの相互乗り入れの実現および分散計算の実装が第一であったが, もちろん数学ソフト間だけでなく, ワープロソフトや, インタラクティブな数学本, さらには数学デジタル博物館用のソフトがこの規約に従い, 数学ソフトを呼び出すことなどにも利用できる. 当面の目標は OpenXM 数学ソフトパッケージを作ることである. これはさまざまな数学ソフトを一つのパッケージとして簡単に好きな言語より使えるようにするプロジェクトである. 現在 OpenXM.tar.gz には, asir, sm1, phc, gnuplot, tigers が入っている. OpenXM 数学ソフトパッケージプロジェクトに参加できるように CMO の規約を拡張していくための規約も定めるものとする.

現在専門的な数学ソフトが多数開発されているが, 次世代の数学ソフトは他のソフトよりサブルーチンとして呼び出せる機能をもつべきである. このように協調機能にすぐれたプログラムを書くための設計のガイドライン, プログラム作法のようなものを提案するのも, このプロジェクトの目的である. このようなガイドラインにそってプログラムすることにより, 数学アルゴリズム自体に集中できる環境が実現できることを期待している.

設計の方針として, (1) 単純 (2) 拡張性 (3) 実装の簡便さ (4) 実用性 (5) 高信頼性 (robustness), に重きをおいている.

OpenXM はなにも考えずに簡単に接続できるシステムを作ろう, というまで野心的ではない. 数学的な object は一筋縄ではいかないし, 完全な統一規格をつくるというのは気が遠くなる仕事である. そのかわり, 今よりすこしだけこういったデータ交換や分散システム構築の仕事を楽にしたいというのがささやかな第 1 目標である. また, Mathematica や Maple といった巨大な統合ソフトを, Free Mathematical Software としてわれわれ自身の手でつくっていくための基礎でもある.

数学的な Object をどのように表現するのか, どう伝えるのかを考えることは決してつまらない問題ではない. このような問題は, 新しい数学記号を創造する問題と似ているかもしれない. 我々は, 数字を 0 を含んだ 10 進数で表記し, 微分を $dx$ と書き, 写像を $\longrightarrow$ であらわす. これらの記号法からどれだけ多くの利益を得ているか, 思いをはせて欲しい.

OpenXM は論理的には OX 層, SM 層, CMO 層にわかれる. OX Object は, データ, コマンドに大別できる. データはこのプロジェクトのオリジナルのデータ形式である CMO (Common Mathematical Object Format) で送出してもよいし, MP や Open MATH などの形式を用いてもよい. SM 層は (スタックマシン) サーバを制御するための命令のあつまりであり, サーバは非同期的に動作させることが可能である. 柔軟性が高いので, IMC などのリモートプロシージャコール系のプロトコルもエミュレートできる. OX メッセージは DTD を用いて定義可

能であり, XML を用いて記述できる.

OpenXM に準拠したシステム xxx を, open xxx とよぶ. たとえば, OpenXM 対応の asir は open asir であり, OpenXM 対応の kan/sm1 は open sm1 である.

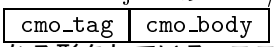
OpenXM は, データ型をきめている部分と, 共通スタック言語を定めている部分にわかれる. よいたとえかどうか自信がないが, OpenXM を定義する作業は, unicode を定める作業に似ている部分もある. たとえば, 漢字は東アジアで共通に使用されているが, 国や地方単位で形がすこしづつ違うものがある. unicode では, 似た漢字を同じ code におしこんだ. OpenXM でも, システム毎にことなるが似ているデータ型を同じ型としてあつかう. たとえば, "分散多項式" の意味は, asir と kan/sm1 で異なるが, OpenXM では, 同じ型としてあつかわれる. ただしこれでは不十分な場合があるので, システム固有のデータ型も扱えるような仕組みも用意している. (さっきのたとえでは, こちらは, unicode でなく, ISO の文字符号系に対応するか.) 共通スタック言語は, ごく小数の共通コマンドをのぞき基本的にシステム毎に固有の関数により実行される. これにより, open xxx のサーバの実装およびマニュアルの記述はいちじるしく簡単になる. システム毎に固有の関数を共通の関数名にするプロジェクトも考えているが, 共通の関数名は OpenMath のものを利用する予定である.

## 2 CMO Primitive object

CMO (Common Mathematical Object format) グループ Primitive の Object は, ローレベルのデータ型, `int` , `string` などに対応する object である. この CMO はすべての OpenXM が実装していると仮定される. この節では, イントロダクションとして, グループ Primitive に属する CMObject (Common Mathematical Object) を形式的な方法をつかわず導入しよう.

このグループの正式な名前は, CMObject/Primitive である. 以下, `int32` で 2 の補数表現された 32 bit integer をあらわす (これはよく使われる計算機での C 言語の int の内部表現). `byte` で 8 bit データをあらわす.

CMObject の TCP/IP 用の実装では, CMO の object は




なる形をしている. ここで, `cmo_tag` は, 正の `int32` で表現するものと規約する.

`cmo_tag` は object のタイプをあらわすタグであり, 以下のように決めている.

@../SSkan/plugin/cmotag.h

```
#define LARGEID  0x7f000000   /* 2130706432 */
#define CMO_ERROR2 (LARGEID+2)
#define CMO_NULL   1
#define CMO_INT32  2
#define CMO_DATUM  3
#define CMO_STRING 4
#define CMO_MATHCAP 5
#define CMO_LIST 17
```

以下, 各 object の フォーマットを説明する. サーバ, クライアントはすべての object の CMO 形式をサポートする必要はないが, `CMO_ERROR2`, `CMO_NULL`,

CMO_INT32, CMO_STRING, CMO_MATHCAP, CMO_LIST は最も基本的なデータであり, また全てのサーバ, クライアントが実装すべき CMO データである.

CMObject Error2 は

| int32 CMO_ERROR2 | *CMObject* ob |
|---|---|

なる形で表現する. エラーの時に push する object であり, *CMObject* ob の部分に詳細なエラー情報がはいる. ob はリストであり, TCP/IP によるストリーム型接続の場合, 最初の成分はエラーを起こした OX メッセージ (後述) のシリアル番号でないといけない. シリアル番号は Integer32 で表現する.
Remark: 歴史的な理由により, エラーオブジェクトの CMO タグは *CMO_ERROR2* と名付けられている. 次の版の仕様書では CMO_ERROR に変更する予定である.

CMObject Null は

| int32 CMO_NULL |
|---|

なる形で表現する.

32 bit integer n は CMObject としては Integer32 と呼ばれ,

| int32 CMO_INT32 | int32 n |
|---|---|

なる形で表現する.

長さ n の バイト列 data は CMObject としては, Datum 型とよばれ

| int32 CMO_DATUM | int32 n | byte data[0] | byte data[1] |
|---|---|---|---|
| ··· | byte data[n-1] | | |

と表現する.

長さ n の 文字列 data は, CMObject としては, Cstring 型とよばれ

| int32 CMO_STRING | int32 n | byte data[0] | byte data[1] |
|---|---|---|---|
| ··· | byte data[n-1] | | |

と表現する. C 言語で普通用いられる, 文字列のおわりの 0 は文字列に含めない.

CMObject Mathcap は

| int32 CMO_MATHCAP | *CMObject* ob |
|---|---|

なる形で表現する. ob はリストであり少なくとも 3 つの要素をもつ. 0 番目の要素は, Integer32 で表した OpenXM protocol version number と, Cstring で表したシステム名, Server version, CPU type, その他の情報のリストである. 1 番目の要素は, システム xxx が扱うことの可能な SM タグを, Integer32 で表現したものを集めたリストである. 3 番目の要素は, システム xxx があつかうことの可能なデータ形式をあつめたリストである. 詳細は mathcap の節 5.1.2 で説明する.

長さ m のリストは

| int32 CMO_LIST | int32 m | CMObject ob[0] | ··· | CMObject ob[$m-1$] |
|---|---|---|---|---|

で表現する.

# 3 CMO の形式的表現方法

前の節で CMO の表現方法を形式的に定義せず, CMO の Primitive の表現法を説明したが, ここでは, CMO の Lisp 風表現 (Lisp-like expression) である CMOexpression および前節で説明した CMO の標準 encoding 法をもう一度説明する.

まず, CMOexpression を形式的に 拡張 BNF 記法を用いて定義しよう. タイプライタフォントでかかれた記号は終端記号を意味する. ":" は定義を意味する. "|" は" または" を意味する. { X } は X の 0 回以上の繰り返しを表す. [ x ] は X が 0 回または 1 回出現することを表す. この記法を用いると CMOexpression は次のように定義できる.

CMOexpression : (cmo_tag { expression})
expression : CMOexpression
| int32
| string
| byte

終端記号 int32 は, 32 bit integer を表す, 10 進または 16 進の数字の列である. 終端記号 string は" 文字" の列である. 終端記号 byte は 8 bit データを表す, 10 進または 16 進の数字の列である.

CMOexpression にあらわれる各要素を区切るために , (コンマ) を用いてもよい. cmo_tag は CMO_ で始まる定数である. CMOexpression で表現される object を CMObject と呼ぶ.

この表記法 CMOexpression を利用して, CMO Primitive の object を記述してみよう. Object 自体の構造を説明するため, BNF をもうすこし拡張して, 非終端記号, 終端記号名のみならず, 変数の名前も書くことにする. こうすることにより, object の意味の説明も容易になるからである. また "—" でコメントのはじまりを表すものとする.

たとえば, (CMObject の) 32 bit integer である integer32 を BNF で定義すれば,

Integer32 : (CMO_INT32, int32)

と書くのが本来の書き方による記法であるが, ここでは,

Integer32 : (CMO_INT32, $int32\ n$)
— 32 bit integer $n$ を表す.

と書くことを許すことにする. このように書くことにより, 非終端記号 Integer32 は,

Integer32 : (CMO_INT32, int32)

のように, 終端記号 CMO_INT32 と int32 を成分にもち, CMObject の (CMO_INT32, *int32* n) は, 32 bit integer $n$ を表現しているんだということが, 1 行でわかる.

この記法を用いて, 前節で導入した, Primitive の CMObject を形式的に定義しよう.

Group CMObject/Primitive requires nothing.
Error2, Null, Integer32, Datum, Cstring, Mathcap, List ∈ CMObject/Primitive.
Document of CMObject/Primitive is at `http://www.math.kobe-u.ac.jp/OpenXM`
(in English and Japanese)

| | | |
|---|---|---|
| Error2 | : | (CMO_ERROR2, *CMObject* ob) |
| Null | : | (CMO_NULL) |
| Integer32 | : | (CMO_INT32, *int32* n) |
| Datum | : | (CMO_DATUM, *int32* n, *byte* data[0], . . . , *byte* data[n-1]) |
| Cstring | : | (CMO_STRING, *int32* n, *string* s) |
| Mathcap | : | (CMO_MATHCAP, *CMObject* ob) |
| List | : | (CMO_LIST, *int32* m, *CMObject* ob[0], . . ., *CMObject* ob[m-1]) |
| | | — m is the length of the list. |

Cstring で, *string* s の部分を byte に分解すれば,

Cstring : (CMO_STRING, *int32* n, *byte* s[0], . . . , *byte* s[n-1])

となる. また, "Group CMObject/Primitive requires nothing" は, 以下は, グループ CMObject/Primitive の定義であり, このグループの CMObject を定義するのに, 要請される CMObject のグループはないことを示す. "Error2, Null, Integer32, Datum, Cstring, Mathcap, List ∈ CMObject/Primitive" は, グループ CMObject/Primitive には, Error2, Null, Integer32, Datum, Cstring なるクラスの object が属することを示す.

では, 実際のデータの表現の例をみてみよう. たとえば, 32 bit integer の 1234 は,

(CMO_INT32, 1234)

とかく. 文字列 "Hello" は

(CMO_STRING, 5, "Hello")

と書く.

CMOexpression と, CMObject の区別を理解しておくのは重要である. たとえば

(CMO_INT32, 234, "abc",(CMO_STRING))

は CMOexpression ではあるが, CMObject ではない. これは, 文法的には正しいプログラムだが, なにをやるのかは全く不明なプログラムと似ている.

さて, Open math 風 ([4]) の XML 表現法も可能であり, その場合は, 上の二つの例は次のように書く.

```
<cmo>
 <cmo_int32>
   <int32> 1234 </int32>
 </cmo_int32>

 <cmo_string>
   <int32 for="length"> 5 </int32>
   <string> "Hello" </string>
 </cmo_string>
</cmo>
```

cmo_string は次のようにあらわしてもよい.

```
<cmo>
 <cmo_string>
   <int32 for="length"> 5 </int32>
   <byte> 'H' </byte> <byte> 'e' </byte>    <byte> 'l' </byte>
   <byte> 'l' </byte> <byte> 'o' </byte>
 </cmo_string>
</cmo>
```

この場合の cmo_string の DTD による定義は次のようになる.

```
<!ELEMENT cmo_string (int32, byte*)>
```

次に, 標準 encoding 法を説明しよう. 標準 encoding 法では, cmo_tag を ネットワークバイトオーダーの 32 bit integer int32 に, その他のフィールドは, 定義に記述されているデータ型に従い, byte データ byte かまたはネットワークバイトオーダーの 32 bit integer int32 に, 変換する.

高速の通信方法を用いて効率を重視する接続の場合には, int32 を network byte order に変換する操作がおおきなオーバヘッドとなることが報告されている. 100Mbps の通信路で 12Mbytes の CMO_ZZ の転送では約 90% の時間が network byte order への変換についやされているという実験データもある. 効率を重視した encoding 法については後述する 8.3.

標準 encoding と CMOexpression の間の変換は容易である. 前節で用いたデータの表記法, たとえば,

| int32 CMO_INT32 | int32 1234 |
|---|---|

は, CMOexpression

(CMO_INT32, 1234)

の 標準 encoding 法による表現である.

(実験的) CMO および OX packets は XML 規格 [10] に準拠している. XML 規格の Attribute を binary encode するために特別なタグ

```
#define CMO_ATTRIBUTE_LIST  (LARGEID+3)
```

を用意してある.

たとえば Attribute `font="Times-Roman"` は

```
(CMO_ATTRIBUTE (CMO_LIST
                 (CMO_LIST (CMO_STRING,"font") (CMO_STRING, "Times-Roman"))))
```

と encoding される.

この特別な CMO tag `CMO_ATTRIBUTE_LIST` 以外は, XML 表現では XML のタグとして理解される.

CMO/XML で *comment*, *for* と名前がついた attributes は CMO binary expression にはエンコードされない.

# 4 OpenXM の通信モデル

われわれは, 数学プロセスがメッセージを交換しながら計算が進行していくというモデルを想定して設計をすすめている. 各プロセスはスタックマシンであり, これを OX スタックマシンとよぶ. 数学プロセスの間の通信路の確保の仕方としては以下のようにいろいろな実現方法を想定している.

1. ファイルを介して通信する.
2. Library として直接リンクして通信する.
3. TCP/IP ソケットの利用.
4. Remote Procedure call の利用.
5. JAVA RMI.
6. マルチスレッドの利用.
7. PVM ライブラリの利用.
8. MPI ライブラリの利用.

通信とはプロセス間のメッセージのやりとりである. メッセージは論理的に次のような構造をもつ:

| destination | origin | |
|---|---|---|
| extension | ox message_tag | message_body |

このメッセージを, OX message (OpenXM message object) とよぶ. OX Message はトップレベルのメッセージ object であり, 仕様書では, さまざま

なグループに属する object が登場する. グループ名は, たとえば, OX Message/TCPIP/Primitive などと書く. message_body の部分の仕様は, OX Message の上位に位置する部分であり, SMobject または CMObject がくる. これらの object はタグをもち, そのタグ定数は SM_ または CMO_ ではじまる. SMobject は, スタックマシンコマンドメッセージ object であり, やはり, グループ分けされている. 各グループ名は, SMobject/Primitive, SMobject/Basic などと書く. SMobject の構造はサーバスタックマシンの節で詳しく説明する. CMObject についてはすでに Primitive の CMObject の説明をしたが, あとで CMObject レベル 1 の説明をする. OX Message の ox message_tag の定数は OX_ で始まる.

## 4.1 OX Message の 表現方法

OpenXM で各プロセスは

(OXexpression を理解するスタックマシン) + (xxx 言語で動くエンジン)

なるハイブリッド構成である. このプロセスを, OX スタックマシンと呼ぶ. ここでは, OX スタックマシンとやりとりするメッセージである, OX Message を表現するための OXexpression, および, スタックマシンの operator に対応する, SMobject を表現するための SMexpression を定義しよう. OX Message を スタックマシンコマンド, SMobject を スタックマシンオペレータともよぶ.

### 4.1.1 OX メッセージの表現法 (Lisp 風)

| | | |
|---|---|---|
| OXexpression | : | (OX_tag [ expression]) |
| expression | : | SMexpression |
| | | \| CMOexpression |
| SMexpression | : | (SM_tag {CMOexpression}) |

expression の各要素を区切るために , (コンマ) を用いてもよい. OX_tag は OX_ で始まる定数である. SM_tag は SM_ で始まるスタックマシンオペレータを識別する定数である. 発信元 AAA, 受信先 BBB を書く必要があるときは, From AAA, To BBB, を OXexpression の前に書く. 必要なければ省略する.

たとえば, あとで説明する, CMO string “Hello” を スタックにプッシュする表現は次のように書く:

(OX_DATA, (CMO_STRING, 5, ”Hello”))

(OX_DATA, (CMO_STRING, 4, ”hoge”))

(OX_COMMAND, SM_executeStringByLocalParser)

標準エンコード法では, それぞれのタグは 32 bit (4 bytes) のネットワークバイトオーダでの整数として表現される.

### 4.1.2 OX メッセージの表現 (XML)

OX メッセージは XML で表現することも可能である. 次に例を示す.

```
<ox>
   <ox_data>
      <ox_serial> <int32> 0 </int32> </ox_serial>
      <cmo>  <cmo_string>
                 <int32 for="length"> 5 </int32>
                 <string> "Hello" </string>
             </cmo_string>
      </cmo>
   </ox_data>
   <ox_command>
       <ox_serial> <int32> 1 </int32> </ox_serial>
       <sm_popCMO/>
   </ox_command>
</ox>
```

## 4.2 OXexpression の 標準 encoding と TCP/IP ソケットによる実装法

通信の実現方法は通信路のとりかたによりかわるが, 論理構造は統一的にあつかわないといけない. OXexpression はその論理構造を記述している.

ここでは OXexpression の標準 encoding の概略を説明する. この encoding 法は TCP/IP ソケット用の encoding 法として現在存在しているサーバに使用されている. さらに OX スタックマシンの計算状態を制御するための, コントロールメッセージについても説明する.

`destination`, `origin` の部分は, ソケットによる peer to peer の接続なので省略する. `extension` フィールドは `message_tag` フィールドの次にくる. `extension` フィールドは OX パケットのシリアル番号がはいる. シリアル番号は `int32` である. この番号は, サーバがエラーを起こした場合, エラーをおこした, OX パケットの番号を戻すのに主に利用される. 以下 `extension` フィールドは, `message_tag` のに含まれると理解し `extension` フィールドは省略する. したがってパケットは次のように記述する

| ox message_tag | message_body |
|---|---|

が, もっとこまかく見ると,

| ox message_tag, serial number | message_body |
|---|---|

となっている.

グループ OX Message/TCPIP/Primitive の `ox message_tag` としては次のものが用意されている.

@plugin/oxMessageTag.h

```
#define   OX_COMMAND         513
#define   OX_DATA            514

#define   OX_DATA_WITH_LENGTH  521
#define   OX_DATA_OPENMATH_XML 523
#define   OX_DATA_OPENMATH_BINARY 524
#define   OX_DATA_MP           525

#define   OX_SYNC_BALL       515
#define   OX_NOTIFY          516
```

通信路は 2 つ用意する. 1 番目の通信路は OX_COMMAND および OX_DATA がながれる. 2 番目の通信路 (control と呼ぶ) には, OX_COMMAND およびそれに続くコントロールコマンド SM_control_* またはコントロール関係のデータ, つまり header OX_DATA ではじまりそれに続く CMO データがながれる. これらをコントロールメッセージおよびコントロールメッセージの結果メッセージと呼ぶ. サンプルサーバでは, この 2 つの通信路を, 2 つのポートを用いて実現している.

OX_COMMAND メッセージは次の形のパケットで表現される:

| OX_COMMAND | int32 function_id |
|---|---|
| *message_tag* | *message_body* |

, (OX_COMMAND, (SM_*))

OX_DATA メッセージは次の形のパケットで表現される:

| OX_DATA | CMO data |
|---|---|
| *message_tag* | *message_body* |

, (OX_DATA, *CMObject* data)

コントロールメッセージは次の形のパケットで表現される:

| OX_COMMAND | int32 function_id |
|---|---|

, (OX_COMMAND,(SM_control_*))

コントロールメッセージは, 計算を中断したい, debug 用の スレッドを起動する, debug モードを抜けたい, などの用途に利用する.

コントロールメッセージの結果メッセージは次の形のパケットで表現される:

| OX_DATA | CMO_INT32 | int32 data |
|---|---|---|

, (OX_DATA, *Integer32* n)

int32 function_id の部分に, サーバスタックマシン の operator に対応する番号がはいる. グループ SMobject/Primitive および SMobject/Basic に属するタグとして以下のものがある.

@plugin/oxFunctionId.h

```
#define SM_popSerializedLocalObject 258
#define SM_popCMO 262
#define SM_popString 263

#define SM_mathcap 264
#define SM_pops 265
#define SM_setName 266
#define SM_evalName 267
```

```
#define SM_executeStringByLocalParser 268
#define SM_executeFunction 269
#define SM_beginBlock  270
#define SM_endBlock    271
#define SM_shutdown    272
#define SM_setMathCap  273
#define SM_executeStringByLocalParserInBatchMode 274
#define SM_getsp       275
#define SM_dupErrors   276


#define SM_control_kill 1024
#define SM_control_reset_connection  1030
```

たとえば,

(OX_COMMAND, SM_pops)

は

| int32 513 | int32 265 |
|---|---|

とエンコードされる.

operator の詳細は次の節で説明する. これらの定数の名前はインプリメントのとき短縮形で表現してもよい.

# 5 OX スタックマシン

この節では, OX スタックマシン operator の説明 (TCP/IP ソケット上での標準 encoding 法 を用いる), および, サンプルサーバとリンクする場合または open XM ライブラリとしてリンクして使用する場合のための C の関数の仕様を説明する.

説明の前に, OX サーバスタックマシンの動作の原則を説明しておく. サーバスタックマシンは, SM_pop* 系のスタックマシンコマンドがこないかぎり, 自発的にメッセージを送信することはない. この原則に基づいて分散計算のプログラミングをおこなう. イベントドリブンなプログラム法とはちがうことに注意しよう.

## 5.1 サーバスタックマシン

サンプルサーバである oxserver00.c は以下の仕様の C の関数を用意して, nullstackmachine.c を置き換えれば一応動作するはずである.

### 5.1.1 サーバスタックマシンのグループ SMobject/Primitive に属するオペレータ

サーバスタックマシンは最低で 1 本のスタック

```
Object xxx_OperandStack[SIZE];
```

をもつ. ここで, Object はそのシステム固有の Object 型で構わない. CMObject は各サーバ固有のローカルオブジェクトに変換してスタックへプッシュしてよい. ただし変換, 逆変換を合成したものは恒等写像であることがのぞましい. CMObject をどのように (local) Object に変換するか, Object が受け付けるメッセージの定義は, 各システムが独自にきめて文書化しておくものとする. つまりすべてのメッセージは, private である. たとえば, add のような基本的な メッセージにたいしても, OX スタックマシンはなにもきめていない. 将来的には open math [4] のように CMObject に対する最大公約数的なメッセージの仕様を content dictionary (CD) の形で定義したい.

以下, xxx_ は誤解の恐れがないときは省略する. xxx_ は local サーバシステムに固有の識別子である. Asir の場合は Asir_ を用いる. kan/sm1 の場合は Sm1_ を用いる. 関数名, タグ名は長いので省略形を用いてもよい.

以下では次のようにパケットを記述する. 各フィールドは, データ型 データ なる形式で書く. たとえば, int32 OX_DATA は 32 bit network byte order の数字 OX_DATA という意味である. "イタリックで書かれているフィールドは, 定義が別のところでなされているか解釈に誤解のないような自然言語で説明されている object を表す." たとえば, *String commandName* は, String データ型の local object *commandName* を意味する. (サーバスタックマシン上の object は, CMO 形式の object とは限らないことに注意. CMO 形式で書いてあっても, それはサーバスタックマシンの local 形式でスタック上にあると解釈して下さい.)

すべてのサーバスタックマシンは以下の操作を実装していないといけない. 各操作に対し, その前後におけるスタックの状態を示す. 図において, 右端のオブジェクトがスタックのトップに対応する.

1. CMObject/Primitive の CMO データのうち必須のもの, CMO_ERROR2, CMO_NULL, CMO_INT32, CMO_STRING, CMO_LIST がおくられて来た場合それをスタックに push する. たとえば, CMO_NULL あるいは CMO_String の場合次のようになる.

   Request:

   | int32 OX_DATA | int32 CMO_NULL |
   |---|---|

   Stack after the request:

   | *NULL* |
   |---|

   Output: none.

   Request:

   | int32 OX_DATA | int32 CMO_String | int32 size | byte s1 | ··· | byte ssize |
   |---|---|---|---|---|---|

   Stack after the request:

   | *String s* |
   |---|

   Output: none.

   CMO データの受け取りに失敗した時のみ

   | int32 OX_DATA | int32 CMO_ERROR2 | *CMObject* ob |
   |---|---|---|

   をスタックへ push する. 現在のところ, ob には,
   [*Integer32* OX パケット番号, *Integer32* エラー番号, *CMObject* optional 情報]
   なるリストを入れる (CMO 形式でかいてあるが, これはサーバ独自の形式でよい. CMO として送出されるときこのような形式でないといけないという意味である.)

2. SM_mathcap

   このサーバの mathcap をもどす (termcap のまね). サーバのタイプ, サーバスタックマシンの能力を知ることができる. C 言語で実装する場合は, mathcap の構造体をシステム毎にきめるものとし, この関数はその構造体へのポインタを戻す. (open sm1 では struct mathcap を用いている.

   @plugin/mathcap.h)

   Request:

   | int32 OX_COMMAND | int32 SM_mathcap |
   |---|---|

   Stack after the request:

   | int32 OX_DATA | *Mathcap* mathCapOb |
   |---|---|

   Output: none.

3. SM_setMathcap

   受け取った Mathcap m を自分のシステムに設定して, 相手側が理解不能な CMO をおくらないようにする. C 言語で実装する場合は, mathcap の構造体をシステム毎にきめるものとし, この関数はその構造体へのポインタを引数とする. (open sm1 では struct mathcap を用いている.

   @plugin/mathcap.h)

   Stack before the request:

   | *Mathcap m* |
   |---|

   Request:

   | int32 OX_DATA | *Mathcap* m |
   |---|---|
   | int32 OX_COMMAND | int32 SM_setMathcap |

   Output: none. 注意: mathcap は一般にクライアント主体で設定する. クライアントがサーバに SM_mathcap をおくり, サーバ側の mathcap を得る. それを, クライアントはそのサーバに付随した mathcap として設定する. 次に, クライアントはサーバに自分の mathcap を SM_setMathcap でおくり, 自分の mathcap を設定させる.

4. SM_executeStringByLocalParser

   文字列 s を stack から pop し, その文字列をシステム固有の文法 (サーバスタックマシンの組み込みローカル言語) にしたがったコマンドとして実行する. コマンドの実行の結果の最後に戻り値があるときは, OperandStack に戻り値を push する. OpenXM では, 現在のところ関数名の標準化はおこなっていない. この関数および popString の機能を実現すれば, 最低限の open XM のサーバになれる. 実装では, まずこの二つの関数の機能を実現すべきである.

   Stack before the request:

   | *String commandString* |
   |---|

   Request:

   | int32 OX_COMMAND | int32 SM_executeStringByLocalParser |
   |---|---|

   Output: none. 参考: 実行前のスタックのデータは, *String commandString* なる local stack machine の object としてスタック上にあるが, TCP/IP の通信路では, 次のようなデータがまずながれて *commandName* がスタックに push される:

   | int32 OX_DATA | int32 CMO_string | *size and the string commandString* |
   |---|---|---|

5. SM_executeStringByLocalParserInBatchMode

   スタックに副作用がない (スタックにたいしてなんの操作もしない) ことを除き上とまったく同じ関数である. エラーの時のみ, Error2 Object をスタックへプッシュする.

6. SM_popString

   OperandStack より Object を pop し, それを xxx の出力規則にしたがい文字列型に変換して送信する. スタックが空のときは, (char *)NULL を戻す. 文字列は TCP/IP stream へ CMO のデータとして送信する. エラーの場合は CMO_ERROR2 を戻すべきである.

   Stack before the request:

   | *Object* |
   |---|

   Request:

   | int32 OX_COMMAND | int32 SM_popString |
   |---|---|

   Output:

   | int32 OX_DATA | int32 CMO_STRING | *size and the string s* |
   |---|---|---|

7. SM_getsp

   現在のスタックポインタの位置をもどす. スタート時点での位置は 0 であり, object が push されたばあい, 1 づつ増えるものとする.

   Stack before the request:

   | *Object* |
   |---|

   Request:

   | int32 OX_COMMAND | int32 SM_getsp |
   |---|---|

   Stack after the request:

   | int32 OX_DATA | int32 CMO_INT32 | *stack pointer value* |
   |---|---|---|

   Output: none.

8. SM_dupErrors

   スタック上のエラーオブジェクトをリストにして戻す. スタック自体は変化させない.

   Request:

   | int32 OX_COMMAND | int32 SM_dupErrors |
   |---|---|

   Stack after the request:

   | int32 OX_DATA | *CMObject* a list of errors |
   |---|---|

   Output: none.

9. SM_pushCMOtag

   スタックの先頭オブジェクトが CMO に変換された場合の CMO tag を CMO_INT32 としてスタックへ push する. 先頭オブジェクトはスタックに留まる. 先頭オブジェクトを CMO へ変換する方法が無い場合は, エラー object を積む.

   Request:

   | int32 OX_COMMAND | int32 SM_pushCMOtag |
   |---|---|

   Stack after the request:

   | int32 OX_DATA | *CMO_INT32* tag |
   |---|---|

   Output: none.

### 5.1.2 MathCap

例: mathcap の問い合わせに対して, ox_sm1 は次のように答える.

```
Class.mathcap
 [ [199909080 , $Ox_system=ox_sm1.plain$ , $Version=2.990911$ ,
    $HOSTTYPE=i386$ ]  ,
   [262 , 263 , 264 , 265 , 266 , 268 , 269 , 272 , 273 , 275 , 276 ]  ,
   [[514] , [2130706434 , 1 , 2 , 4 , 5 , 17 , 19 , 20 , 22 , 23 , 24 ,
             25 , 26 , 30 , 31 , 60 , 61 , 27 , 33 , 40 , 34 ]]]
```

mathcap は 3 つの要素をもつリストである. まづ, 最初の要素 (これもリスト) を見よう. リストの先頭の数字は OpenXM プロトコルのバージョンである. Ox_system は openXM システム名である. 読み込むライブラリがちがっていて, 関数名 (または シンボル) の意味がちがうときはこの名前もかえる. たとえば, open math の basic content dictionary 対応の関数定義マクロを読みこんだ sm1 は, ox_sm1_basicCD なる名前にする. HOSTTYPE 値は, CPU の種類をあらわし unix では環境変数$HOSTTYPE の値である. 2 番目の要素は 利用可能な SM コマンドをあつめたリストである. 3 番目のリストは, 処理可能な数学データの形式, および CMO の場合なら処理可能な CMO のタグのリストが続く. 上の例では, 514 は OX_DATA をあらわし, 数学データのフォマットは (サイズ情報なしの) CMO であることを示す.
OpenXM/XML expression of the example above:

```
<cmo_mathcap>
   <cmo_list for="mathcap">

     <cmo_list>
        <int32 for="length"> 4 </int32>
        <cmo_int32 for="Protocol version">  001001003 </cmo_int32>
        <cmo_string for="system name"> Ox_system=ox_sm1.plain  </cmo_string>
        <cmo_string for="system version"> Version=2.990911  </cmo_string>
        <cmo_string for="hosttype"> HOSTTYPE=i386  </cmo_string>
     </cmo_list>

     <cmo_list for="Available SM tags">
        <int32 for="length"> 11 </int32>
        <cmo_int32> 262 </cmo_int32>
        <cmo_int32> 263 </cmo_int32>
         ...
     </cmo_list>

     <cmo_list for="Available OX_DATA tags">
        <int32 for="length"> 2 </int32>
        <cmo_list for="OX_DATA tag">
           <int32 for="length"> 1 </int32>
           <cmo_int32 comment="OX_DATA">  514 </cmo_int32>
```

```
        </cmo_list>
        <cmo_list for="Available CMO tags">
           <int32 for="length"> 21 </int32>
           <cmo_int32 comment="CMO_ERROR2">  2130706434 </cmo_int32>
           <cmo_int32 comment="CMO_NULL"> 1 </cmo_int32>
            ....
        </cmo_list>

     </cmo_list>
   </cmo_list>
</cmo_mathcap>
```

### 5.1.3 例

例: message_body の実例をあげる. シリアル番号部は除いてある.

1. executeStringByLocalParser("12345 ;"); は次のようなパケットに変換される. 各数字は 16 進 1 バイトをあらわす. xx(yy) のなかの (yy) は対応するアスキーコードをあわらす.

   ```
   0   0   2   2   0   0   0   4   0   0   0   7
   31(1)  32(2)  33(3)  34(4)  35(5)  20  3b(;)
   0   0   2   1   0   0   1   c
   ```

   それぞれのデータの意味は次のとおりである.

   ```
   0   0   2   2  (OX_DATA) 0   0   0   4  (CMO_STRING)
   0   0   0   7  (size)
   31(1)  32(2)  33(3)  34(4)  35(5)  20  3b(;)   (data)
   0   0   2   1  (OX_COMMAND)
   0   0   1   c  (SM_executeStringByLocalParser)
   ```

   これを OXexpression で表記すると次のようになる.

   $$(\text{OX_DATA}, (\text{CMO_STRING}, 7, \text{"12345 ;"}))$$

   $$(\text{OX_COMMAND}, (\text{SM_executeStringByLocalParser}))$$

2. popString() を要請するメッセージ:

   ```
   0   0   2   1  (OX_COMMAND)
   0   0   1   7  (SM_popString)
   ```

   OXexpression では (OX_COMMAND, (SM_popString)).
   これにたいして次の返答メッセージがくる.

```
0   0   2   2   (OX_DATA)
0   0   0   4   (CMO_STRING) 0   0   0   5  (size)
31(1)  32(2)  33(3)  34(4)  35(5)
```

OXexpression でかくと, (OX_DATA, (CMO_STRING, 7, "12345 ;")).

### 5.1.4 グループ SMobject/Basic に属するオペレータ

1. `SM_pops`

   operand stack より, $n$ 個の元 (*obj1, obj2, ..., objn* を pop して捨てる.

   Stack before the request:

   | *obj1* | *obj2* | $\cdots$ | *objn* | *Integer32 n* |
   |---|---|---|---|---|

   Request:

   | `int32 OX_COMMAND` | `int32 SM_pops` |
   |---|---|

   Output: none.

2. `int SM_setName`

   `OperandStack` より *name* を pop し, つぎに `OperandStack` より *obj* を pop し, それを現在の名前空間で変数 *name* に bind する. 正常終了なら 0 を, 異常終了なら -1 をもどす. TCP/IP による通信では, 異常終了の時のみ, `CMO_ERROR2` を stack へ push する.

   Stack before the request:

   | *obj* | *String name* |
   |---|---|

   Request:

   | `int32 OX_COMMAND` | `int32 SM_setName` |
   |---|---|

   Output: none.

3. `SM_evalName`

   現在の名前空間で変数 *name* を評価する. 評価の結果 *OutputObj* をスタックへ戻す. 関数自体は正常終了なら 0 を, 異常終了なら -1 をもどす. TCP/IP の場合, 異常終了の場合のみ `CMO_ERROR2` を stack へ push する.

   Stack before the request:

   | *String name* |
   |---|

   Request:

   | `int32 OX_COMMAND` | `int32 SM_evalName` |
   |---|---|

   Stack after the request:

   | *OutputObj* |
   |---|

   Output: none.

4. `SM_executeFunction`

   スタックより $n$ 個のデータを pop して, サーバのローカル関数 $s$ を実行する. エラーのときのみ `CMO_ERROR2` を stack へ push する.

   Stack before the request:

   | *objn* | $\cdots$ | *obj1* | *INT32 n* | *String s* |
   |---|---|---|---|---|

   Request:

   | `int32 OX_COMMAND` | `int32 SM_executeFunction` |
   |---|---|

   Stack after the request: 関数実行の結果.

   Output: none.

5. `SM_popSerializedLocalObject`

   スタックより pop した object を local 形式で serialization して OX message として stream へ出力する. OX message tag としては, local 形式に対応したものが定義されていることが必要である. この関数はおもに, homogeneous な分散システムで用いる.

6. `SM_popCMO`

   `OperandStack` より object を pop し CMO 形式の serialized object を stream へ header `OX_DATA` をつけてながす.

   Request:

   | `int32 OX_COMMAND` | `int32 OX_popCMO` |
   |---|---|

   Output:

   | `int32 OX_DATA` | *Serialized CMO* |
   |---|---|

7. `SM_executeFunctionWithOptionalArgument`

   スタックより $n$ 個のデータおよび一つの optional 引数 *opt* を pop して, サーバのローカル関数 $s$ を実行する. エラーのときのみ `CMO_ERROR2` を stack へ push する. *opt* はキーワードと値のリストのリストである. キーワードは文字列型である.

   Stack before the request:

   | *objn* | $\cdots$ | *obj1* | *INT32 n* | *Obj opt* | *String s* |
   |---|---|---|---|---|---|

   Request:

   | `int32 OX_COMMAND` | `int32 SM_executeFunctionWithOptionalArgument` |
   |---|---|

   Stack after the request: 関数実行の結果.

   Output: none.

   Example of *opt* : `(("p", 13),("vars",("x","y")))`

   [Added in 2004-3-8]

# 6 現在検討中の機能

## 6.1 OX DATA with Length の構造

Digital signature 付の `OX_DATA` はヘッダ `OX_DATA_WITH_LENGTH` で, はじまり, CMO がきて, それから, 終りのマークおよびデジタル署名がくる. この形のデータを *secured OX DATA* と呼ぶ.

```
#define OX_SECURED_DATA                 521
```

| `int32 OX_DATA_WITH_LENGTH` | `int32` serial | `int32` size | *CMObject* o | *tail* |
|---|---|---|---|---|

`size` フィールドに 値 -1 が入っている場合この情報を無視する.

*tail* は次のように定義する.

| `int32 CMO_START_SIGNATURE` | `int32` size | *signature* |
|---|---|---|

ここで, `size` はバイト列 *signature* 部の長さ. *signature* は, Hash 関数を用いた, *CMO data* にたいする, デジタル署名をいれ, 不正な serialized object を検出する. Tail の `size` フィールドが 0 の場合, デジタル署名部はない.

クライアント, サーバの実装には次の 4 つの選択がある.

1. OX_DATA のみを用いて CMObject を送る (mathcap 付).
2. OX_SECURED_DATA のみを用いて CMObject を送る.
3. OX_DATA および OX_SECURED_DATA を混在して使用できるようにする.
4. OX_DATA のみを用いてかつ mathcap も用いない (一番高速).

1 の短所は, mathcap の実装が不十分で, 理解できない CMObject をうけとると, 以後の CMObject は理解できなくなる. 1 の長所は, 長さフィールドの計算をおこなわないため, データを送り出す側に負担がかからないことである. 2 の長所は, mathcap の実装が不十分で, 理解できない CMObject をうけとっても, 長さフィールドを用いて, 通信回線のダウンを防ぐことができる. 2 の短所は, 長さフィールドの計算をおこなうため, データを送り出す側に負担がかかることである.

現在のすべてのサンプルサーバは, 1, 4 のみを実装している.

mathcap の交換はセッションの開始時点で必須という訳ではないことに注意されたい. たとえば, モード 4 で通信して, それから, mathcap を交換して, モード 1 へ移行することも可能なように実装すべきである.

### 6.2 サーバスタックマシンはローカルな拡張機能をもってよい

```
#define  CMO_PRIVATE   0x7fff0000  /* 2147418112 */
```

CMO_PRIVATE = OX_PRIVATE = SM_PRIVATE よりはじまる, 0x10000 個の ID は, private 領域として予約されている. 各サーバの開発者間で表現方法につきまだ合意がないような, CMObject, OX Message, SMobject を表現するために使用する.

### 6.3 MathLink や OpenMath などの通信規約の実装

Open asir, open sm1 と Mathematica の用いている MathLink や Open Math ([4]) プロトコルを変換するためのライブラリやサーバを用意すれば, asir や kan/sm1 の内部構造を理解することなしに, これらに準拠したシステムと通信できる.

### 6.4 共通スタックマシン言語

CMO に付随した, たとえば, Integer 32 に対する, add, sub, mul, などの基本的な計算は SM_executeFunction で全てのシステムで実行できるように検討している. スタックマシンの制御構造についても検討している.

## 7 Session Management

### 7.1 Control server

OpenXM では, 次に述べるような単純かつロバストなサーバの制御方法を採用している.

OpenXM サーバは論理的に 2 つの I/O channel をもつ: 一方は計算データ用であり, 他方は計算制御用である. 制御 channel はサーバを制御するためのコマンドを送るために使われる. サンプルサーバ (oxmain.c) では, そのようなコントロールメッセージは別のプロセスが行っている. 以下, そのプロセスをコントロールサーバと呼ぶ. これに対して, 計算用サーバをエンジンと呼ぶ. コントロールサーバとエンジンは同一のマシン上で動作する. このため, コントロールサーバからエンジンに signal を送ることは容易である. コントロールサーバ自体も OX スタックマシンであり SM_control_* コマンドを受け取る. それらはエンジンへの signal 送信, engine process の終了などの request のためのコマンドである.

## 7.2 New OpenXM control servers

OpenXM RFC 101 Draft を見よ http://www.math.kobe-u.ac.jp/OpenXM/OpenXM-RFC.html.

## 7.3 OpenXM reset protocol

クライアントはコントロールサーバ経由でいつでもエンジンに signal を送ることができる. しかし, I/O 操作は通常バッファリングされているため, トラブルが生ずる場合がある. エンジンを安全にリセットするため次が必要である.

1. 全ての OX メッセージは Java の意味で synchronized object である.
2. エンジンのリセット後に送られるクライアントからの計算要求メッセージとエンジンからの返答が正しく対応していなければならない.

SM_control_reset_connection は, エンジンの安全なリセットを行う一連の手続きを開始するための SM コマンドである. クライアントから SM_control_reset_connection がコントロールサーバに送られると, コントロールサーバは SIGUSR1 をエンジンに送る. 以後の手続きは次の通りである.

クライアント側

1. SM_control_reset_connection をコントロールサーバに送った後, クライアントはリセット状態に入る. リセット状態では, OX_SYNC_BALL を受け取るまですべてのメッセージを読みとばす.
2. OX_SYNC_BALL を受け取ったあと, クライアントは OX_SYNC_BALL をエンジンに送り, 通常状態に戻る.

エンジン側

1. SIGUSR1 をコントロールサーバから受け取ったら, エンジンはリセット状態に入る. OX_SYNC_BALL をクライアントに送る. この時点でクライアントは既にリセット状態にあるので, この送信がブロックされることはない.
2. エンジンは OX_SYNC_BALL を受け取るまですべてのメッセージを読みとばす. OX_SYNC_BALL を受け取ったら通常状態に戻る.

図 1 はデータの流れを示す. OX_SYNC_BALL は特殊な OX メッセージであり, I/O stream に残るデータの終りを示す. OX_SYNC_BALL を読んだ後, それぞれの stream は空であり, 後に続くクライアントからのリクエストと, エンジンからの返答が正しく対応する.

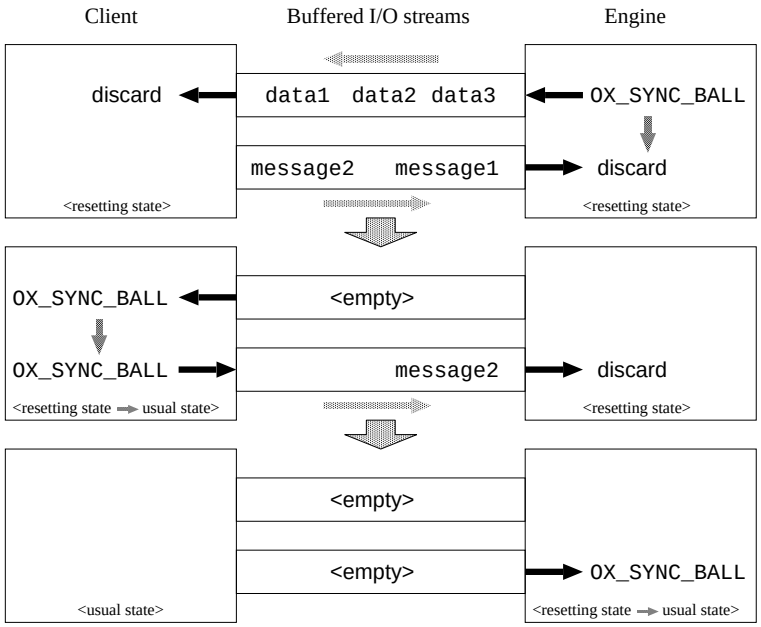



Figure 1: OpenXM reset procedure

## 7.4 Control message (SMObject/TCPIP/Control)

1. SM_control_reset_connection

   コントロールサーバに, SIGUSR1 をエンジンに送るよう要求する.

   Request: | int32 OX_COMMAND | int32 SM_control_reset_connection |

   Result: | int32 OX_DATA | CMO_INT32 result |

2. SM_control_kill

   サーバはこのメッセージを受信したらすべてのファイルをクローズして終了する.

   Request: | int32 OX_COMMAND | int32 SM_control_kill |

   Result: none.

例: (シリアル番号は省略してある.)

```
0  0 2 01 (OX_COMMAND)
0  0 4 06 (SM_control_reset_connection)
```

Reset に対する返事.

```
0  0 2 02 (OX_DATA)
0  0 0  2 (CMO_INT32)
0  0 0  0 (  0   )
```

第 1 のチャンネルでは次の OX_SYNC_BALL が交換されて同期が取られる.

```
0   0   2   03   (OX_SYNC_BALL)
```

### 7.5 Notification from servers

OpenXM サーバは, 可能であるかぎり寡黙である. たとえばエラーをおこしても, エラーはサーバのエンジンスタックにつまれるだけであり, サーバはクライアントが pop_cmo メッセージをおくらないかぎり何も送信しない.

OpenXM はこの原則をやぶる例外的な方法を一つ用意している. コントロールサーバは, OX_NOTIFY ヘッダおよびそれにつづく OX_DATA パケットを送ることができる. この機能は mathcap で禁止することも可能である.

この機能をどのように使うか例をあげて説明しよう. Asir の ox_plot サーバは, quit ボタンをもっている. quit ボタンがおされると canvas が消滅するが, エンジン自体は存在をつづける. この状態で描画命令がくると, エンジンスタックに, "canvas does not exist" というエラーがつまれる. エンジンがこのエラーが生じたことを緊急に知らせたいときに OX_NOTIFY を用いる.

ここで, OX_NOTIFY をおくるのは, コントロールプロセスであることに注意しよう. したがってエンジンはなんらかの方法で, コントロールサーバに OX_NOTIFY をおくることを依頼しないといけない. この方法は, OS によりいろいろな方法が可能だか, たとえば, unix ではファイル /tmp/.ox_notify.pid に touch することでこれを一つの実現することが可能である. ここで pid はエンジンのプロセス番号である. コントロールサーバはファイル /tmp/.ox_notify.pid が touch されたことを検出したら, クライアントに OX_NOTIFY パケットおよび OX_DATA で cmo_null を送る. エンジンはファイルを用いてコントロールサーバに急を知らせる以外に, 共有メモリやシグナルを用いてしらせてもよい.

## 8 TCP/IP でのセッションのスタート

### 8.1 Standard I/O on OX servers

サーバの実装を容易にするために, サーバは, 起動時に file descriptor 3, 4 がそれぞれクライアントからの入力, クライアントへの出力用に open されていると仮定する. すなわち, サーバ自体が通信を成立させるための socket 操作を行う必要はない. 但し, OX message の効率よい送受信のために, data のバッファリングを行うことはサーバ側の責任である. UNIX においては, file descriptor への buffer の attach は fdopen() で行うことができる.

## 8.2 Launcher

サーバは, クライアントから起動されるが, TCP/IP 接続のためには socket の生成から始まる一連の操作が必要である. 一方で, すでに述べたように, サーバ自体はそういう機能はもたない. launcher は, この両者の橋渡しをする application である. launcher の働きは次の通りである.

1. launcher はクライアントが起動する. launcher には使用する port 番号, サーバ application のファイル名が伝えられる.
2. client はまず launcher と通信を成立させる. この際, one time password が用いられる場合もある.
3. launcher は新しいプロセスを起動し, そのプロセスにおいて client と通信を成立させる. そして, その通信路が file descriptor の 3, 4 となるようにセットし, サーバを exec する.

以上の操作のあと, launcher はコントロールサーバとなり, 自身が作ったしたサーバの制御を行う.

## 8.3 byte order の決定

コントロールプロセス, 計算プロセスともに, 起動直後に 1 byte のデータを書き出し flush する. そのあと, 1 byte のデータを読み込む. クライアントはコントロールプロセス, 計算プロセスにつながるファイルディスクリプタの両方からまず 1 byte のデータを読む. そのあと 1 byte のデータを書き出し flush する. 1 byte のデータは, `0`, `1`, `FF` のどれかであり, `0` は以下の通信において `int32` をおくるのに, network byte order を使用したい, `1` は以下の通信において `int32` をおくるのに, little endian を使用したい, `FF` は以下の通信において `int32` をおくるのに, big endian を使用したい, という意味である. 両者の希望が一致しない場合はつねに `0` (network byte order) を使用する. Network byte order しか実装していないシステムでは, `0` をおくればよい. ただし効率が問題となる通信において, network byte order への変換はおおきなボトルネックとなることがあることを了解しておくべきである.

64 bit machine double および 128 bit machine double (浮動小数点数) のやりとりに際しても, 上と同様な方法をもちいるものとする. すなわち `double64 *` を `int32 *` にキャストしてから, 中身を `int32` と同様の処理方法で送受信する. 例については `CMO_64BIT_MACHINE_DOUBLE` を見よ.

```
#define OX_BYTE_NETWORK_BYTE_ORDER    0
#define OX_BYTE_LITTLE_ENDIAN         1
#define OX_BYTE_BIG_ENDIAN         0xff
```

## 8.4 launcher の例 : `ox`

注意: `OpenXM/src/kxx` に含まれる, `ox` (コントロールプロセス, 計算プロセスをたちあげるローンチャ) は, 標準で One Time Password の機能をもっています. この機能を OFF にするには `-insecure` option を使用して下さい. One Time

Password は 0 で終了するバイト列であり, コントロール, 計算双方のプロセスをたちあげるまえに, ox はコントロール, 計算双方のプロセスに対応する, ふたつのポートに One Time Password バイト列を送出しています.

ox (ソースは oxmain.c, kan96xx/plugin/oxmisc.c) においては oxTellMyByteOrder() が, サーバについての byte order 情報の送出, 読み込みをやっている. クライアントについては, oxSetByteOrder() が, byte order 情報の読み込み, 送出をおこなっている.

One time パスワードは安全な通信路で配送される必要があります. また, 現在の ox の実装では, One time パスワードをサーバ, クライアントに login している人はすべて見ることができますので, サーバ, クライアントには悪意のある人はいないと仮定しないといけません. One time パスワードを安全に配送し, リモートマシンの ox を立ち上げるにはたとえば ssh を -f オプションを用いて使用します.

以下は sm1 での実装, 動作例です. ここでは, yama の sm1 より dc1 の ox を立ち上げています.

```
yama% sm1
sm1>(ox.sm1) run ;
ox.sm1, --- open sm1 protocol module 10/1,1999  (C) N.Takayama. oxhelp for help
sm1>[(dc1.math.kobe-u.ac.jp) (taka)] sm1connectr-ssh /ox.ccc set ;
Hello from open. serverName is yama.math.kobe-u.ac.jp and portnumber is 0
Done the initialization. port =1024
Hello from open. serverName is yama.math.kobe-u.ac.jp and portnumber is 0
Done the initialization. port =1025
[    4 , 1025 , 3 , 1024 ]
Executing the command : ssh -f dc1.math.kobe-u.ac.jp -l taka
"/home/taka/OpenXM/bin/oxlog /usr/X11R6/bin/xterm -icon
-e /home/taka/OpenXM/bin/ox -reverse -ox /home/taka/OpenXM/bin/ox_sm1
-host yama.math.kobe-u.ac.jp -data 1025 -control 1024 -pass 518158401   "
[
taka@dc1.math.kobe-u.ac.jp's password:
Trying to accept... Accepted.
Trying to accept... Accepted.

Control port 1024 : Connected.

Stream port 1025 : Connected.
Byte order for control process is network byte order.
Byte order for engine process is network byte order.
```

### 8.5 OX サーバの利用例

ox サーバを TCP/IP で $C$ より利用する例は OpenXM/doc/oxlib/test1-tcp.c にある.

## 9 オブジェクトの文字列表現

文字列表現は, システム xxx のマニュアルに記述されている一次元的入出力形式にしたがう.

# 10 数, 多項式 の CMO 表現

@../SSkan/plugin/cmotag.h

```
#define     CMO_MONOMIAL32  19
#define     CMO_ZZ          20
#define     CMO_QQ          21
#define     CMO_ZERO        22
#define     CMO_DMS_GENERIC  24
#define     CMO_DMS_OF_N_VARIABLES  25
#define     CMO_RING_BY_NAME   26
#define     CMO_DISTRIBUTED_POLYNOMIAL 31
#define     CMO_RATIONAL       34


#define     CMO_INDETERMINATE  60
#define     CMO_TREE           61
#define     CMO_LAMBDA         62    /* for function definition */
```

以下, グループ CMObject/Basic, CMObject/Tree および CMObject/DistributedPolynomial に属する CMObject の形式を説明する.
OpenXM/src/ox_toolkit にある bconv をもちいると CMO expression を binary format に変換できるので, これを参考にするといい.
Example:

```
bash$ ./bconv
> (CMO_ZZ,123123);
00 00 00 14 00 00 00 01 00 01 e0 f3
```

Group CMObject/Basic requires CMObject/Primitive.
ZZ, QQ, Zero, Rational, Indeterminate ∈ CMObject/Basic.

Zero : (CMO_ZERO)
— ユニバーサルな ゼロを表す.

ZZ : (CMO_ZZ, $int32$ f, $byte$ a[1], ..., $byte$ a[|f|])
: — bignum をあらわす. a[i] についてはあとで説明

QQ : (CMO_QQ, $int32$ m, $byte$ a[1], ..., $byte$ a[|m|], $int32$ n, $byte$ b[1], ..., $byte$ b[|n|])
— 有理数 $a/b$ を表す.

Rational : (CMO_RATIONAL, $CMObject$ a, $CMObject$ b)
— $a/b$ を表す.

Indeterminate : (CMO_INDETERMINATE, $Cstring$ v)
— 変数名 $v$ .

Indeterminate は変数名をあらわす. v はバイト列であればなにを用いてもよいが, システム毎に変数名として用いられるバイト列は制限がある. 各システム xxx は任意の文字列を各システム固有の変数名へ 1 対 1 に変換できるように実装しないといけない. (これを Dx は #dx と変換するなどの escape sequence を用いて実現するのは, 無理があるようである. テーブルを作成する必要があるであろう.)

## 10.1 Indeterminate および Tree

Group CMObject/Tree requires CMObject/Basic.
Tree, Lambda ∈ CMObject/Tree.

Tree : (CMO_TREE, $Cstring$ name, $List$ attributes, $List$ leaves)
— 名前 name の定数または関数. 関数の評価はおこなわない.
— attributes は空リストでなければ name の属性を保持している.
— 属性リストは, key と 値のペアである.

Lambda : (CMO_LAMBDA, $List$ args, $Tree$body)
— body を args を引数とする関数とする.

数式を処理するシステムでは, Tree 構造が一般にもちいられる. たとえば, $\sin(x+e)$ は, (sin, (plus, x, e)) なる Tree であらわすのが一般的である. Tree の表現を スタックマシンのレベルでおこなうとすると, ox_BEGIN_BLOCK, ox_END_BLOCK で評価を抑制するのが一つの方法である (cf. Postscript の { , } ). たとえば上の方法では x, e, plus, sin を begin block, end block でかこめばよろしい. われわれはスタックマシンの実装をなるべく簡単にするという立場をとりたい, また数学オブジェクトを OX スタックマシンと CMObject を混在して表現したくない. したがって, Tree 構造は Open Math 風の表現をもちいた CMO を導入することにした. またこのほうが, われわれの想定するシステム xxx において, Open XM 対応がはるかに容易である. なお, Tree は, Open Math では, Symbol, Application のメカニズムに相当する.

Lambda は関数を定義するための関数である. Lisp の Lambda 表現と同じ. 例: $sin(x+e)$ の表現.

```
(CMO_TREE, (CMO_STRING, "sin"),
    (CMO_LIST,[size=]1,(CMO_LIST,[size=]2,(CMO_STRING, "cdname"),
                                          (CMO_STRING,"basic")))
    (CMO_LIST,[size=]1,
        (CMO_TREE, (CMO_STRING, "plus"), (CMO_STRING, "basic"),
            (CMO_LIST,[size=]2, (CMO_INDETERMINATE,"x"),
                 (CMO_TREE,(CMO_STRING, "e"),  自然対数の底
    (CMO_LIST,[size=]1,(CMO_LIST,[size=]2,(CMO_STRING, "cdname"),
                                          (CMO_STRING,"basic")))
        ))
```

```
    )
)
```

Leave の成分には, 多項式を含む任意のオブジェクトがきてよい.
Example:

```
sm1> [(plus) [[(cdname) (basic)]] [(123).. (345)..]] [(class) (tree)] dc ::
Class.tree [$plus$ , [[$cdname$ , $basic$ ]], [ 123 , 345 ]  ]
```

Example:

```
asir
[753] taka_cmo100_xml_form(quote(sin(x+1)));
<cmo_tree>  <cmo_string>"sin"</cmo_string>
 <cmo_list><cmo_int32 for="length">1</cmo_int32>
   <cmo_list><cmo_int32 for="length">2</cmo_int32>
     <cmo_string>"cdname"</cmo_string>
     <cmo_string>"basic"</cmo_string>
   </cmo_list> </cmo_list>
<cmo_tree>    <cmo_string>"plus"</cmo_string>
  <cmo_list><cmo_int32 for="length">1</cmo_int32>
    <cmo_list><cmo_int32 for="length">2</cmo_int32>
      <cmo_string>"cdname"</cmo_string>
      <cmo_string>"basic"</cmo_string>
    </cmo_list> </cmo_list>
 <cmo_indeterminate> <cmo_string>"x"</cmo_string>  </cmo_indeterminate>
 <cmo_zz>1</cmo_zz>
</cmo_tree></cmo_tree>
```

次に, 分散表現多項式に関係するグループを定義しよう.

Group CMObject/DistributedPolynomials requires CMObject/Primitive, CMObject/Basic.
Monomial, Monomial32, Coefficient, Dpolynomial, DringDefinition, Generic DMS ring, RingByName, DMS of N variables ∈ CMObject/DistributedPolynomials.

| | | |
|---|---|---|
| Monomial | : | Monomial32 \| Zero |
| Monomial32 | : | (CMO_MONOMIAL32, $int32$ n, $int32$ e[1], ..., $int32$ e[n], Coefficient) |
| | | — e[i] で, $n$ 変数 monomial $x^e = x_1^{e_1} \cdots x_n^{e_n}$ の各指数 $e_i$ をあらわす. |
| Coefficient | : | ZZ\|Integer32 |
| Dpolynomial | : | Zero |
| | | \| (CMO_DISTRIBUTED_POLYNOMIAL, $int32$ $m$, DringDefinition, [Monomial32\|Zero], {Monomial32}) |

— m はモノミアルの個数である.

DringDefinition : DMS of N variables
| RingByName
| Generic DMS ring
— 分散表現多項式環の定義.

Generic DMS ring : (CMO_DMS_GENERIC) — 新版はこちら

RingByName : (CMO_RING_BY_NAME, $Cstring$ s)
— 名前 s で, 格納された ring 定義.

DMS of N variables : (CMO_DMS_OF_N_VARIABLES,
(CMO_LIST, $int32$ m, $Integer32$ n, $Integer32$ p
[, $object$ s, $Cstring$ c, $List$ vlist, $List$ wvec, $List$ outord])
— m はあとに続く要素の数
— n は変数の数, p は 標数
— s は ring の名前
— c は係数環, QQ, ZZ の場合は文字列で QQ, ZZ と書く.
— vlist は Indeterminate のリスト (新版). 多項式環の変数リスト
— wvec は order をきめる weight vector,
— outord は出力するときの変数順序.

RingByName や DMS of N variables はなくても, DMS を定義できる. したがって, これらを実装してないシステムで DMS を扱うものがあってもかまわない.

以下, 以上の CMObject にたいする, xxx = asir, kan の振舞いを記述する.

## 10.2 Zero

CMO では ゼロの表現法がなんとおりもあることに注意.

## 10.3 整数 ZZ

```
#define     CMO_ZZ          20
```

この節では Open xxx 規約における任意の大きさの整数 (bignum) の扱いについて説明する. Open XM 規約における多重精度整数を表すデータ型 CMO_ZZ は GNU MP ライブラリなどを参考にして設計されていて, 符号付き絶対値表現を用いている. (cf. kan/sm1 の配布ディレクトリのなかの plugin/cmo-gmp.c) CMO_ZZ は次の形式をとる.

| int32 CMO_ZZ | int32 $f$ | int32 $b_0$ | $\cdots$ | int32 $b_n$ |
|---|---|---|---|---|

$f$ は 32bit 整数である. $b_0, \ldots, b_n$ は unsigned int32 である. $|f|$ は $n+1$ である. この CMO の符号は $f$ の符号で定める. 前述したように, 32bit 整数の負数は 2 の補数表現で表される.

Open xxx 規約では上の CMO は以下の整数を意味する. ($R = 2^{32}$)

$$\mathrm{sgn}(f) \times (b_0 R^0 + b_1 R^1 + \cdots + b_{n-1} R^{n-1} + b_n R^n).$$

例: int32 を network byte order で表現しているとすると, 例えば, 整数 14 は CMO_ZZ で表わすと,

$$(\mathrm{CMO_ZZ},\ 1,\ 0,\ 0,\ 0,\ \mathrm{e}),$$

と表わす. これはバイト列では

```
00 00 00 14 00 00 00 01 00 00 00 0e
```

となる.

なお ZZ の 0 ( (ZZ) 0 と書く ) は, (CMO_ZZ, 00,00,00,00) と表現する.

## 10.4 分散表現多項式 Dpolynomial

環とそれに属する多項式は次のような考えかたであつかう.

Generic DMS ring に属する元は, 変数を $n$ 個持つ 適当な係数集合 $K$ を持つ多項式環 $K[x_1, \ldots, x_n]$ の元である. 係数集合 $K$ がなにかは, 実際データを読み込み, Coefficient を見た段階でわかる. この環に属する多項式を CMO 形式でうけとった場合, 各サーバはそのサーバの対応する Object に変換しないといけない. この変換の仕方は, 各サーバ毎にきめる.

Asir の場合は, $K[x_1, \ldots, x_n]$ の元は分散表現多項式に変換される.

kan/sm1 の場合は事情は複雑である. kan/sm1 は, Generic DMS ring にあたる クラスをもたない. つまり, Default で存在する, $n$ 変数の分散表現多項式環は存在しないわけである. したがって, kan/sm1 では, DMS of N variables が来た場合, これを CurrentRing の元として読み込む. CurrentRing の変数の数が $n'$ で, $n' < n$ だと新しい多項式環を生成してデータを読み込む. Order その他の optional 情報はすべて無視する.

DMS の 2 番目のフィールドで, Ring by Name を用いた場合, 現在の名前空間で変数 yyy に格納された ring object の元として, この多項式を変換しなさいという意味になる. kan/sm1 の場合, 環の定義は ring object として格納されており, この ring object を 変数 yyy で参照することにより CMO としてうけとった多項式をこの ring の元として格納できる.

**Example**: (すべての数の表記は 16 進表記)

```
Z/11Z [6 variables]
(kxx/cmotest.sm1) run
[(x,y) ring_of_polynomials ( ) elimination_order 11 ] define_ring ;
(3x^2 y). cmo /ff set ;
[(cmoLispLike) 1] extension ;
ff ::
Class.CMO CMO StandardEncoding: size = 52, size/sizeof(int) = 13,
tag=CMO_DISTRIBUTED_POLYNOMIAL

  0  0  0 1f  0  0  0  1  0  0  0 18  0  0  0 13  0  0  0  6
  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  1
```

```
  0  0  0  0  0  0  0  2  0  0  0  3

ff omc ::
 (CMO_DISTRIBUTED_POLYNOMIAL[1f],[size=]1,(CMO_DMS_GENERIC[18],),
  (CMO_MONOMIAL32[13],3*x^2*y),),
```

$3x^2y$ は 6 変数の多項式環の 元としてみなされている.

## 10.5 再帰表現多項式の定義

```
#define CMO_RECURSIVE_POLYNOMIAL        27
#define CMO_POLYNOMIAL_IN_ONE_VARIABLE  33
```

Group CMObject/RecursivePolynomial requires CMObject/Primitive, CMObject/Basic.
Polynomial in 1 variable, Coefficient, Name of the main variable, Recursive Polynomial, Ring definition for recursive polynomials ∈ CMObject/RecursivePolynomial

Polynomial in 1 variable : (CMO_POLYNOMIAL_IN_ONE_VARIABLE, *int32* m,
Name of the main variable ,
{ *int32* e, Coefficient })
— m はモノミアルの個数.
— e, Coefficieint はモノミアルを表現している.
— 順序の高い順にならべる. 普通は巾の高い順.
— e は 1 変数多項式の巾をあらわす.

Coefficient : ZZ| QQ | integer32 | Polynomial in 1 variable
| Tree | Zero | Dpolynomial

Name of the main variable : *int32* v
— v は 変数番号 (0 からはじまる) を表す.

Recursive Polynomial : ( CMO_RECURSIVE_POLYNOMIAL,
RringDefinition,
Polynomial in 1 variable | Coefficient)

RringDefinition : *List* v
— v は, 変数名 (indeterminate) または Tree のリスト.
— 順序の高い順.

Example:

```
(CMO_RECURSIEVE_POLYNOMIAL, ("x","y"),
(CMO_POLYNOMIAL_IN_ONE_VARIABLE, 2,      0,  <--- "x"
  3, (CMO_POLYNOMIAL_IN_ONE_VARIABLE, 2, 1,  <--- "y"
```

```
      5, 1234,
      0, 17),
  1, (CMO_POLYNOMIAL_IN_ONE_VARIABLE, 2, 1,  <--- "y"
      10, 1,
      5, 31)))
```

これは,

$$x^3(1234y^5+17)+x^1(y^{10}+31y^5)$$

をあらわす.

```
sm1
sm1>(x^2-h). [(class) (recursivePolynomial)] dc /ff set ;
sm1>ff ::
Class.recursivePolynomial h * ((-1)) + (x^2  * (1))
```

## 10.6 CPU 依存の double

```
#define CMO_64BIT_MACHINE_DOUBLE   40
#define CMO_ARRAY_OF_64BIT_MACHINE_DOUBLE  41
#define CMO_128BIT_MACHINE_DOUBLE   42
#define CMO_ARRAY_OF_128BIT_MACHINE_DOUBLE  43
```

Group CMObject/MachineDouble requires CMObject/Primitive.
64bit machine double, Array of 64bit machine double 128bit machine double,
Array of 128bit machine double ∈ CMObject/MachineDouble

64bit machine double : (CMO_64BIT_MACHINE_DOUBLE,
*byte* s1 , . . . , *byte* s8)
— s1, . . ., s8 は double (64bit).
— この表現は CPU 依存である.
byte order negotiation を用いる.

Array of 64bit machine double : (CMO_ARRAY_OF_64BIT_MACHINE_DOUBLE, *int32* m,
*byte* s1[1] , . . . , *byte* s8[1], . . . , *byte* s8[m])
— s*[1], . . . s*[m] は m 個の double (64bit) である.
— この表現は CPU 依存である.
byte order negotiation を用いる.

128bit machine double : (CMO_128BIT_MACHINE_DOUBLE,
*byte* s1 , . . . , *byte* s16)
— s1, . . ., s16 は long double (128bit).
— この表現は CPU 依存である.
byte order negotiation を用いる.

Array of 128bit machine double : (CMO_ARRAY_OF_128BIT_MACHINE_DOUBLE, *int32* m,
*byte* s1[1] , . . . , *byte* s16[1], . . . , *byte* s16[m])
— s*[1], . . . s*[m] は m 個の long double (128bit) である.
— この表現は CPU 依存である.
byte order negotiation を用いる.

```
#define CMO_BIGFLOAT   50
#define CMO_IEEE_DOUBLE_FLOAT 51
```

IEEE 準拠の float については, IEEE 754 double precision floating-point format (64 bit) の定義を見よ.

256.100006 の Intel Pentium の double64 での内部表現は `cd 0c 80 43`
256.100006 の PowerPC (Mac) の double64 での内部表現は `43 80 0c cd` . この例でみるように byte の順序が逆である. エンジンスタートの時の byte order negotiation で byte の順序を指定する.
Group CMObject/Bigfloat requires CMObject/Primitive, CMObject/Basic.
Bigfloat ∈ CMObject/Bigfloat

Bigfloat : (CMO_BIGFLOAT,
*ZZ* a , *ZZ* e)
— $a \times 2^e$.

# 11 OX Local Data

システム固有の Object の内部表現は,

```
#define   OX_LOCAL_OBJECT       0x7fcdef30
```

より増えていく数でタグづけする. 0x200000 の領域をわりふることとする. タグはシステムごとに割り振る.

```
#define OX_LOCAL_OBJECT_ASIR   (OX_LOCAL_OBJECT+0)
#define OX_LOCAL_OBJECT_SM1    (OX_LOCAL_OBJECT+1)
```

これらは, OX_DATA の代わりとして使用できる.

# 12 CMO ERROR2

エラー番号 (共通番号)

```
#define  Broken_cmo    1
#define  mathcap_violation  2
```

# 13 メンバの再定義に関する注意

われわれは, Coefficient を Group DistributedPolynomials の メンバとして定義した. これを新しい group のメンバーとして定義を拡張することができる. たとえば, Coefficient を, 再帰表現多項式のメンバとしても定義できる. この場合副作用として, DistributedPolynomials のメンバとしての Coeffients の意味も拡張される. システムがそのあたらしい DistributedPolynomials をすくなくとも CMO としては理解できないといけないが, それを実際にそのシステムのローカルな Object に変換できるかはシステム依存である. このような再定義をゆるすのは, 一般的な実装法がやはりこのような仕組みだからである. mathcap にデータ転送の保護機能はこのような再帰的表現もチェックできないといけない.

# 14 実装, デバッグ, 検証

## 14.1 実装の手順

ソフト xxx を, open XM 対応にするのには以下のような手順をふむと開発が容易であろう.

Step 1. `executeStringByLocalParser` および `popString` の機能を実現して, xxx をライブラリとしてまとめ, 他のソフトとリンクして使用できるかテストする. C での実現の場合 割り込みの取扱に注意を要する. これだけの実現でも, サンプルサーバ (`nullserver00.c`) とリンクすれば, open XM 対応のクライアントと通信できる. クライアント側では, このシステムに対応した機能呼び出しプログラムを書く.

Step 2. 次に, CMO を可能な限り実装する. open sm1 や open asir で, CMO 形式のデータを作成して, 読み込めるかテストする.

Step 2'. `kan/sm1` の plugin として組み込むとサーバの開発が楽かもしれない. `kan/sm1` のソースファイルのディレクトリ `plugin` を見よ.

Step 3. CMO の stream への転送, stream よりの転送は, 巨大データの送信受信にきわめて大切である. これを実装しサンプルサーバとリンクする.

## 14.2 歴史

kan – asir 間でも以上のように開発がすすんだ.

Risa/Asir の開発が沼津の富士フォーラムでおこなわれていたころ, 私が沼津を, 1996 年, 1 月 19 日に訪問し, `Asir_executeString()` の機能を野呂さんに書いてもらって, kan より asir を文字列で呼び出す機能およびその逆を実現したのがことの発端である. たとえば, asir より kan/sm1 の機能を呼び出すには,

```
F = x; G = y;
Ans = kan("(%p).  (%p).  mul toString",F,G)
```

と入力すればよい. x と y の積が kan で解釈実行されて, 結果をもどす. このレベルの結合では kan/sm1 は, 内蔵インタプリタ付のライブラリとして利用できて

ずいぶん便利である. この関数 kan は builtin/parif.c に組み込んだ. asir は pari の関数を組み込んでいるが, この組み込みのメカニズムを利用すると容易にリンクできた. 参照: noro/src/Old1/parif.c.

次に, CMO Primitive の機能を 1997, 5 月, 6 月に実現した. その後, 1997 年 7 月に, SMObject/Basic の実装, 1997 年 7 月には, 野呂がイタリアの CoCoA のワークショップにおいて, 大阿久の b-function を stratification 込みで計算する計算プログラムを asir, kan を連係してデモした. このときは, Sm1_executeStringByLocalParser 関数を用いて, ライブラリとしてリンクしてつかった. 1997 年 11 月に TCP/IP による, サーバスタックマシン間の通信の実装をおこなっている. 通信の実装テストのために, Java および C で null server , null client を作成した. 以下, これにつき述べる.

## 14.3 サンプルサーバ, クライアント

Open XM では, 現在のところ, サンプルサーバとして oxserver00.c を提供している. このサーバをもとにして, asir および kan/sm1 の open XM サーバを試験中である (ox_asir, ox_sm1). ox_sm1 では, sm1stackmachine.c が open XM スタックマシンを実現している. サンプルクライアントは, ネットワークにデータを送出および受信する機能のみをもつ, testclient.c を提供している. asir および kan/sm1 には本格的なクライアント機能 (open XM サーバを呼び出す機能) を組み込んである. サーバを起動するプログラムは, kan/sm1 グループでは, ox なる名前で, asir グループでは, ox_lauch なる名前であるが, 機能は同じである. ox のソースは oxmain.c である.

### 14.3.1 OpenXM/src/ox_toolkit にあるサンプル実装

このディレクトリの解説文書を見よ.

### 14.3.2 ox_null

Primitive のスタックマシンのソケットによる実装. スタックマシンは nullstackmachine.c に実装されており, oxserver00.c にリンクしてサーバとなる. サンプルサーバであり, これに CMO Primitive 仕様の関数を結合すれば, 一応 サーバが動作するはずである. スタックには,CMO の Primitive の object へのポインタがそのまま push される. コントロール機能なし. 1997/11/29 版よりコントロール機能追加. @Old/nullserver00.c.19971122c, @Old/mytcpip.c.19971122c

現在はこのサーバはメンテナンスされていない (object 関係の関数を追加しないとコンパイルできない.)

### 14.3.3 testclient

Java による実装: @Old/client.java.19971122c これも現在はふるい. OX パケットのシリアル番号に対応していない. ちかいうちに改訂する予定. executeString および popString を要請する能力しか持たない. 受信は スレッド listner がおこなっている. 受信は byte データを表示するのみである. スレッドの優先度をうまくバランスとらないと, 受信データがあるのに表示しなかったりする.

C による testclient 同じような機能をもつプログラムの実装もある.

```
./ox -ox ox_sm1 -host localhost -data 1300 -control 1200  (サーバの立ち上げ)
./testclient     (testclient の立ち上げ)
```

これも現在はふるい.

田村 (tamura@math.kobe-u.ac.jp) による Java への新しい実装がある (1999, 3 月, 神戸大学修士論文).

### 14.3.4 ox

ox は ox サーバをたちあげるためのプログラムである. クライアントよりサーバへ接続するには二つの方法がある. 一つは ox で データとコントロール用の二つのポート番号を明示的に起動し, クライアントがこのポートへつなぎにいく方法である. もう一つは, クライアント側でまず, 空いているポートを二つさがし, それから ox を -reverse で起動してサーバ側がクライアントにつなぎにくる方法である. この場合, ox はたとえば次のように起動される.

```
/home/nobuki/kxx/ox -reverse -ox /home/nobuki/kxx/ox_sm1
-data 1172 -control 1169 -pass 1045223078
```

ox は, 子どもプロセスとして, ox_asir, ox_sm1 などを起動するのであるが, これらのプロセスは 3 より OX データ, コマンドを読み込み, 4 へ OX データを書き出す. 現在の実装では 3 と 4 は dup して同一視してしまっている. ox は TCP/IP のデータ転送のポートを, 3, 4 へわりあてて, 子どもプロセスを起動する.

```
close(fdControl);   /* close(0); dup(fdStream); */
dup2(fdStream,3);
dup2(fdStream,4);
```

### 14.3.5 ox_asir phrase book

[ この節の記述は古い] CMObject と asir の object は次の規則にしたがって変換される. なお asir の object のタグをみるには関数 type を用いる.

1. Null : 0 として使用される.
2. Integer32 : 内部的にのみ使用される. 送出は, ZZ に変換される. -1 は (unsigned int) -1 に変換されてから, ZZ に変換されるので, 正の数となる.
3. Cstring : 文字列 (type = 7) に変換される.
4. ZZ : 数 (type = 1 ) に変換される.
5. QQ : 数 (type = 1 ) に変換される.
6. List : リスト (type = 4) に変換される.
7. Dpolynomial : 分散表現多項式 (type = 9) に変換される. order はうけとったモノミアルのリストと同じ order である.
8. RecursivePolynomial : 再帰表現多項式に変換される. 内部順序に自動変換される.
9. Indeterminate : 不定元に変換される.

記述のない CMObject に関しては, 利用できない (cf. mathcap ).
問題点: 0 の扱いの仕様がまださだまっていない. Null が数 (type = 1) の 0 に変換される版もある.

例: 分散表現多項式の $x^2 - 1$ の因数分解を asir にやってもらう OXexpression の列をあげる.

```
(OX_DATA, (CMO_LIST, 4, CMO_DMS,CMO_DMS_GENERIC,
(CMO_MONOMIAL32,1,2,(CMO_ZZ,1)),
(CMO_MONOMIAL32,1,0,(CMO_ZZ,-1)))),
(OX_DATA, (CMO_INT32,1))
(OX_DATA, (CMO_STRING,"ox_dtop"))
(OX_COMMAND,(SM_executeString))

(OX_DATA, (CMO_INT32,1))
(OX_DATA, (CMO_STRING,"fctr"))
(OX_COMMAND,(SM_executeString))


(OX_DATA, (CMO_INT32,1))
(OX_DATA, (CMO_STRING,"ox_ptod"))
(OX_COMMAND,(SM_executeString))

(OX_COMMAND,(SM_popCMO))
```

ここで, ZZ の元を普通の整数表記であらわした. `dtop1` および `ptod1` はそれぞれ, 分散表現多項式を, Asir の再帰表現多項式に, 逆に, Asir の再帰表現多項式を, 分散表現多項式に変換する, ユーザ定義の 1 引数関数である. これらの関数は Asir の リストにも作用させることが可能であり, その場合は要素としてでてくる, 分散表現多項式 または Asir の再帰表現多項式を必要な形に変換する. `fctr` は因数分解をする組み込み関数である.

`kxx/oxasir.asir` のソース.

```
OxVlist = [x,y,z]$

def ox_ptod(F) {
extern OxVlist;
if (type(F) == 4) return(map(ox_ptod,F));
else if (type(F) == 2) return(dp_ptod(F,OxVlist));
else return(F);
}

def ox_dtop(F) {
extern OxVlist;
if (type(F) == 4) return(map(ox_dtop,F));
else if (type(F) == 9) return(dp_dtop(F,OxVlist));
else return(F);
}

end$
```

### 14.3.6 ox_sm1 phrase book

[ この節の記述は古い] CMObject と kan/sm1 の object は次の規則にしたがい変換される. なお, kan/sm1 の object のタグをみるには, `tag` または `etag` を用いる.

1. Error2 : Error (etag = 257) に変換される.
2. Null : null (tag = 0) に変換される.
3. Integer32 : integer (tag = 1) に変換される.
4. Cstring : 文字列 (type = 5) に変換される.
5. ZZ : universalNumber (type = 15 ) に変換される.
6. QQ : rational (tag = 16 ) に変換される.
7. List : array (tag = 6) に変換される.
8. Dpolynomial : 多項式 (tag = 9) に変換される.

注意: `ReverseOutputOrder = 1` (標準) のとき, `xn, ..., x0, dn, ..., d0` の順番で (`show_ring` の形式) Dpolynomial に変換される (印刷形式だと, `xn` は `e`, `d0` は `h`, `x0` は `E`, `dn` は `H`). たとえば, `ox_send_cmo(id,<<1,0,0,0,0,0>>)` は, `x2` に変換される. `ox_send_cmo(id,<<0,0,1,0,0,0>>)` は, `x0` に変換される.
`OxVersion` 変数で openXM のプロトコルの version を表す.

### 14.3.7 `ox_sm1` を用いたクライアントのテスト方法

まだかいてない.

### 14.3.8 `Asir` を用いたサーバのテスト方法

## 14.4 最小の TCP/IP クライアントの例

Java または M2 によるソースコードを掲載の予定.

## 14.5 クライアント asir, sm1

sm1 については, ox.sm1 , oxasir.sm1 がクライアントパッケージ. `ox`, `ox_asir`, `ox_sm1` の存在するパス, および sm1 より呼び出すための asir の関数定義である `oxasir.asir` のあるパスをこれらのパッケージに書き込んでおく必要がある.
`ox_asir` は, `asir` なる名前でよばれると asir として動作し, `ox_asir` なる名前でよばれると, open XM サーバとして動作する. `/usr/local/lib/asir` または `ASIR_LIBDIR` へ `ox_asir` 実体をおき, `ox_launch` をおなじディレクトリへ `ox_asir` へのシンボリックリンクとして作成する. コマンドサーチパスにあるディレクトリへ `asir` を `ox_asir` へのシンボリックリンクとして作成する.

```
This is Asir, Version 990413.
Copyright (C) FUJITSU LABORATORIES LIMITED.
3 March 1994. All rights reserved.
0
[324] ox_launch(0,"/usr/local/lib/asir/ox_asir");
1      <=== これがサーバの番号.
[326] ox_execute_string(1,"fctr(x^10-1);");
0
[327] ox_pop_local(1);
```

```
[[1,1],[x-1,1],[x+1,1],[x^4+x^3+x^2+x+1,1],[x^4-x^3+x^2-x+1,1]]
[328] ox_execute_string(1,"dp_ptod((x+y)^5,[x,y]);");
0
[329] ox_pop_cmo(1);
(1)*<<5,0>>+(5)*<<4,1>>+(10)*<<3,2>>+(10)*<<2,3>>+(5)*<<1,4>>+(1)*<<0,5>>
[330] ox_rpc(1,"fctr",x^10-y^10);
0
[331] ox_pop_local(1);
[[1,1],[x^4-y*x^3+y^2*x^2-y^3*x+y^4,1],[x^4+y*x^3+y^2*x^2+y^3*x+y^4,1],[x+y,1],[x-y,1]]
[332] ox_rpc(1,"fctr",x^1000-y^1000);   ox_cmo_rpc もあり.
0
[333] ox_flush(1);
1
[334] ox_pop_local(1);
0

ox_sync(1);   --- sync ball を送る.
```

## 14.6 開発中のサーバ, クランアント

Mathematica サーバ, クライアント : 小原. Java クライアント, Open Math proxy : 田村. Gnuplot サーバ, Macaulay 2 クライアント, サーバ, その他小さいソフト (Toric GB, Grobner fan) のサーバ, Algebraic equation solver by numerical method: 高山. Open Asir マニュアル, サンプル: 小原, 高山.

数学的におもしろい問題を実際にあつかってみないとわからない問題点もおおくあると思う. 現在, $\mathcal{A}$-超幾何関数の解のグラフ表示, パラメータ付積分のグラフ表示のソフトをかくことでさらに問題点をさぐることを計画している. グレブナ deformation による多項式解, 有理解の導出 (線形および非線形方程式, 非線形微分方程式から出発すると代数方程式を解く問題になる) は OpenXM 的なプログラムのおもしろい練習問題. Java の並列計算記述能力をつかって ox サーバを使うのもおもしろい.

世界中の人につかってもらえる規格にするにはまだまだ実験と経験をかさねないといけない. Free Mathematical Software Initiative を作るべきだろう.

## 14.7 Change log

1. 1997/11/20 : この document の最初の version が生まれた. kxx/openxxx.tex なる名前であった.
2. 1999/07 : CMO_ZZ の形式を変えた.
3. 1999/09/7, 9/8 : 分散表現多項式, Mathcap, RecursivePolynomial, の形式を変えた. asir, sm1 に実装した. エラー処理のために, dupErrors, getsp を SM コマンドに加えた.

## 14.8

Java で sm1 サーバをすべて書いてみるか?

# 15 どのように協調機能にすぐれた Math Soft を書くのか?

いくつかの経験的なプログラム作法をメモとしてまとめておく.

1. 文字列を読み込み, 計算して, また文字列を結果として出力するプログラムおよび関数は協調機能にすぐれたプログラムの基本である. グラフィカルインターフェースのみのプログラムは他のソフトと協調させるのがむずかしい.

2. Output を読むのは人間とは限らない. Output の読者は, 他のソフトかもしれない. したがって, Output もやはり, 文法にしたがって出力すべきである. エラー出力も文法にしたがうべきである.

## 15.1 open gnuplot の実装

書いてない.

## 15.2 open phc の実装

書いてない.

## 15.3 open M2 の実装

書いてない.

この節の補足は, `this/open/ohp` を見よ (Open XM day, 1999, 7/12 Mon の原稿)

# 16 新しい CMO の登録

## 16.1 新しい CMO を作るときの約束

CMO では, 再帰的なデータ表現をゆるしているので, Object を増やした場合, もとからある CMO の動作も拡張される. そのシステムが正しく動作するのか確認する必要がある. 計算することはできなくても, 通信の中断なしに, データの受渡しをできないといけない.

## 16.2 OpenXM プロジェクトに参加するには?

OpenXM にパッケージを加えてくれるのは大歓迎です. takayama@math.kobe-u.ac.jp に連絡下さい. 新しい CMO を必要なら加えて下さい. 加えた場合は

1. 定義した CMObject の形式的な定義と説明.

2. システム xxx が, この CMObject に関してどのように振舞うかの説明.

3. この CMObject, システム xxx に関する URL.

をおくって下さい. 必要ならディスカッションをおこない, 確定版の仕様を作成し, 確定した段階で CMO のタグを発行し, この CMObject に関連する URL を OpenXM ホームページ [5] よりリンクします.

# 17 OX サーバに対する C ライブラリインタフェース

一部の OX サーバでは C のライブラリとしてリンクして使用するもできる. ライブラリとして使用するための C の関数は Asir の OX サーバ用クライアント関数に似たインタフェースを持つ.

ライブラリ関数には, CMO を binary encoding して渡す. ライブラリ関数からは, CMO が binary encoded form で戻る.

```
int xxx_ox_init(int type)
```

この関数はライブラリインタフェースの初期化を行う. OX サーバ xxx へ int32 を送るための byte order を type で指定する. type = 0 の場合 native byte order が設定される. type = 1 の場合 network byte order が設定される. 失敗した場合, -1 を戻す.

```
void xxx_ox_push_cmo(void *cmo)
```

Binary encoded された CMO を, OX サーバ xxx の stack に push する.

```
int xxx_ox_pop_cmo(void cmo, int limit)
```

Binary encoded された CMO (サイズは size) を, OX サーバより pop して, cmo に書き込む. 戻り値は CMO のサイズを byte で戻す. Stack underflow の場合の戻り値は 0 である. ただし, limit より大きいサイズの CMO は書き込まれない. この場合, 戻り値は -1 となる.

```
int xxx_ox_peek_cmo_size()
```

stack の一番上にある CMO のサイズを byte で戻す.

```
void xxx_ox_push_cmd(int cmd)
```

(OX_COMMAND,int32 cmd) をサーバに送る.

```
void xxx_ox_execute_string(char *s)
```

文字列 s をサーバに実行させる. s はサーバのパーザが受理可能なものでなければならない.

ライブラリモードのサンプルソースは OpenXM/doc/oxlib にある.

## References


[1] Gray, S., Kajler, N. and Wang, P. S., Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions, *Journal of Symbolic Computation*, 1996.

[2] Linton, S. and Solomon, A., OpenMath, IAMC and `GAP`, preprint, 1999.

[3] Noro, M. et al., A Computer Algebra System `Risa/Asir`, 1993, 1995, 2000 ftp://archives.cs.ehime-u.ac.jp/pub/asir2000/

[4] `http://www.openmath.org`

[5] `http://www.math.kobe-u.ac.jp/OpenXM/`

[6] Ohara, Takayama, Noro: Introduction to Open Asir , 1999, (in Japanese), Suushiki-Shyori, Vol 7, No 2, 2–17. (ISBN4-87243-086-7, SEG , Tokyo).

[7] Takayama, N., *Kan: A system for computation in algebraic analysis,* 1991 version 1, 1994 version 2, the latest version is 2.991106. ftp://ftp.math.kobe-u.ac.jp/pub/kan

[8] Verschelde, J., PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. ACM Transaction on Mathematical Softwares, 25(2) 251-276, 1999.

[9] Wang, P., Design and Protocol for Internet Accessible Mathematical Computation. Technical Report ICM-199901-001, ICM/Kent State University, 1999.

[10] XML `http://www.w3c.org`



Masayuki Noro,
FUJITSU LABORATORIES LTD., Kawasaki, Japan; (by Aug., 2000)
`noryo@flab.fujitsu.co.jp`
Current Address: Department of Mathematics, Kobe University, Rokko, Kobe, 657-8501, Japan;
`noro@math.kobe-u.ac.jp`

Nobuki Takayama,
Department of Mathematics, Kobe University, Rokko, Kobe, 657-8501, Japan;
`takayama@math.kobe-u.ac.jp`